NodeBrain Tutorial

Release 0.8.17

NodeBrain Tutorial
August 2014
NodeBrain Open Source Project

Release 0.8.17

Author: Ed Trettevik

History

2005-10-12 Title: *NodeBrain Tutorial*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

Version 0.6.3 (not released)

- Operating mode tutorial

2006-04-04 Release 0.6.4

- Included interface examples of release 0.6.4 enhancement.
- Included cell expression and ignorance tutorials.

2007-01-03 Version 0.6.6 (not released)

- Minor corrections

2007-07-03 Version 0.6.8 (not released)

- Terminology change: "node" replaces "expert"

2008-08-20 Version 0.7.1 (not released)

- Updated with 0.7 syntax changes (see *NodeBrain Language Reference*)
- Included translator, servant, pipe, audit, peer, and webster node tutorials

2009-02-05 Release 0.7.4

- Included tree, cache, syslog, and snmptrap node tutorials

2009-03-04 Release 0.7.5

- Refined for PDF
- Several minor corrections

2010-12-31 Release 0.8.3

- Moved the Webster tutorial to the Webster NodeBrain Module manual.

2012-06-03 Version 0.8.10 (Not released)

- Corrections by Cheryl Hauser

2012-12-30 Release 0.8.13

- Included additional examples of rule actions and shell commands.

Preface

This tutorial 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 in this tutorial 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

Caboodle NodeBrain Kit - A framework for managing rules
System NodeBrain Kit - A small sample application

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 Invoking NodeBrain

This tutorial illustrates how to invoke NodeBrain and introduces you to modes of operation that enable NodeBrain to adapt in minor ways to the requirements of different components within a monitoring application. Some basic concepts about the language syntax and semantics are introduced along the way.

1.1 Interactive Mode

The easiest way to get started with NodeBrain is to invoke it interactively, type some commands, and watch it respond. This tutorial walks you through some simple interactive sessions.

1.1.1 Starting and Ending an Interactive Session

If you execute NodeBrain with no arguments, it starts in interactive mode and presents a prompt. At a Unix or Linux shell command prompt, you start NodeBrain by typing `nb`. It looks like this.

```
$ nb

N o d e B r a i n   0.8.17 (Dopey) 2014-06-10

Compiled Jun 12 2014 19:20:12 x86_64-unknown-linux-gnu

Copyright (C) 2014 Ed Trettevik <eat@nodebrain.org>
MIT or NodeBrain License
----------------------------------------------------------------

nb

Date       Time     Message
---------- -------- ------------------------------------------
2014/06/12 19:56:02 NB000I NodeBrain nb[10656] myuser@myhost
2014/06/12 19:56:02 NB000I Reading from standard input.
---------- --------

>
```

Since the information NodeBrain displays when starting up is not important to the topics covered in this tutorial, I don't include it in the examples. The example above is shown as this.

```
$ nb
>
```

At a Windows command prompt, it looks like this.

```
C:> nb
>
```

For the examples in this tutorial, use a Unix-style shell command prompt.

To end an interactive session, just tell NodeBrain you want to quit.

```
$ nb
> quit
$
```

You may also use Cntl-D or Cntl-C to end an interactive session.

1.1.2 Assertions

> *The partisan, when he is engaged in a dispute, cares nothing about the rights of the question, but is anxious only to convince his hearers of his own assertions.* —Plato (427 BC–347 BC), *Dialogues, Phaedo*

It is quite easy to convince NodeBrain of your own assertions. It looks like this.

```
$ nb
> assert a=1,b=2,mood="happy";
```

To verify that NodeBrain is willing to accept your assertions, ask it to show you what it now believes about a, b, and mood:

```
> show a
a = 1
> show b
b = 2
> show mood
mood = "happy"
> quit
```

1.1.3 Rules

> *Hell, there are no rules here—we're trying to accomplish something.* —Thomas A. Edison (1847–1931)

If you are trying to accomplish something with NodeBrain and there are no rules, you are using the wrong tool. NodeBrain is a rule engine, not a general purpose shell or scripting language. There are many good shells and scripting languages to choose from, and much to be accomplished without rules—at least not the kind of rules we are talking about here. In a more general sense, no computing exists without rules, but the kind of rules you find in NodeBrain and similar tools are not appropriate for most computing problems. However, they can be quite handy when applied to the right problems.

Let's start with a rule that doesn't accomplish anything.

```
$ nb
> define myFirstRule on(a=1 and b=2);
> assert a=1;
> assert c=3;
> assert b=2;
2008/06/05 12:19:05 NB000I Rule myFirstRule fired
> assert a=0;
> assert c=1;
> assert a=1;
2008/06/05 12:19:21 NB000I Rule myFirstRule fired
> quit
$
```

What just happened there? Well, you defined a rule named `myFirstRule` and specified a condition `(a=1 and b=2)` under which you want it to fire. Each time this condition transitioned to a "true" state, the rule fired.

Let's modify this example to make it accomplish something—perhaps an assertion.

```
$ nb
> define mySecondRule on(a=1 and b=2) mood="happy";
> assert mood="sad";
> show mood
mood = "sad"
> assert a=1,b=2,c=3;
2008/06/05 12:20:48 NB000I Rule mySecondRule fired(mood="happy")
> show mood
mood = "happy"
> quit
$
```

The assertion that a=1, b=2, and c=3 changed the mood from "sad" to "happy." Who said you can't accomplish something by following rules?

1.1.4 Shell Commands

> *She did not talk to people as if they were strange hard shells she had to crack open to get inside. She talked as if she were already in the shell. In their very shell.* —Marita Bonner

From NodeBrain you can get into the host systems command shell by using a command prefix of dash (`-`).

```
$ nb
> -date
[17445] Started: -date
[17445| Thu Jun  512:22:34 PDT 2008
[17445] Exit(0)
> -ls -al kim.nb
[17478] Started: -ls -al kim.nb
[17478| -rw-r--r-- 1 myuser users 52 2006-01-05 11:44 kim.nb
[17478] Exit(0)
> quit
$
```

When you talk as if you are in someone else's shell, you need to speak their language. Here's an example on Windows.

```
C:> nb
> -dir
[1060] Started: -dir
[1060|  Volume in drive C has no label.
[1060|  Volume Serial Number is 0C62-77CF
[1060|
[1060|  Directory of C:\home\nodebrain\nb-0.6.4-source.msdsw\Release
[1060|
[1060| 03/28/2006  05:23 PM    <DIR>          .
[1060| 03/28/2006  05:23 PM    <DIR>          ..
[1060| 03/28/2006  04:26 PM           303,104 nb.exe
[1060| 03/28/2006  04:26 PM            10,163 nb.exp
[1060| 03/28/2006  04:26 PM            17,728 nb.lib
[1060|                3 File(s)      2,942,497 bytes
[1060|                2 Dir(s)  11,946,586,112 bytes free
[1060] Exit(0)
> quit
$
```

You can run a shell command in blocking mode using the `-` symbol, or non-blocking mode using the `=` symbol. The following example illustrates the different using `=echo` and `-cat` commands. The interpreter waits for a blocking shell command to finish before prompting for the next interactive command. It does not wait for a non-blocking shell command to finish.

```
> =echo `date` ": it happened" >> happening.log
2012/12/30 13:30:42 NB000I [18589] Started: =echo `date` ": it happened" >> happening.log
> -cat happening.log
2012/12/30 13:30:49 NB000I [18589] Exit(0)
[18592] Started: -cat happening.log
[18592| Sun Dec 30 13:30:42 PST 2012 : it happened
[18592] Exit(0)
> =echo `date` ": it happened" >> happening.log
2012/12/30 13:30:55 NB000I [18594] Started: =echo `date` ": it happened" >> happening.log
>
2012/12/30 13:30:56 NB000I [18594] Exit(0)
> -cat happening.log
[18597] Started: -cat happening.log
[18597| Sun Dec 30 13:30:42 PST 2012 : it happened
[18597| Sun Dec 30 13:30:55 PST 2012 : it happened
[18597] Exit(0)
```

Try the example above for yourself. After the second `=echo` command, press enter without a command. In interactive mode, the command prompt blocks the interpreter. This explains why the interpreter didn't report the completion of the first `=echo` command until after the subsequent `-cat` command.

By default the output of a non-blocking shell command is not displayed by the interpreter. You can direct the interpreter to display the output of a non-blocking shell command using

`=|`. Try the following example, pressing the enter key once without a command after the `=|echo` command.

```
> =|echo abc;echo def;echo xyz
2012/12/30 13:53:54 NB000I [19490] Started: =|echo abc;echo def;echo xyz
>
2012/12/30 13:53:54 NB000I [19490| abc
2012/12/30 13:53:54 NB000I [19490| def
2012/12/30 13:53:54 NB000I [19490| xyz
```

When you direct NodeBrain to execute a shell command to perform a task, the child process is called a *servant*.

1.1.5 Consulting Servant Scripts

> *CONSULT, v.i. To seek another's disapproval of a course already decided on.*
> — Ambrose Bierce (1842–1914), *The Devil's Dictionary*

NodeBrain is willing to accept the advice of others—even servants. To illustrate, let's first write a simple Perl script named `processCount.pl` that checks to see how many processes are running on a Unix system.

```
#!/usr/bin/perl
chomp($processes=‘ps -e|wc -l‘);
print("assert processes=$processes;\n");
```

Now let's execute it in the Unix command shell.

```
$ ./processCount.pl
assert processes=40;
```

Now you can consult this script from NodeBrain by using a prefix of `-:` instead of just `-`. This tells NodeBrain to accept the output of the shell command as input to the NodeBrain interpreter.

```
$ nb
> -:./processCount.pl
[17537] Started: -:./processCount.pl
[17537: assert processes=40;
[17537] Exit(0)
> show processes
processes = 40
> quit
$
```

To consult a servant using non-blocking mode, use `=:`. This is illustrated by the following example. Since we are in interactive mode, the interpreter blocks waiting for command input, and you must enter empty commands twice to watch the shell command finish.

```
$ nb
> assert !x;
> show x
x = 0
> =:echo "assert x=1";echo "show x";echo "assert x=2"
2012/12/30 13:14:31 NB000I [18218] Started: =:echo "assert x=1";echo "show x";echo "assert x=2"
>
2012/12/30 13:14:31 NB000I [18218: assert x=1
2012/12/30 13:14:31 NB000I [18218: show x
x = 1
2012/12/30 13:14:31 NB000I [18218: assert x=2
>
2012/12/30 13:14:32 NB000I [18218] Exit(0)
> show x
x = 2
```

1.1.6 Rule Actions

> *You ask me why I do not write something...I think one's feelings waste themselves in words, they ought all to be distilled into actions and into actions which bring results.* —Florence Nightingale (1820–1910), in Cecil Woodham-Smith, *Florence Nightingale,* 1951

Any NodeBrain command can be used as a rule action by following the rule condition with a colon (`:`) and the action command.

```
$ nb
> define cmdRule on(a and b): show x;
> assert x=3;
> assert a,b;
2008/06/05 12:27:41 NB000I Rule cmdRule fired
: show x;
x = 3
> quit
$
```

An assertion may be included as part of a rule action following the condition and preceeding the action command.

```
$ nb
> define cmdRule on(a and b) x=3:show x;
> assert a,b;
2012/12/30 14:07:43 NB000I Rule cmdRule fired (x=3)
: show x;
x = 3
> quit
$
```

A rule action may also have an assertion without a command, as illustrated below.

```
$ nb
> define cmdRule on(a and b) x=3;
> assert a,b;
2012/12/30 14:11:19 NB000I Rule cmdRule fired (x=3)
> show x
x = 3
> quit
$
```

1.1.7 Action Servant Scripts

When the action you want to take in response to a condition is not a trivial command, you can often turn it into one by writing a script in your favorite scripting language. I illustrate this in an odd way by writing a trivial Perl script and ask you to imagine that it isn't trivial. Call this script `alarmMe.pl`.

```
alarmMe.pl
#!/usr/bin/perl
$id=shift();
$msg=shift();
$me=getpwuid($<);
system("mailx -s \"$id: $msg\" $me < note$id.txt");
```

Here's a session that uses a rule that invokes the `alarmMe.pl` script as a servant.

```
$ nb
> define alarmRule on(a):-./alarmMe.pl 1 "Things aren't looking good"
> assert a;
2008/06/05 12:31:42 NB000I Rule alarmRule fired
: -./alarmMe.pl 1 "Things aren't looking good"
[17571] Started: -./alarmMe.pl 1 "Things aren't looking good"
[17571] Exit(0)
> quit
$
```

1.1.8 Rule Files (Scripts)

A rule file is just a file that contains NodeBrain rules. By convention, use a `.nb` suffix on these files. You can combine what you've learned in this section into a rule file named `roger.nb`.

```
# roger.nb
define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes"
define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes"
define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
```

When you specify a file as an argument to NodeBrain, it reads the file and processes the commands.

```
$ nb roger.nb
2008/06/05 14:37:14 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2008/06/05 14:37:14 NB000I Source file "roger.nb" included. size=246
2008/06/05 14:37:14 NB000I NodeBrain nb[5393] terminating - exit code=0
$
```

But wait, this is a tutorial on interactive sessions. What happened to the prompt? Remember that NodeBrain assumes you want an interactive session when you don't specify any arguments? Well, when you specify a file, NodeBrain assumes you want a batch session and just processes the file and quits.

To tell NodeBrain you want an interactive session after the file is processed, you need to give it an additional - argument.

```
$ nb roger.nb -
2009/03/16 11:32:35 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2009/03/16 11:32:35 NB000I Source file "roger.nb" included. size=243
2009/03/16 11:32:35 NB000I Argument [2] -
2009/03/16 11:32:35 NB000I Reading from standard input.
---------- --------
> assert a,b=2;
2009/03/16 11:32:48 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[3515] Started: -:./processCount.pl
[3515: assert processes=74;
2009/03/16 11:32:48 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[3519] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[3519] Exit(0)
[3515] Exit(0)
> quit
2009/03/16 11:32:52 NB000I NodeBrain nb[3514] terminating - exit code=0
```

1.2 Batch Mode

Except for the user prompt, everything you learned in the previous section on interactive sessions applies to batch sessions. The rules are the same; you just no longer assume a user is typing commands.

1.2.1 Starting and Stopping a Batch Session

In the previous section, you actually ran a batch session by accident.

```
$ nb roger.nb
2008/06/05 14:42:37 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2008/06/05 14:42:37 NB000I Source file "roger.nb" included. size=246
2008/06/05 14:42:37 NB000I NodeBrain nb[5409] terminating - exit code=0
$
```

You start a batch session by giving `nb` at least one source file without specifying the `-` option. A batch session ends when all source files have been processed.

Create a second source file with some assertions. Call it `abby.nb`.

```
# abby.nb
assert a=1,b=3;
assert b=2;
assert a=0;
assert a=1;
```

Combine the rule file and assertions in a single batch session.

```
$ nb roger.nb abby.nb
2008/06/05 20:35:59 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2008/06/05 20:35:59 NB000I Source file "roger.nb" included. size=251
2008/06/05 20:35:59 NB000I Argument [2] abby.nb
> # abby.nb
> assert a=1,b=3;
> assert b=2;
2008/06/05 20:35:59 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7141] Started: -:./processCount.pl
[7141: assert processes=58;
2008/06/05 20:35:59 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[7145] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[7145] Exit(0)
[7141] Exit(0)
> assert a=0;
> assert a=1;
2008/06/05 20:36:00 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7152] Started: -:./processCount.pl
[7152: assertprocesses=62;
[7152] Exit(0)
2008/06/05 20:36:00 NB000I Source file "abby.nb" included. size=52
2008/06/05 20:36:00 NB000I NodeBrain nb[7140] terminating - exit code=0
$
```

This time the batch session generated an alarm saying "ok - under 100 processes." Do you know why rule `myProcessRule3` fired twice, but rule `myProcessRule2` only fired once? The rule condition for `myProcessRule3` transitioned to a true state twice, while the condition for `myProcessRule2` only transitioned to a true state once. An `on` rule whose condition is true will not fire when a new assertion causes it to remain true.

1.2.2 Piping Commands to a Batch Session

If you give NodeBrain an equals sign (`=`) as an argument, it processes `stdin` like any other file. Do the same thing you did in the previous example using a pipe to `stdin`.

```
$ cat roger.nb abby.nb | nb =
```

You can combine the `=` argument with other source file arguments.

```
$ cat abby.nb | nb roger.nb =
```

Files are processed in the order of arguments. If you change the argument order, the assertions are made before you define the rules, so you don't get the same results.

```
$ cat abby.nb | nb = roger.nb
```

This is like specifying the following command.

```
$ nb abby.nb roger.nb
```

Okay, so you now know how to pipe commands into NodeBrain. Maybe it would be handy to use a script as the source of assertions. Here's a script called `randy.pl` that will generate some pseudo random values for a and b.

```
randy.pl
#!/usr/bin/perl
for($i=1;$i<10;$i++){
  $r=int(rand()*4);
  if($r%2){print("assert a=$r;\n");}
  else{print("assert b=$r;\n");}
  }
```

You can use this script as the source of your assertions by piping the output of the script to NodeBrain.

```
$ ./randy.pl | nb roger.nb =
2008/06/05 20:54:23 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2008/06/05 20:54:23 NB000I Source file "roger.nb" included. size=251
2008/06/05 20:54:23 NB000I Argument [2] =
2008/06/05 20:54:23 NB000I Reading from standard input.
---------- --------
| assert a=3;
| assert a=1;
| assert a=1;
| assert b=0;
| assert b=2;
2008/06/05 20:54:23 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7251] Started: -:./processCount.pl
[7251: assert processes=58;
2008/06/05 20:54:23 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[7255] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[7255] Exit(0)
[7251] Exit(0)
| assert b=0;
| assert b=2;
2008/06/05 20:54:24 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7262] Started: -:./processCount.pl
[7262: assert processes=62;
[7262] Exit(0)
| assert a=1;
| assert b=0;
2008/06/05 20:54:25 NB000I NodeBrain nb[7250] terminating - exit code=0
$
```

You can also pipe from within a script. In `randal.pl`, you open a pipe to NodeBrain and write to the pipe.

```
randal.pl
#!/usr/bin/perl
open(NB,"|nb roger.nb =")||die;
for($i=1;$i<10;$i++){
  $r=int(rand()*4);
  if($r%2){print(NB "assert a=$r;\n");}
  else{print(NB "assert b=$r;\n");}
  }
close(NB);
```

The result is the same.

```
$ ./randal.pl
2008/06/05 21:02:43 NB000I Argument [1] roger.nb
> # roger.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
2008/06/05 21:02:43 NB000I Source file "roger.nb" included. size=251
2008/06/05 21:02:43 NB000I Argument [2] =
2008/06/05 21:02:43 NB000I Reading from standard input.
---------- --------
| assert b=0;
| assert b=0;
| assert a=1;
| assert b=2;
2008/06/05 21:02:43 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7376] Started: -:./processCount.pl
[7376: assert processes=59;
2008/06/05 21:02:44 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[7380] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[7380] Exit(0)
[7376] Exit(0)
| assert a=3;
| assert b=2;
| assert b=0;
| assert a=3;
| assert a=1;
2008/06/05 21:02:44 NB000I NodeBrain nb[7375] terminating - exit code=0
$
```

So, any way you can establish a pipe to NodeBrain is fine, just remember to tell NodeBrain to read `stdin` by specifying `=` as a file argument.

1.3 Servant Mode

> *If thou are a master, be sometimes blind; if a servant, sometimes deaf.* — Thomas Fuller (1608–1661)

When NodeBrain reads commands from `stdin` in batch mode, it uses blocking I/O. This means it does nothing while waiting for the next command from the master. If the master is mute, NodeBrain is an idle slave. In servant mode, NodeBrain reads `stdin` using non-blocking I/O and is free to perform other duties while waiting for the next command from the master.

1.3.1 Scheduling Rules

Let's first learn a simple way to give NodeBrain something else to do while waiting for commands. Here's a rule file like `roger.nb` that includes a scheduling rule. Call it `jeeves.nb`.

```
# jeeves.nb
define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
define mySchedulingRule on(~(3s)) a=0;
```

The rule condition `~(3s)` specifies that the rule should fire every 3 seconds. When the `mySchedulingRule` fires, it asserts that a=0. If NodeBrain is waiting on input from `stdin`, it cannot trigger this rule as specified. So you have to change the way NodeBrain runs.

1.3.2 Servant Option

Here's a script called `bertie.pl` that uses the `-s` option to put NodeBrain in servant mode once all the input files specified as arguments have been processed.

```
bertie.pl#!/usr/bin/perl
use FileHandle;
open(NB,"|nb jeeves.nb -s")||die;  # use servant option
NB->autoflush(1); # force output to nb as soon as we send a command
for($i=1;$i<10;$i++){
  $r=int(rand()*4);
  if($r%2){print(NB "assert a=$r;\n");}
  else{print(NB "assert b=$r;\n");}
  sleep(5);             # pretend like we are busy working on something
  }
print(NB "stop\n");    # stop the servant mode nb
close(NB);
```

You can see from the output below that the activity scheduled by `mySchedulingRule` is taking place between commands sent by `bertie.pl`.

```
$ ./bertie.pl
2008/06/05 21:31:00 NB000I Argument [1] jeeves.nb
> # jeeves.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> define myProcessRule3 on(a=1 and b=2):-:./processCount.pl
> define mySchedulingRule on(~(3s)) a=0;
2008/06/05 21:31:00 NB000I Source file "jeeves.nb" included. size=283
2008/06/05 21:31:00 NB000I Argument [2] -s
2008/06/05 21:31:00 NB000T Servant mode selected
---------- ----------------------------------------------------
> assert a=3;
2008/06/05 21:31:03 NB000I Rule mySchedulingRule fired (a=0)
> assert b=0;
2008/06/05 21:31:06 NB000I Rule mySchedulingRule fired (a=0)
2008/06/05 21:31:09 NB000I Rule mySchedulingRule fired (a=0)
> assert a=3;
2008/06/05 21:31:12 NB000I Rule mySchedulingRule fired (a=0)
2008/06/05 21:31:15 NB000I Rule mySchedulingRule fired (a=0)
> assert b=2;
2008/06/05 21:31:18 NB000I Rule mySchedulingRule fired (a=0)
> assert b=0;
2008/06/05 21:31:21 NB000I Rule mySchedulingRule fired (a=0)
2008/06/05 21:31:24 NB000I Rule mySchedulingRule fired (a=0)
> assert a=3;
2008/06/05 21:31:27 NB000I Rule mySchedulingRule fired (a=0)
2008/06/05 21:31:30 NB000I Rule mySchedulingRule fired (a=0)
> assert b=2;
2008/06/05 21:31:33 NB000I Rule mySchedulingRule fired (a=0)
> assert b=2;
2008/06/05 21:31:36 NB000I Rule mySchedulingRule fired (a=0)
2008/06/05 21:31:39 NB000I Rule mySchedulingRule fired (a=0)
> assert a=1;
2008/06/05 21:31:40 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[7537] Started: -:./processCount.pl
[7537: assert processes=59;
2008/06/05 21:31:40 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[7541] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[7541] Exit(0)
[7537] Exit(0)
2008/06/05 21:31:43 NB000I Rule mySchedulingRulefired (a=0)
2008/06/05 21:31:45 NB000I Rule mySchedulingRulefired (a=0)
> stop
2008/06/05 21:31:45 NB000I NodeBrain nb[7536] terminating - exit code=0
$
```

1.4 Agent Mode

Monitoring applications normally require a program that runs constantly and mostly without user control but acts on the user's behalf. These qualities of persistence, relative autonomy, and user-directed goals are characteristic of a software agent. These programs are known as as daemons on Unix or Linux, and services on Windows.

Let's modify the rules from the previous example so the program can execute the process counting script every minute.

```
# maxwell.nb
define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
define myProcessRule3 on(~(m)):-:./processCount.pl
```

Now you'll execute it.

```
$ nb maxwell.nb
2008/06/05 21:45:53 NB000I Argument [1] maxwell.nb
> # maxwell.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> # check process count every minute
> define myProcessRule3 on(~(m)):-:./processCount.pl
2008/06/05 21:45:53 NB000I Source file "maxwell.nb" included. size=273
2008/06/05 21:45:53 NB000I NodeBrain nb[7618] terminating - exit code=0
$
```

But wait, that's not right. The program just ended after loading the rules. You need a way to tell it to run as a daemon. Try the daemon option `-d`.

```
$ nb .d maxwell.nb
2008/06/05 21:47:28 NB000I Argument [1] -d
2008/06/05 21:47:28 NB000I Argument [2] maxwell.nb
> # maxwell.nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> # check process count every minute
> define myProcessRule3 on(~(m)):-:./processCount.pl
2008/06/05 21:47:28 NB000I Source file "maxwell.nb" included. size=273
2008/06/05 21:47:28 NB000I NodeBrain nb[7621,7060]daemonizing
$
```

Okay, that's better, it daemonized. It disconnected from the terminal and is running in the background with no `stdin` or `stdout`. It actually forks a new process to run as the daemon and then ends.

```
$ ps -ef | grep nb
myuser   7622     1  0 21:47 ?       00:00:00 nb -d maxwell.nb
```

Well, you have an agent running now, but you don't have any way to see what it is doing. For better visibility, you need to instruct NodeBrain to write to a log file. While you're at it, make it an executable script by adding a she-bang line. Guess you should stop the running agent also.

```
$ kill 7622
```

Here's the enhanced agent called `james.nb`.

```
#!/usr/local/bin/nb -d
# james.nb
set log="james.log";
define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
# check process count every minute
define myProcessRule3 on(~(m)):-:./processCount.pl
```

If you want to execute it, set the file permissions.

```
$chmod 755 james.nb
```

Now let's start it.

```
$ ./james.nb
2008/06/05 21:55:31 NB000I Argument [1] -d
2008/06/05 21:55:31 NB000I Argument [2] ./james.nb
> #!/usr/local/bin/nb -d
> # james.nb
> set log="james.log";
2008/06/05 21:55:31 NB000I NodeBrain nb will log to james.log
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
> define myProcessRule2 on(processes<100):-./alarmMe.pl 2 "ok - under 100 processes."
> # check process count every minute
> define myProcessRule3 on(~(m)):-:./processCount.pl
2008/06/05 21:55:31 NB000I Source file "./james.nb" included. size=370
2008/06/05 21:55:31 NB000I NodeBrain nb[7688,7060] daemonizing
$
```

Now you can see what the agent is doing by tailing the log file. Wait a minute, it isn't doing much. Yes, wait another minute.

```
$ tail james.log
2009/01/28 09:04:26 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[5619] Started: -:./processCount.pl
[5619: assert processes=75;
2009/01/28 09:04:26 NB000I Rule myProcessRule2 fired
: -./alarmMe.pl 2 "ok - under 100 processes."
[5623] Started: -./alarmMe.pl 2 "ok - under 100 processes."
[5623] Exit(0)
[5619] Exit(0)
2009/01/28 09:05:00 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[5648] Started: -:./processCount.pl
[5648: assert processes=78;
[5648] Exit(0)
2009/01/28 09:06:00 NB000I Rule myProcessRule3 fired
: -:./processCount.pl
[5659] Started: -:./processCount.pl
[5659: assert processes=78;
[5659] Exit(0)
```

The log tells you the scheduled process is running and that 78 active processes on this system. If the count goes over 200, you will be notified by `rule myProcessRule1` via the `alarmMe.pl` script.

That's it—you have a working agent. But clearly this example was not designed to convince you that NodeBrain should be used when the application is this simple. This entire problem could be solved with one simple script written in your favorite scripting language. NodeBrain is appropriate for situations where there is a benefit to separating declarative rules from procedural logic in a modular fashion. It may take experimentation with more complex and evolving agent requirements to appreciate when it is appropriate.

2 Cell Expressions

I like nonsense; it wakes up the brain cells. —Dr. Seuss (1904–1991)

What could be more nonsensical than a program called NodeBrain? And of course, if I claim this program is a brain, there must be brain cells and a way to wake them up. So here goes.

You can think of a cell as a structure capable of storing a value and optionally a formula for computing a value. You may be familiar with this notion if you use a spreadsheet program. A NodeBrain cell is vary similar to a spreadsheet cell, but without the tabular model. There is no physical relationship between NodeBrain cells; no notions like "two to the left" or "one cell up." NodeBrain cells are referenced only by their formula, which is called a cell expression.

Cell expressions are allowed in various places within the command syntax. I can't tell you where exactly because I promised not to get BNF (Backus-Naur Form) on you here, but I can say cell definitions, assertions, and rule conditions are the primary places where cell expressions are used.

2.1 Numbers and Strings

The simplest cell expressions identify a literal number or string. Here you define a cell `a` with the expression `5`, and a cell `b` with the expression `"abc"`.

```
$ nb
> define a cell 5;
> define b cell "abc";
> show a
a = 5
> show b
b = "abc"
>
```

2.2 Arithmetic

NodeBrain can do simple arithmetic in a cell expression.

```
$ nb
> define a cell 5+4;
> define b cell a+3;
> show a
a = ! == (5+4)
> show b
b = ! == (a+3)
>
```

Wait, this must not be what Dr. Seuss meant by nonsense because these brain cells don't seem to be awake! NodeBrain accepted the expression `5+4` but didn't compute `9`. This is because NodeBrain is lazy. Nobody cares what the value of `a` or `b` is, so NodeBrain doesn't

bother to compute them. You need more nonsense to wake up these brain cells. Let's try adding a rule that depends on the value of `b`.

```
> define rule1 on(b<20);
> show a
a = 9 == (5+4)
> show b
b = 12 == (a+3)
```

When you added the rule referencing `b`, NodeBrain was motivated to compute the value of `a+3`. To get the value of `a`, it computes the value of `5+4`.

The double equals signs symbol (`==`) can be read as "is defined as." The response to `show b` above can be read as "`b` is `12` and is defined as `a+3`."

Just for fun, let's see what happens when you assert new cell expressions for `a`.

```
> assert a=19;
> show a
a = 19
> show b
b = 22 == (a+3)
>assert a=1;
2008/06/05 22:04:51 NB000I Rule rule1 fired
> show b
b = 4 == (a+3)
> assert a=5*15/7+2;
> show a
a = 12.71428571
>
```

Although you are on the topic of arithmetic, it is perhaps less interesting that NodeBrain does arithmetic than when it does it. When you asserted that `a=1`, NodeBrain recomputed `b` and got `4`. It then recomputed `b<20`, which transitioned from false to true and `rule1` fired.

2.3 Logic

> *Logic is like the sword—those who appeal to it shall perish by it.* —Samuel Butler (1835–1902)

One can appeal to logic in cell expressions, much as in other programming languages. Some familiar operators are illustrated here.

```
> assert a=1,b=2,c=3,d=3;
> define x cell a>b;
> define rule2 on(a=b or b=c or c>25);
> define rule3 on(a=21 and (b>5 or c<2));
> define rule4 on(x and not a<5.73);
> define rule5 on(a=b and c=d);
```

Relational operators (`> < <> = >= <=`) have precedence over Boolean operators (`and`, `or`, `not`). Use parentheses to force the order of precedence when you don't want or don't know the default.

As demonstrated with arithmetic expressions, NodeBrain evaluates logical expressions when there are assertions that change the value of operands. For example, consider the following expression used by `rule5`.

```
a=b and c=d
```

If you assert that `a=3`, the interpreter has to recomputed `a=b`, but doesn't recompute the larger expression `a=b and c=d`. Since `c` and `d` didn't change, the value of `c=d` has not changed, and since `a=b` remains false, the value of the larger expression is known to be unchanged without further computation.

There is only one cell for computing any given expression. No matter how many rule conditions reference `a=b`, there is only one `a=b` cell, as there is only one `a` cell and one `b` cell. Every cell knows which cells reference it. In this example, both `a` and `b` know that `a=b` depends on them. If `a` changes, it reports a change to `a=b`. If `b` changes, it does the same. If `a` changes, `b` changes, or both `a` and `b` change in the same assertion, `a=b` is recomputed once. If `a=b` changes, expressions that reference it are recomputed.

How do you think the interpreter would respond to the following assertion?

```
assert a=21,b=21.
```

Because either of `a` or `b` has changed (in this case both), the interpreter must evaluate `a=b` again. Since the value is still true, no further computation is required.

2.4 Time Conditions

I must govern the clock, not be governed by it. —Golda Meir (1898–1978)

NodeBrain is happy to be governed by the clock. It has an extensive algebra for expressing schedules called time conditions. A time condition is a function of time that is true or false at any given second.

In the spirit of this tutorial, I have been deliberately incomplete here and give you a sense of it. I'm somewhat proud of this feature, so I've exercised conscious restraint. (If you need a cure for insomnia, read about time conditions in the *NodeBrain Language Reference*.)

I'll just jump into it with a few examples. The following script, `schedule.nb`, has rules with names describing their time condition. Let's start it in interactive mode.

```
$ nb schedule.nb -
2008/06/06 06:38:14 NB000I Argument [1] schedule.nb
> define FourOClockHour on(~(h(4)));
> define MondayWednesayAndFriday on(~(mo,we,fr));
2008/06/06 06:38:14 NB000I Rule MondayWednesayAndFriday fired
> define Four2FiveMonWedFri on(~(h(4).(mo,we,fr)));
> define January9 on(~(jan(9)));
> define WeekOfJanuary15 on(~(w.jan(15)));
> define ThursdayOfWeekOfJanuary15 on(~(th.w.jan(15)));
> define LastFridayInJanuary on(~(fr[-1]jan));
> define TuesdayOfWeekOfLastFridayInJanuary on(~(tu.w.fr[-1]jan));
2008/06/06 06:38:14 NB000I Source file "schedule.nb" included. size=371
2008/06/06 06:38:14 NB000I Argument [2] -
2008/06/06 06:38:14 NB000I Reading from standard input.
---------- --------
>
```

To see the next event for each schedule, use the `show` command with a `/c` (clock) option. I issued this command at 6:41 on Friday, June 6, 2008. The `h(4)` condition will transition to true at 04:00, and then back to false at 05:00. The `(mo,we,fr)` condition will transition to true at 00:00 Monday the 9th. See if you can make sense out of the rest of the conditions in this list, given the hint that weeks start at 00:00 Sunday. Match the conditions to the name of the corresponding rule above if you need help.

```
> show /c
~ 2008/06/06 06:41:17 Clock
~ 2008/06/07 00:00:00 ~(mo,we,fr)
                      on(~(mo,we,fr));
~ 2008/06/07 04:00:00 ~(h(4))
                      on(~(h(4)));
~ 2008/06/09 04:00:00 ~(h(4).(mo,we,fr))
                      on(~(h(4).(mo,we,fr)));
~ 2009/01/09 00:00:00 ~(jan(9))
                      on(~(jan(9)));
~ 2009/01/11 00:00:00 ~(w.jan(15))
                      on(~(w.jan(15)));
~ 2009/01/15 00:00:00 ~(th.w.jan(15))
                      on(~(th.w.jan(15)));
~ 2009/01/27 00:00:00 ~(tu.w.fr[-1]jan)
                      on(~(tu.w.fr[-1]jan));
~ 2009/01/30 00:00:00 ~(fr[-1]jan)
                      on(~(fr[-1]jan));
```

A `forecast` command is used to see when a time condition will transition to true (left column) and false (right column).

```
> forecast ~(mo,we,fr)
mo,we,fr schedule ~0-0 interval=0,duration=0)
fr 2008/06/06 00:00:00 1212735600 - sa 2008/06/0700:00:00 1212822000
mo 2008/06/09 00:00:00 1212994800 - tu 2008/06/1000:00:00 1213081200
we 2008/06/11 00:00:00 1213167600 - th 2008/06/1200:00:00 1213254000
fr 2008/06/13 00:00:00 1213340400 - sa 2008/06/1400:00:00 1213426800
mo 2008/06/16 00:00:00 1213599600 - tu 2008/06/1700:00:00 1213686000
we 2008/06/18 00:00:00 1213772400 - th 2008/06/1900:00:00 1213858800
> forecast ~(h(4));
h(4) schedule ~0-0 interval=0,duration=0)
sa 2008/06/07 04:00:00 1212836400 - sa 2008/06/0705:00:00 1212840000
su 2008/06/08 04:00:00 1212922800 - su 2008/06/0805:00:00 1212926400
mo 2008/06/09 04:00:00 1213009200 - mo 2008/06/0905:00:00 1213012800
tu 2008/06/10 04:00:00 1213095600 - tu 2008/06/1005:00:00 1213099200
we 2008/06/11 04:00:00 1213182000 - we 2008/06/1105:00:00 1213185600
th 2008/06/12 04:00:00 1213268400 - th 2008/06/1205:00:00 1213272000
```

The fundamental concepts to grasp here are that time conditions have

1. Identifiers like `mo` for Monday, `h` for hour, `jan` for January, and `w` for week that define a set of time intervals
2. Operators (`, . (n) [n]`) that operate on time interval sets to produce new sets

The operators are based on formal concepts like intersection, union, selection, rejection, inverse, and so on.

2.5 Node Conditions

An expert knows all the answers—if you ask the right questions. —Unknown

A NodeBrain node is an expert, with a set of knowledge and associated skill. The skill is provided by a node module (plug-in) that extends NodeBrain functionality. Since NodeBrain is not a general purpose scripting language, many things are simply inappropriate to attempt without help from a node module. It is the combination of skill and knowledge that makes a node an expert. Of course, you need to know how to ask the right questions.

Node modules can be developed to perform any function imaginable, but within the context of a cell expression they only provide the value of node conditions. These expressions look like a function call. In rule `r1` below, `sally(x)` is a node condition and `sally` is a node with the skills provided by the tree module.

```
$ nb sally.nb
2008/06/06 07:05:49 NB000I Argument [1] sally.nb
> # sally.nb
> define sally node tree;
> assert sally("abc"),sally("def",5)=2;
> show sally
sally = ! == node tree
  "def"
    5=2
  "abc"=1
> define r1 on(sally(x));
> assert x="abc";
2008/06/06 07:05:49 NB000I Rule r1 fired
```

Do you see why rule `r1` fired when you asserted that `x="abc"`? It is because `sally("abc")` was asserted to be true, so `sally(x)` is true when `x="abc"`. What a node module does with assertions and how it computes the value of node conditions is entirely up to it. In this case, the `tree` node module keeps track of assertions and returns a value for conditions by doing a lookup. When the value of `x` changes, the interpreter reports a change to the node module and it computes a new value for the cell. When the cell value changes, it reports a change to the referencing cells, in this case rule `r1`, which fired.

2.6 Complex Expressions

> *A complex system that works is invariably found to have evolved from a simple system that works.* —John Gaule (c. 1603–1687)

Every type of cell expression introduced in this tutorial can be combined into complex expressions as illustrated by rule `complex` below.

```
$ nb complex.nb
2008/06/06 07:14:28 NB000I Argument [1] complex.nb
> # complex.nb
> define sally node tree;
> assert sally("abc"),sally("def",5)=2;
> define complex on(sally(x,y)=a+b and y<7 and ~(fr));
> assert y=5;
> assert a,b;
> assert x="def";
2008/06/06 07:14:28 NB000I Rule complex fired
```

If the complex rule doesn't fire when you run `complex.nb`, perhaps you are not running it on a Friday. If you try it on a Friday, it will fire because `y<7` is true and `sally(x,y)=a+b` is true. The fact that the last expression is true may not be so obvious, so let's break it down. The command `assert a,b;` sets `a` and `b` to 1 (true). That means the expression `a+b` is 2. Since `x` is `"def"` and `y` is 5, the expression `sally(x,y)=a+b` is `sally("def",5)=2`, which you assert to be true.

3 Ignorance

> *Ignorance more frequently begets confidence than does knowledge.* —Charles Darwin (1809–1882)

This tutorial is about ignorance: when NodeBrain is aware of it and how it is handled. It is perhaps obvious, but important to remember, that NodeBrain assumes that whatever is asserted is the truth. It makes no attempt to independently verify assertions, or check whether assertions should have been made that were not. So what NodeBrain thinks it knows may not be the truth. In this respect, NodeBrain is unaware of its ignorance and blissfully acts with great confidence. No different than most of us.

However, within NodeBrain's own version of reality, it can be aware of a form of ignorance. The notion here is that something can be known to be unknown, a possibility often dismissed by people rushing to judgment. How NodeBrain deals with that condition is the topic of this tutorial.

3.1 True, False, and Unknown

> *Something unknown is doing we don't know what.* —Sir Arthur Eddington (1882–1944)

The NodeBrain interpreter recognizes three logical states: true, false, and unknown. These states are represented as follows.

Value	**Representation**
Unknown	?
False	0
True	1 or anything else (e.g., 5.2, –7, "abc", "")

You can test for these states by using the `=` relational operator or a prefix operator. The `!` prefix operator means "not" and the `?` prefix operator means "is unknown."

```
> define r1 on(a=1 and b=0 and c=? and x and !y and ?z);
```

Values can be asserted similarly.

```
$ nb
> assert a=1,b=0,c=?,x,!y,?z;
> show -t
@ =! == node
z =?
y =0
x =1
c =?
b =0
a =1
>
```

Although anything other than `?` or `0` is interpreted as true, logical operators often return a value of `1` to represent a true state. Below you see that the value of `a<b` is `1`.

```
> assert a=5,b=7;
> define r1 on(a<b);
> show (a<b)
()= 1 == (a<b)
```

Here's the important thing to understand. Logical operators accept the unknown value as an operand and return the unknown value under specific conditions. Logic tables for `A and B` and `A or B` are shown below. Notice that in some cases NodeBrain knows the result without knowing both operands.

A	B	A&B	A|B
0	0	0	0
0	1	0	1
0	?	0	?
1	0	0	1
1	1	1	1
1	?	?	1
?	0	0	?
?	1	?	1
?	?	?	?

The `?` prefix operator enables you to test for an unknown condition and act on it.

```
$ nb
> define x cell a+b;
> define r1 on(?x);
> assert a=7,b=12;
> show -t
@ =! == node
r1= ! == on(?x);
x =19 == (a+b)
b =12
a =7
> assert ?b;
2008/06/0607:41:34 NB000I Rule r1 fired
>
```

Let's pause to make sure you know what just happened here. The condition `?x` is true when `x` is unknown. This was true when `r1` was defined because `x` is `a+b`and both `a` and `b` are unknown at that time. Hey, the arithmetic operators understand the unknown value also. When you assert that `a=7` and `b=12`, `x` was known to be 19 and `?x` was false. Then you assert `?b`, making `a+b` unknown, making `x` unknown, making `?x` true and `r1` fired.

3.2 Closed World Assumption

Some systems use a closed world assumption, where anything not known to be true is assumed to be false. As illustrated in the previous section, NodeBrain does not generally make this assumption. Instead it allows the unknown state to propagate through the evaluation of cell expression.

However, node modules are allowed to handle the unknown state using either the closed world assumption or three-state logic. This means you must be aware of how each node module handles the unknown state when coding rules.

I'll illustrate using two nodes, one with the `tree` skill and one with the `cache` skill. The `tree` module uses three-state logic and the `cache` module uses the closed world assumption.

```
$ nb closedWorld.nb
2008/06/0607:44:05 NB000I Argument [1] closedWorld.nb
> define myTree expert tree;
> define myCache expert cache:(soup);
> define rMyTreeHasSoup on(myTree(soup));
> define rMyCacheHasSoup on(myCache(soup));
> define rMyTreeHasNotSoup on(!myTree(soup));
> define rMyCacheHasNotSoup on(!myCache(soup));
> define rMyTreeKnowsNotSoup on(?myTree(soup));
> define rMyCacheKnowsNotSoup on(?myCache(soup));
> assert myTree("Chicken"),myCache("Chicken");
> assert soup="Tomato";
> assert soup="Chicken";
2008/06/0607:44:05 NB000I Rule rMyCacheHasSoup fired
2008/06/0607:44:05 NB000I Rule rMyTreeHasSoup fired
> assert ?myTree("Chicken"),?myCache("Chicken");
2008/06/0607:44:05 NB000I Rule rMyTreeKnowsNotSoup fired
2008/06/0607:44:05 NB000I Rule rMyCacheHasNotSoup fired
> assert !myTree("Chicken"),!myCache("Chicken");
2008/06/0607:44:05 NB000I Rule rMyTreeHasNotSoup fired
2008/06/0607:44:05 NB000I Source file "closedWorld.nb" included. size=514
2008/06/0607:44:05 NB000I NodeBrain nb[13980] terminating - exit code=0
$
```

This may just be a blur, so let's take the assertions one at a time. Initially, `soup` is unknown, `myTree(soup)` is unknown, and `myCache(soup)` is false. The reason `myCache(soup)` is false is because the `cache` skill uses the closed world assumption; what is not known to be true is false. When you assert `"Chicken"` to both the `tree` and the `cache`, there is no change to the value of `myTree(soup)` and `myCache(soup)`. When you assert that `soup="Tomato"`, there is still no change. A value of `"Tomato"` is not known to `tree` or `cache`, in the same way an unknown soup is not known.

Next, you assert that `soup="Chicken"`. Both `myTree(soup)` and `myCache(soup)` transition to true and the corresponding rules `rMyCacheHasSoup` and `rMyTreeHasSoup` fire.

This is followed by an assertion that `myTree("Chicken")` and `myCache("Chicken")` are unknown. Because the `cach` skill uses the closed world assumption, this is the same as asserting that `myCache("Chicken")` is false. Consequently, `rMyCacheHasNotSoup` fires while `myTreeKnowsNotSoup` fires.

Finally, you assert `myTree("Chicken")` and `myCache("Chicken")` to be false. This causes the rule `rMyTreeHasNotSoup` to fire. There is no change to `myCache("Chicken")`, so no cache-related rule fires.

The difference between the `tree` and `cache` skills is that you must assert `myTree(soup)` to be false before `myTree(soup)` evaluates to false, while `myCache(soup)` is false simply because you haven't asserted it to be true. Asserting either `!myCache("Chicken")` or `?myCache("Chicken")` removes knowledge of `"Chicken"` from the cache. Only an assertion of `?myTree("Chicken")` removes knowledge of `"Chicken"` from the tree. Asserting `!myTree("Chicken")` asserts a value of false to the `"Chicken"` entry in the tree.

3.3 Reactive Mode

In all the previous examples in this tutorial, the interpreter has been operating in reactive mode when processing rules. In this mode, the interpreter waits for assertions and responds appropriately. It makes no effort to convert unknown values into known values. Consider the following example.

```
$ nb
> define myProcessRule1 on(processes>200):-./alarmMe.pl 1 "dang - more than 200 processes."
>
```

The interpreter knows the value of processes is unknown and it does nothing to find out. This is reasonable because you haven't asked it to and you haven't told it how. These are possibilities covered in the next section.

3.4 Query Mode

There is an interpreter mode called *query* mode, in which NodeBrain attempts to solve for unknown values as required to obtain a known value for rule conditions. This mode approximates what your doctor does when you go in with a medical problem, or your car mechanic does when you take your car in with a mechanical problem. They ask you questions, and they ask themselves questions that they try to answer by running tests. They attempt to diagnose your problem by applying rules and seeking values for parameters that are initially unknown. They only ask the questions and run the tests that are necessary to arrive at a conclusion. This helps to avoid running unnecessary expensive tests.

If you tell NodeBrain how to query for unknown values and ask it to solve for rule conditions that have an unknown value, it can diagnose or investigate with similar efficiency.

3.4.1 Query Rule Files

A query rule file is simply a rule file designed to be processed in query mode. Here's a file named `illness.nb` that is a trivial (and not useful) query rule file.

```
# File: tutorial/Ignorance/illness.nb
define CommonCold on(cough and soreThroat);
define Flu on(fever and achy and upsetStomach);
```

As you can see, these query rules have the same syntax as reactive rules. The difference is not in the rules, but in how you use them.

3.4.2 Interactive Query Mode

Let's invoke this script in interactive mode and issue a `query` command.

```
$ nb illness.nb -
> # File: tutorial/Ignorance/illness.nb
> define CommonCold on(cough and soreThroat);
> define Flu on(fever and achy and upsetStomach);
2008/06/10 15:51:44 NB000I Source file "illness.nb" included. size=111
2008/06/10 15:51:44 NB000I Argument [2] -
2008/06/10 15:51:44 NB000I Reading from standard input.
---------- --------
> query

  (fever&(achy&upsetStomach))

Enter cell fever==0

  (cough&soreThroat)

Enter cell cough==1

  (cough&soreThroat)

Enter cell soreThroat==1
2008/06/10 15:52:11 NB000I Rule CommonCold fired
>
```

Since you are running in interactive mode and have not provided any alternate methods for NodeBrain to solve for unknown values, the interpreter prompts the user to resolve unknown values. If you study the rule file, you will find five unknown terms in the two rule conditions: `cough`, `soreThroat`, `fever`, `achy`, and `upsetStomach`. However, the interpreter only prompted the user for values of three of these terms. This is because requesting values for the other two terms would not have changed the result. Since `fever=0`, the `flu` condition is false for any value of the other two terms in the condition.

The response to each query may be any valid cell expression. This means you must enclose strings in quotes and have the option of introducing new terms for which the interpreter does not have a value. When new terms are introduced, the interpreter attempts to resolve them as necessary. If you respond with `5 or soreEye`, the interpreter will not attempt to solve for `soreEye` because it knows the expression is true without knowledge of `soreEye`.

3.4.3 Batch Query Mode

In most applications of NodeBrain, you are attempting to automate decisions, so you want to query software to resolve unknowns where possible. Here's a script called `illnessConsult.pl` that simulates the running of potentially complex tests to arrive at the same answers you provided interactively above. There would be no point in doing something this simple for real, and no point in doing something more complicated in a tutorial. You can imagine a much more complicated problem.

```
#!/usr/bin/perl
# File: tutorial/Ignorance/illnessConsult.pl
$state{"cough"}=1;
$state{"soreThroat"}=1;
$state{"fever"}=0;
$state{"achy"}=1;
$state{"upsetStomach"}=0;

$symptom=shift;
if(exists($state{$symptom})){print("$state{$symptom}\n");}
else{print("?\n");}
```

Let's run this script that you are pretending is complicated software to determine the value of the `cough` symptom.

```
$./illnessConsult.pl cough
1
```

Create a new rule file called `illnessConsult.nb` to consult the Perl script.

```
# File: tutorial/Ignorance/illnessConsult.nb
use : ./illnessConsult.pl
defineCommonCold on(cough and soreThroat);
defineFlu on(fever and achy and upsetStomach);
query;
```

Diagnose an illness in batch mode.

```
$ nb illnessConsult.nb
2008/06/1016:12:25 NB000I Argument [1] illnessConsult.nb
> # File: tutorial/Ignorance/illnessConsult.nb
> use : ./illnessConsult.pl
> define CommonCold on(cough and soreThroat);
> define Flu on(fever and achy and upsetStomach);
> query;
2008/06/1016:12:25 NB000T Resolving "fever" via command : ./illnessConsult.pl "fever"
Value=(0)
2008/06/1016:12:25 NB000T Resolving "cough" via command : ./illnessConsult.pl "cough"
Value=(1)
2008/06/1016:12:25 NB000T Resolving "soreThroat" via command : ./illnessConsult.pl "soreThroat"
Value=(1)
2008/06/1016:12:25 NB000I Rule CommonCold fired
2008/06/1016:12:25 NB000I Source file "illnessConsult.nb" included. size=149
2008/06/1016:12:25 NB000I NodeBrain nb[13690] terminating - exit code=0
$
```

In this example, the consulted script `illnessConsult.pl` returned simple 0 and 1 values. The interpreter will accept any valid cell expression and will attempt to resolve any term whose value is both unknown and needed.

4 Rules

> *If you obey all the rules, you miss all the fun.* —Katharine Hepburn (1907–2003)

In this tutorial you'll see how much fun it is to make up rules and watch NodeBrain obey them.

4.1 When and If a Rule Fires on What

NodeBrain `on`, `when`, and `if` rules have almost identical syntax. Their conditional cell expressions are resolved by the same cell evaluation algorithm, yet they respond differently to new information. You can study these differences using rules that have no action. Start by creating the following script called `rules.nb`.

```
# File: tutorial/RuleTypes/rules.nb
define r0 when(a=1 and b=2);
define r1 on(a=1 and b=2);
define r2 on(a=2 and b=2);
define r3 if(a=1 and b=2);
assert a=1,b=2;
assert a=1,b=2;
assert a=2;
assert a=27;
alert a=1;
alert a=1;
```

When you execute this script you will notice that rules fire in response to assert and alert commands.

```
$ nb rules.nb
2008/06/1016:54:24 NB000I Argument [1] rules.nb
> # File: tutorial/RuleTypes/rules.nb
> define r0 when(a=1 and b=2);
> define r1 on(a=1 and b=2);
> define r2 on(a=2 and b=2);
> define r3 if(a=1 and b=2);
> assert a=1,b=2;
2008/06/1016:54:24 NB000I Rule r0 fired
2008/06/1016:54:24 NB000I Rule r1 fired
> assert a=1,b=2;
> assert a=2;
2008/06/1016:54:24 NB000I Rule r2 fired
> assert a=27;
> alert a=1;
2008/06/1016:54:24 NB000I Rule r1 fired
2008/06/1016:54:24 NB000I Rule r3 fired
> alert a=1;
2008/06/1016:54:24 NB000I Rule r3 fired
2008/06/1016:54:24 NB000I Source file "rules.nb" included. size=206
2008/06/1016:54:24 NB000I NodeBrain nb[13859] terminating - exit code=0
$
```

Although rules `r0`, `r1`, and `r3` are based on the same condition `(a=1 and b=2)`, they react differently to the `assert` and `alert` commands.

Q: Why didn't r3 fire on the first `"assert a=1,b=2;"` command?

A: Because `if` rules only respond to `alert` commands.

Q: Why didn't `r1` fire on the second `assert a=1,b=2;` command?

A: Because `on` rules only fire when their condition transitions to true. The second assertion made no change in the state of the condition.

Q: Why didn't `r0` fire when `r1` fired the second time?

A: Because `when` rules only fire once. They are removed from the rule set once they fire.

Q: Why did `r3` fire on the second `alert a=1;` command?

A: Because `if` rules fire on every `alert` command that leaves their condition in a true state. Unlike `on` and `when` rules, they are not required to transition to a true state from a false or unknown state.

4.2 State Monitoring vs. Event Monitoring

In the previous example, you may have noticed that NodeBrain supports both state and event monitoring. The difference between the two is subtle. The `assert` command and `on` rule are designed for state monitoring. The `alert` command and `if` rule are designed for event monitoring. But it isn't that simple. The `on` rule responds to both `assert` and `alert` commands. To an `on` rule, `alert` is just an alias for `assert`. In fact, everything about the way an `alert` command is processed is identical to the way an `assert` command is processed, except `alert` brings `if` rules into play. In other words, `alert` makes an assertion, and then says, "Oh, and by the way, this is an event to be considered by the `if` rules."

What makes the notion of state and event monitoring hard to untangle is that an event is described in the same way as a state, an event is interpreted as a new state, a state change is recognized as an event, and a state change can trigger an event. Let's look at these confusing statements individually.

```
# describe an event in the same way we describe a state
alert a=1,b=2;        # event
assert a=1,b=2;       # state

# interpret an event as a new state
define r1 on(a=1 and b=2);
define r2 if(a=1 and b=2);
alert a=1,b=2;        # recognized as new state to both IF and ON rules
alert a=1,b=2;        # recognized as new state to IF rule, not to ON rule

# recognize state changes as events
define r1 on(a=1 and b=2);
assert a=0,b=0;
assert a=1,b=2;       # triggers internal events that cause r1 to fire

# translate state changes into events
define r1 on(a=1 and b=2): alert x="abc",y="def";
assert a=1,b=2;       # r1 translated state change into event(alert)
```

4.3 Actions as Reactions

Rules are defined to react to state or event conditions with actions. An action can be an assertion, a command, or a combination of both, The following `on` rules have actions that use different approaches to accomplish the same result. These same actions could be used with `if` or `when` rules, since their actions are specified in the same way.

```
define r1 on(a) x=5,y=2;         # action is assertion
define r2 on(b):assert x=5,y=2;  # action is assert command
define r3 on(c) x=5:assert y=2;  # action is assertion and assert command
assert a;
show x
show y
assert ?x,?y;
show x
show y
assert b;
show x
show y
assert ?x,?y;
show x
show y
assert c;
show x
show y
```

Copy the script above into a file named `xwise.nb` and execute it as illustrated below.

```
$ nb xwise.nb
> define r1 on(a) x=5,y=2;         # action is assertion
> define r2 on(b):assert x=5,y=2;  # action is assert command
> define r3 on(c) x=5:assert y=2;  # action is assertion and assert command
> assert a;
2012/12/30 12:17:22 NB000I Rule r1 fired (x=5,y=2)
> show x
x = 5
> show y
y = 2
> assert ?x,?y;
> show x
x = ?
> show y
y = ?
> assert b;
2012/12/30 12:17:22 NB000I Rule r2 fired
: assert x=5,y=2;  # action is assert command
> show x
x = 5
> show y
y = 2
> assert ?x,?y;
> show x
x = ?
> show y
y = ?
> assert c;
2012/12/30 12:17:22 NB000I Rule r3 fired (x=5)
: assert y=2;  # action is assertion and assert command
> show x
x = 5
> show y
y = 2
```

When making a simple assertion, a rule like `r1` above is the recommended approach because the assertion is parsed when the rule is defined. When using a command, as in `r2`, the command is not parsed until the rule fires. In `r3`, the `x=5` assertion is parsed when the rule is defined, and the `assert y=2` command is parsed when the rule fires.

A command action is required for a response other than a simple assertion, and even for an assertion when symbolic substitution requires waiting until the rule fires to parse the assertion.

```
define r1 if(a):$ assert x="${n} cases of ${e}";
assert n=5,e="something";
alert a;
```

Issue an `nb` command at a shell prompt and paste the example above at the NodeBrain prompt. You should see the following.

```
$ nb
> define r1 if(a):$ assert x="${n} cases of $x{e}";
> assert n=5,e="something";
> alert a;
2012/12/30 16:52:54 NB000I Rule r1 fired
: assert x="5 cases of something";
```

Use shell commands when the action can not be performed by NodeBrain directly. Try this.

```
define r1 on(a):-|echo ‘date‘ ":it happened" >> happening.log;cat happening.log
assert a;
```

More complicated actions can be coded in your favorite scripting language or developed as a compiled executable.

5 Syntax Surprises

This section provides answers to some syntax questions.

5.1 When do I use a semi-colon to end a command?

> *From now on, ending a sentence with a preposition is something up with which I will not put.*—Sir Winston Churchill (1874–1965)

NodeBrain does not require a semi-colon to terminate a command at the end of a line. The following examples are both acceptable.

```
> assert a=1,b=2;
> assert a=1,b=2
```

A semi-colon is required to terminate a command before a comment.

```
> assert a=1,b=2; # This is valid
> assert a=1,b=2  # This is not valid
```

You can not put two commands on one line.

```
> assert a=1,b=2; define x cell 10; # This is not valid
```

In some cases, a NodeBrain command ends with a command directed to another interpreter. In those cases, you must follow the syntax rules of the other interpreter. The following rule that uses a host shell command as the action is illustrated in this concept.

```
> define psNb on(~(m)):-ps -ef | grep nb
```

5.2 How do I continue a command onto the next line?

There is no support for multiple line commands. A new-line character (`\n`) is always interpreted as the end of a command.

5.3 How can I specify multiple commands for a rule action?

If you are specifying multiple shell commands, the shell supports semi-colon (`;`) separated commands on a single line.

```
> define r1 on(a=1 and b=2):= myscript1.pl; rm*.foo; myscript2.pl
```

You may also include multiple comma-separated NodeBrain assertions.

```
> define r1 on(a=1 and b=2) x=5,y="abc",z=20 := myscript1.pl;rm *.foo; myscript2.pl
```

If you need to specify multiple NodeBrain commands, they cannot be specified directly in the action of a rule. But let's look at some alternatives.

If order is important, and you want to specify several commands, place the commands in a separate source file, and use the `source` command as the rule action.

```
> define r1 on(a=1 and b=2):source r1Action.nb
```

You may also use a `-:` or `=:` (servant) command and have a servant script present multiple commands to the interpreter.

```
> define r1 on(a=1 and b=2):-:r1Action.pl
```

If the order of the commands is not important and the number is relatively small, you can use multiple rules with the same condition. To avoid typing the condition multiple times, specify the name of the first rule as the second rule's condition.

```
> define r1 on(a=1 and b=2):-myscript1.pl
> define r2 on(r1): sally. alert x=5;
> define r3 on(r2): source r3Action.nb
```

Under the current implementation, you should not expect these rules to fire in any particular order.

5.4 How do I include quotes within a string?

Including quotes within a string is not allowed. Because NodeBrain is not a general purpose or report writing language, I have not yet found a reason to support quotes within a string. When generating cell expressions manually or with a program, simply avoid generating values containing quotes.

5.5 Is there an escape sequence for including special characters in strings?

No, and I like it that way. Special characters aren't necessary in NodeBrain strings. In commands sent to the host shell and messages sent to servants or node modules, NodeBrain does not interfere with the use of any escape sequences supported by the target interpreter. NodeBrain is oblivious to it, and this is actually why there is no interference. If a string value assigned to a NodeBrain term is ultimately destined to another interpreter (shell or node module), then you are free to include an escape sequence (except an escaped double quote). Again, NodeBrain is oblivious to it.

6 Tree Node

> *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.

6.1 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
2009/01/31 12:36:15 NB000I Argument [1] ./stuff.nb
> #!/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;
2009/01/31 12:36:15 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
2009/01/31 12:36:15 NB000I Source file "./stuff.nb" included. size=301
2009/01/31 12:36:15 NB000I NodeBrain nb[24913] terminating - exit code=0
$
```

6.2 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
2009/01/31 12:56:27 NB000I Argument [1] ./watch.nb
> #!/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";
2009/01/31 12:56:27 NB000I Rule r1 fired
> alert type="tide in";
> alert type="frog croaked";
2009/01/31 12:56:27 NB000I Rule r1 fired
2009/01/31 12:56:27 NB000I Source file "./watch.nb" included. size=289
2009/01/31 12:56:27 NB000I NodeBrain nb[26148] terminating - exit code=0
$
```

6.3 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.

```
2009/01/31 13:22:07 NB000I Argument [1] ./sequence.nb
> #!/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)): ...
... $ # ${namer} 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";
2009/01/31 13:22:07 NB000I Rule r1 fired (jumped(name)=1)
> alert event="jump",name="joe";
2009/01/31 13:22:07 NB000I Rule r1 fired (jumped(name)=1)
> alert event="land",name="sally";
2009/01/31 13:22:07 NB000I Rule r2 fired (jumped(name)=?)
> alert event="land",name="joe";
2009/01/31 13:22:07 NB000I Rule r2 fired (jumped(name)=?)
> alert event="jump",name="sally";
2009/01/31 13:22:07 NB000I Rule r1 fired (jumped(name)=1)
> alert event="land",name="joe";
2009/01/31 13:22:07 NB000I Rule r4 fired
: # joe landed twice without an intervening jump
> alert event="jump",name="sally";
2009/01/31 13:22:07 NB000I Rule r3 fired
: # sally jumped twice without an intervening landing
2009/01/31 13:22:07 NB000I Source file "./sequence.nb" included. size=631
2009/01/31 13:22:07 NB000I NodeBrain nb[26552] 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`).

7 Cache Node

> *There is a measure in everything. There are fixed limits beyond which and short of which right cannot find a resting place.* —Horace (65 BC–8 BC)

A cache node, like a tree node, provides a place to remember events as associations, or relations. But a cache adds to this the ability to measure repetition and variation within intervals of time and respond when predefined limits are reached. It enables you to detect patterns of events that can be described as follows:

- X happened N times within time period P or interval I.
- X was associated with Y in N events within time period P or interval I.
- X was associated with N different values of Y within time period P or interval I.
- X happened within interval I after Y happened.

Here X and Y represent the values of a set of one or more event attributes (A, B, C, ...). For example, X might represent a (type, city) tuple while Y represents a (customer, item, quantity) tuple. If you think of a cache as a table, which is sometimes a good way to visualize it, a *tuple* is just a table row. If you think of a cache as a table-of-tables, a better way to visualize it, then Y is a row in a table associated with row X.

7.1 Thresholds

Here's a 24-hour cache node with four thresholds and four rules to respond when they are reached.

```
#!/usr/local/bin/nb
# File: tutorial/cache/cache.nb
define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
horace. define r1 if(type._hitState): $ # There have been ${type._hits} ${type} events
horace. define r2 if(city._rowState): ...
... $ # (${type},${city}) had ${city._rows} different events
horace. define r3 if(city._kidState): ...
... $ # (${type},${city}) had ${city._kids} different customers
horace. define r4 if(quantity._hitState): ...
... $ # (${type},${city},${customer},${item},${quantity}) happened ${quantity._hits} times
```

Here's a set of events represented as assertions to the 24-hour cache node named "horace."

```
# File: tutorial/cache/events.nb
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","iPod",5);
horace. assert ("purchase","Paris","Bruno","shirt",2);
horace. assert ("purchase","Paris","Bruno","shoes",1);
horace. assert ("purchase","Paris","Madeleine","shoes",5);
horace. assert ("purchase","Paris","Madeleine","skirt",1);
horace. assert ("purchase","Paris","Madeleine","bread",2);
horace. assert ("purchase","Paris","Jeannine","bread",1);
horace. assert ("purchase","Paris","Laure","bread",1);
horace. assert ("purchase","Paris","Henri","iPod",1);
horace. assert ("return","Paris","Henri","iPod",1);
horace. assert ("return","Paris","Madeleine","iPod",1);
horace. assert ("purchase","London","Abigail","milk",1);
horace. assert ("purchase","London","Addie","bread",1);
horace. assert ("purchase","London","Alston","bread",1);
horace. assert ("purchase","London","Alston","candy",1);
horace. assert ("purchase","London","Alston","candy",2);
horace. assert ("purchase","London","Alston","candy",3);
horace. assert ("purchase","London","Alston","candy",4);
horace. assert ("purchase","London","Alston","candy",5);
horace. assert ("purchase","London","Alston","candy",6);
horace. assert ("purchase","London","Alston","candy",7);
horace. assert ("purchase","London","Alston","candy",8);
horace. assert ("purchase","London","Alston","candy",9);
```

Here's a sample execution.

```
$ ./cache.nb events.nb
2009/01/31 09:10:04 NB000I Argument [1] ./cache.nb
> #!/usr/local/bin/nb
> # File: tutorial/cache/cache.nb
> define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
> horace. define r1 if(type._hitState): ...
... $ # There have been ${type._hits} ${type} events
> horace. define r2 if(city._rowState): ...
... $ # (${type},${city}) had ${city._rows} different events
> horace. define r3 if(city._kidState): ...
... $ # (${type},${city}) had ${city._kids} different customers
> horace. define r4 if(quantity._hitState): ...
... $ # (${type},${city},${customer},${item},${quantity}) happened ${quantity._hits} times
2009/01/31 09:10:04 NB000I Source file "./cache.nb" included. size=511
2009/01/31 09:10:04 NB000I Argument [2] events.nb
> # File: tutorial/cache/events.nb
> horace. assert ("purchase","Paris","Bruno","iPod",5);
> horace. assert ("purchase","Paris","Bruno","iPod",5);
> horace. assert ("purchase","Paris","Bruno","iPod",5);
2009/01/31 09:10:04 NB000I Rule horace.r4 fired
: horace. # (purchase,Paris,Bruno,iPod,5) happened 3 times
> horace. assert ("purchase","Paris","Bruno","shirt",2);
> horace. assert ("purchase","Paris","Bruno","shoes",1);
> horace. assert ("purchase","Paris","Madeleine","shoes",5);
> horace. assert ("purchase","Paris","Madeleine","skirt",1);
> horace. assert ("purchase","Paris","Madeleine","bread",2);
> horace. assert ("purchase","Paris","Jeannine","bread",1);
> horace. assert ("purchase","Paris","Laure","bread",1);
> horace. assert ("purchase","Paris","Henri","iPod",1);
2009/01/31 09:10:04 NB000I Rule horace.r3 fired
: horace. # (purchase,Paris) had 5 different customers
> horace. assert ("return","Paris","Henri","iPod",1);
> horace. assert ("return","Paris","Madeleine","iPod",1);
> horace. assert ("purchase","London","Abigail","milk",1);
> horace. assert ("purchase","London","Addie","bread",1);
> horace. assert ("purchase","London","Alston","bread",1);
> horace. assert ("purchase","London","Alston","candy",1);
> horace. assert ("purchase","London","Alston","candy",2);
> horace. assert ("purchase","London","Alston","candy",3);
> horace. assert ("purchase","London","Alston","candy",4);
> horace. assert ("purchase","London","Alston","candy",5);
> horace. assert ("purchase","London","Alston","candy",6);
2009/01/31 09:10:04 NB000I Rule horace.r1 fired
: horace. # There have been 20 purchase events
> horace. assert ("purchase","London","Alston","candy",7);
2009/01/31 09:10:04 NB000I Rule horace.r2 fired
: horace. # (purchase,London) had 10 different events
> horace. assert ("purchase","London","Alston","candy",8);
> horace. assert ("purchase","London","Alston","candy",9);
2009/01/31 09:10:04 NB000I Source file "events.nb" included. size=1407
2009/01/31 09:10:04 NB000I NodeBrain nb[27523] terminating - exit code=0
$
```

Okay, now why did rules `r1`, `r2`, `r3`, and `r4` fire when they did? Each time a tuple is asserted to the cache, it updates a tree-of-trees structure as needed to retain the full set of asserted tuples. It also updates three counters at each node within the tree-of-trees: hits, rows, and kids. The cache module defines *hits* as the number of times an assertion arrives

at a given node within the tree, *rows* as the number of subordinate table rows represented by the subordinate tree-of-trees, and *kids* as the number of directly subordinate nodes—or the number of unique values in the first column of the subordinate table. In the definition of the "horace" cache node, you specify thresholds for hits using (), rows using `{}`, and kids using []. When a threshold is reached, the cache node alerts itself. The rules defined for the node handle the alerts. In the example above, when `r4` fired, it was responding to an invisible alert that would look something like the following if it were not invisible.

```
horace. alert quantity._hitState,quantity._hits=3,type="purchase",city="Paris", ...
... customer="Bruno",item="iPod",quantity=5;
```

This alert was triggered because the hit counter for the (`purchase,Paris,Bruno,iPod,5`) node in the cache reached the specified limit of 3 within a 24-hour interval. See if you can figure out why rules `r1`, `r2`, and `r3` fired when they did.

In this example, the rules just issue comments. In a real application, the rules would have taken action of some kind: an alert, assertion, alarm, shell command, and so on.

7.2 Tuple Expiration

A tuple in a cache can be retained for a defined time interval (e.g., 24 hours in the example above), until the end of a period (e.g., end of current minute, hour, day), until some state is detected, or indefinitely.

```
# interval
define horace node cache:(~(24h):type(20),city{10}[5],customer,item,quantity(3));
# end of period
define horace node cache(~(d)):(type(20),city{10}[5],customer,item,quantity(3));
# state a=1
define horace node cache(a=1):(type(20),city{10}[5],customer,item,quantity(3));
# indefinitely
define horace node cache:(type(20),city{10}[5],customer,item,quantity(3));
```

You can also remove a tuple or set of tuples from a cache at any time. The following command removes all tuples starting with (`"purchase","London"`).

```
horace. assert ?("purchase","London");
```

In addition to alerting when a threshold is reached, you can direct a cache to alert when a tuple expires. This enables us to assert a tuple to the cache and take action if you haven't asserted it again within the expiration period. In other words, you can use a cache to know when something hasn't happened for some interval.

```
#!/usr/local/bin/nb
# File: tutorial/cache/tardy.nb
define tardy node cache:(!~(6s):Source);
tardy. define r1 if(_action="expire"): $ # ${Source} has been quiet for ${_interval}
tardy. assert ("Fred");
```

When executed below, you are notified when "Fred" has not been asserted to the cache for 6 seconds.

```
./tardy.nb -
2009/01/31 10:38:06 NB000I Argument [1] ./tardy.nb
> #!/usr/local/bin/nb
> # File: tutorial/cache/tardy.nb
> define tardy node cache:(!~(6s):Source);
> tardy. define r1 if(_action="expire"): $ # ${Source} has been quiet for ${_interval}
> tardy. assert ("Fred");
> # Press the ENTER key once repeatedly until the rule fires
> # Should happen in 6 seconds
2009/01/31 10:38:06 NB000I Source file "./tardy.nb" included. size=290
2009/01/31 10:38:06 NB000I Argument [2] -
2009/01/31 10:38:06 NB000I Reading from standard input.
---------- --------
>
>
>
>
>
>
2009/01/31 10:38:12 NB000I Rule tardy.r1 fired
: tardy. # Fred has been quiet for 6 seconds
>
```

7.3 Event Sequence

A cache node can be used like a tree node for detecting a sequence of events. However, the cache node, having support for scheduled tuple expiration, can also support a timing condition. The `OnJust` cache below is used to remember for 5 seconds that a switch has been turned on. The `TurnedOn` rule asserts the name of a switch to the cache each time a switch is turned on. The `TurnedOff` rule responds to a switch being turned off if the cache still remembers the switch being turned on.

```
#!/usr/local/bin/nb
# File: tutorial/cache/sequence.nb
define OnJust node cache:(~(5s):switch);
define TurnedOn if(on) OnJust(switch);
define TurnedOff if(!on and OnJust(switch)):...
... $ # The ${switch} turned off within ${OnJust._interval} of turning on

# Sample events
alert on,switch="kitchen light";
alert on,switch="porch light";
alert !on,switch="kitchen light";
-sleep 6
alert !on,switch="porch light";
alert on,switch="porch light";
alert !on,switch="porch light";
```

In the execution below, notice that the `TurnedOff` rule did not respond when the porch light stayed on for 6 seconds.

```
$ ./sequence.nb
2009/01/31 11:23:14 NB000I Argument [1] ./sequence.nb
> #!/usr/local/bin/nb
> # File: tutorial/cache/sequence.nb
> define OnJust node cache:(~(5s):switch);
> define TurnedOn if(on) OnJust(switch);
> define TurnedOff if(!on and OnJust(switch)): ...
... $ # The ${switch} turned off within ${OnJust._interval} of turning on
> alert on,switch="kitchen light";
2009/01/31 11:23:14 NB000I Rule TurnedOn fired (OnJust(switch)=1)
> alert on,switch="porch light";
2009/01/31 11:23:14 NB000I Rule TurnedOn fired (OnJust(switch)=1)
> alert !on,switch="kitchen light";
2009/01/31 11:23:14 NB000I Rule TurnedOff fired
: # The kitchen light turned off within 5 seconds of turning on
> -sleep 6
[6410] Started: -sleep 6
[6410] Exit(0)
> alert !on,switch="porch light";
> alert on,switch="porch light";
2009/01/31 11:23:20 NB000I Rule TurnedOn fired (OnJust(switch)=1)
> alert !on,switch="porch light";
2009/01/31 11:23:20 NB000I Rule TurnedOff fired
: # The porch light turned off within 5 seconds of turning on
2009/01/31 11:23:20 NB000I Source file "./sequence.nb" included. size=453
2009/01/31 11:23:20 NB000I NodeBrain nb[6409] terminating - exit code=0
```

7.4 Event Correlation

A cache node is useful for event correlation where the goal is to detect repetition, variation, or sequence. Rules can be used to recognize input events and assert multiple attribute combinations (tuples) to different cache nodes to detect different patterns. A single cache can detect multiple patterns, but it often necessary to specify attributes in a different order in different cache nodes to detect all the required patterns. For example, a cache specified as (`child,action[5],toy`) could detect a given child taking a given action on five different toys, while a cache specified as (`toy[10],child[3],action(7)`) could detect when 10 different children performed a given action on a given toy, a given child performed three different actions on a given toy, and a given child performed a given action on a given toy seven times. The last condition could be detected by the first cache if you included another threshold (`child,action[5],toy(7)`). But the first two conditions detected by the second cache could not be detected by the first cache. It is necessary to use two different cache nodes with the attributes in a different order to detect all four conditions.

8 Baseline Node

> *Do not put your faith in what statistics say until you have carefully considered what they do not say.* —William W. Watt

A Baseline node is used to detect statistical anomalies. It does not tell us what is good, bad, or important. However, it can help to find conditions worthy of further investigation.

8.1 Creating Profile Directory

Create a directory for storing period profiles. Since you are going to create a statistical profile for weather measurements, name the directory `weather`, and create it in the `cache/baseline` subdirectory of the caboodle (NodeBrain application directory).

```
$ cd CABOODLE
$ mkdir -p cache/baseline/weather
```

8.2 Constant Expectation

Let's say you are monitoring the temperature and humidity inside a house where it is expected to be relatively constant over time. There is no need to manage multiple period profiles, nor is there a reason to learn a statistical definition of normal. Instead, you can use a single period with a static profile that you define manually.

```
# cache/baseline/weather/00000000.nb
.("temperature"):set 68,2;
.("humidity"):set 45,5;
```

In this case, define the Baseline node as follows.

```
# arguments: directory, weight, tolerance, cycle, period
define indoor node baseline("cache/baseline/weather",0,0.8,60,60):static;
```

Here's a quick explanation of your choices.

- The `static` option is used to avoid updating the profile at the end of a period.
- A weight of 0 is used to avoid adjusting the average based on new actual values. This is not technically required under the `static` option, but any other value would be misleading.
- A tolerance of 0.8 is specified so you can express the range of normal values in the same units as the measure. This may seem a bit odd, but because of the approximation of standard deviation, 0.8 standard deviation is 1 average deviation. Since the profile contains average deviation, you can set it to half the normal range and set the average to the center of the normal range.
- A one-hour (60 minute) cycle and period are used so only one period profile is needed. Since the period duration divides into the cycle duration one time, keep repeating a single period.

8.3 Asserting Values

There is nothing special about the way to make assertions to a Baseline node. You assert values in the same way you would assert to a Tree node. Here are some examples.

```
indoor. assert ("temperature")=67.4,("humidity")=45.3;
assert x=1,y="abc",indoor("temperature")=67.4,indoor("humidity")=45.3;
indoor. assert x=2,("temperature")=73;
```

8.4 Anomaly Response Rules

Given the last assertion above, where temperature is asserted to be 73, you may expect the following alert to be generated by the "indoor" node shown previously.

```
alert _measure="'temperature'",_value=73,_average=68,_sigma=2.5,_deviation=5, \
      _threshold=5,_level=1;
```

By itself this alert does nothing. You need to provide rules to specify a response. Here's an example that simply echoes an entry out to an application log, and when it is a temperature anomaly also logs to the system log.

```
# arguments: directory, weight, tolerance, cycle, period
define indoor node baseline("cache/baseline/weather",0,0.8,60,60):static;
# alert _measure="...",_value=n,_average=n,_sigma=n,_deviation=n,_threshold=n,_level=n;
indoor. define logecho if(1):$ =echo `date` "${_measure} value of ${_value} is out of  \
                                    range - threshold is ${_threshold}" >> weather.log
indoor. define logger if(_measure="'temperature'"):$ =logger -p local0.notice -t WEATHER \
                                     "${_measure}=${_value} threshold=${threshold}"
```

Here's another example where the response is more complicated. The `alarmit` rule invokes the `alarm` macro, which is defined on the third line. The macro makes an assertion to the `alarmMessage` cache. The cache logs a message to the system log, but only when the measure has had no anomaly alerts for at least 1 hour prior. The `cacheit` rule asserts the measure name to the `anomaly` cache, which can be used to check whether a measure has had an anomaly alert in the past 2 hours.

```
define alarmMessage node cache(~(h):measure,message(1));
alarmMessage. define alarmit if(message._hitState):$ =logger -p local0.notice -t WEATHER \
                                                                    "${message}"

define alarm macro(msgid,measure,text):% alarmMessage. assert ("%{measure}","%{msgid} %{text}");

# keep track of anomalies within the past 2 hours
define anomaly node cache:(~(2h):measure);

# arguments: directory, weight, tolerance, cycle, period
define indoor node baseline("cache/baseline/weather",0,0.8,60,60):static;
# alert _measure="...",_value=n,_average=n,_sigma=n,_deviation=n,_threshold=n,_level=n;
indoor. define alarmit if(1):$ $alarm("XYZ0001","${_measure}", \
                   "Weather measure ${_measure}=${_value} threshold=${_threshold}");
indoor. define cacheit if(1) anomaly(_measure);
```

8.5 Periodicity

Now let's suppose you live in a barn, where the temperature and humidity are not so constant, but where the temperature values are expected to vary by time of day. You may want a profile for every 4-hour period during the day. There would be six periods per day. The period profiles are stored in files with names incrementing by 4*60*60, or 14400 seconds.

Period	**Profile**
00:00 to 04:00	00000000.nb
04:00 to 08:00	00014400.nb
08:00 to 12:00	00028800.nb
12:00 to 16:00	00043200.nb
16:00 to 20:00	00057600.nb
20:00 to 00:00	00072000.nb

The Baseline node now has a 24-hour cycle time and a 4-hour period.

```
# arguments: directory, weight, tolerance, cycle, period
define indoor node baseline("cache/baseline/weather",0,0.8,24*60,4*60):static;
```

8.6 Learning

Suppose you don't have a preconceived notion of what a normal range of values is for your measures. In this case, you want to let the Baseline module create the profiles for you and adjust the average value and average deviation based on actual values experience while learning is enabled. To do this, simply remove the `static` option from the Baseline node definition and provide a non-zero weight to control how fast to adapt averages to new actuals. Because you no longer know the range of normal values, you express the tolerance in standard deviations (e.g., 3 below).

```
# arguments: directory, weight, tolerance, cycle, period
define indoor node baseline("cache/baseline/weather",0.2,3,24*60,4*60);
```

At the end of each period, the profile is updated with weight-adjusted averages for each measure's value and deviation.

8.7 Computer Network Periods

Measurements within a computer network (e.g., number of packets by protocol) typically vary based on user and software schedules that vary by time of day and day of week. If you are monitoring measures with this type of periodicity, it is appropriate to use a weekly cycle and hourly period.

```
# arguments: directory, weight, tolerance, cycle, period
define packets node baseline("cache/baseline/packetsbyprotocol",0.2,3,7*24*60,60);
```

8.8 Summation

When your measures are counts (e.g., number of packets by protocol), your information source may provide counts over units of time much smaller than the period duration. In this case, you want to sum up the counts from the information source over the period duration. You do this by specifying the `sum` option.

```
# arguments: directory, weight, tolerance, cycle, period
define packets node baseline("cache/baseline/packetsbyprotocol",0.2,3,7*24*60,60):sum;
```

Now when you assert values to measures within the Baseline node, the measures are incremented instead of being set to the new value.

```
packets. assert ("icmp")=465,("http")=1024,("smtp")=34; # increment measures
```

At the end of each period, the measures are all reset to zero when the `sum` option is used.

9 Servant Node

> *In order to become the master, the politician poses as the servant.* —Charles de Gaulle (1890–1970)

NodeBrain is not intended to be the master of all things. As in politics, it is often more convenient to let a servant be the master. In this tutorial, you will learn how to create a servant in your favorite programming language to obtain information needed to make decisions. You will see that when a servant sends commands to NodeBrain, it becomes the master—like a politician once elected.

9.1 Creating a Servant Program

To keep it simple and only hint at something useful, let's create a servant script using Perl that pretends to tell you the cost of gas and bread, something every politician should be prepared to include in a campaign speech.

```
#!/usr/bin/perl
# File: tutorial/Servant/charles.pl
my $gas=2.50;
my $bread=1.10;
$|=1;
while(<>){
  chomp($_);
  if(/gas/){print("assert gas=$gas;\n");$gas+=.50;}
  elsif(/bread/){print("assert bread=$bread;\n");$bread+=.25}
  else{print("alert msg=\"item '$_' not recognized\";\n");}
  }
```

9.2 Specifying a Servant Node

Now you need some rules to use the servant program. Create a script that looks like this.

```
#!/usr/local/bin/nb -s
# File: tutorial/Servant/charles.nb
define price node servant:|=|:./charles.pl
define ouch on(gas>4 or bread>3):stop;
define getgasprice on(~(3s)):price:gas
define getbreadprice on(~(3s)):price:bread
```

The Servant node module specification includes an `=` command to specify the program and what to do with `stdout` and `stderr`. It also supports a leading pipe (`|`) to enable the sending of text to the program on `stdin`.

9.3 Execution

This script is designed to run like an agent without detaching from the terminal. The `-s` option is the trick. The script will pause for 3 seconds between scheduled events, so just be patient and the script will end when the price of one of the items gets too painful.

```
$ ./charles.nb
2008/08/21 19:08:28 NB000I Argument [2] ./charles.nb
> #!/usr/local/bin/nb -s
> # File: tutorial/Servant/charles.nb
> define price node servant:|=|:./charles.pl
> define ouch on(gas>4 or bread>3):stop;
> define getgasprice on(~(3s)):price:gas
> define getbreadprice on(~(3s)):price:bread
2008/08/21 19:08:28 NB000I Source file "./charles.nb" included. size=194
2008/08/21 19:08:28 NB000T Servant mode selected
---------- ----------------------------------------------------
2008/08/21 19:08:28 NM000I servant price: Enabling|=|:./charles.pl
2008/08/21 19:08:28 NM000I servant price: Enabled[21633] |=|:./charles.pl
2008/08/21 19:08:31 NB000I Rule getbreadprice fired
: price:bread
2008/08/21 19:08:31 NB000I Rule getgasprice fired
: price:gas
> price. assert bread=2.1;
> price. assert gas=3.25;
2008/08/21 19:08:34 NB000I Rule getbreadprice fired
: price:bread
2008/08/21 19:08:34 NB000I Rule getgasprice fired
: price:gas
> price. assert bread=2.35;
> price. assert gas=3.75;
2008/08/21 19:08:37 NB000I Rule getbreadprice fired
: price:bread
2008/08/21 19:08:37 NB000I Rule getgasprice fired
: price:gas
> price. assert bread=2.6;
> price. assert gas=4.25;
2008/08/21 19:08:37 NB000I Rule ouch fired
: stop;
2008/08/21 19:08:37 NB000I [21633] Killed(1)
2008/08/21 19:08:37 NB000I NodeBrain nb[21632] terminating - exit code=0
```

10 Translator Node

> *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.

10.1 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.
```

10.2 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 nth 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.

10.3 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`.

10.4 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.

11 Audit Node

> *If Edison had a needle to find in a haystack, he would proceed at once with the diligence of the bee to examine straw after straw until he found the object of his search.... I was a sorry witness of such doings, knowing that a little theory and calculation would have saved him ninety per cent of his labor.* —Nikola Tesla (1857–1943), *New York Times*, October 19, 1931

Effective review of system and application logs can be like trying to find a needle in a haystack. It requires at least one Edison and one Tesla working as a team. The Audit node module works like Edison when reviewing logs so you can work like Tesla.

An Audit node is similar to the Translator node covered in an earlier tutorial, but differs in the way lines of text are input for translation. An Audit node starts at the end of a log file and periodically checks for new lines to translate. When a log file rolls, the Audit node starts at the beginning of the new log file.

The content of system and application log files can vary significantly depending on the mix of applications on a system and how they are configured. A good strategy is to treat log entries as worthy of investigation by default. Duplicate suppression and other flood protection techniques are helpful when using this strategy. As new log entries are reported and investigated, you can decide if they should be suppressed or handled in a special way.

11.1 Agent Rules

The `syslog.nb` file below provides an agent with an Audit node to monitor a log file called `syslog`. It specifies a translator named `syslog.nbx` and a polling interval of 10 seconds. (A longer interval is recommended for real applications.) It also includes a deduplication cache.

```
#!/usr/local/bin/nb -d
# File: tutorial/Audit/syslog.nb
set log="syslog.log",out=".";
define syslog node cache(~(h(8))):(~(1h):route,appl,group,node,object,severity,text(1));
syslog. define alarm if(text._hitState):$ -|mail.form \
  source=tutorial route="${route}" appl="${appl}" group="${group}" \
  node="${node}" severity="${severity}" text="${text}" >> mail.log
syslog. define audit node audit("syslog","syslog.nbx",~(10s));
```

11.2 Sample Log

Here's a small sample of a log file in the format to use for this tutorial. A copy of this file is stored as `tutorial/Audit/syslog.sample`.

```
Feb  1 19:00:04 smk001 sshd[3972]: Accepted publickey for myuser \
  from ::ffff:192.168.1.100 port 53403 ssh2
Feb  1 19:00:06 smk001 sshd[3980]: Accepted publickey for myuser \
  from ::ffff:192.168.1.101 port 53410 ssh2
Feb  1 19:00:16 smk001 kernel: z90crypt: probe_crypto_domain -> \
  Unable to find crypto domain: No devices found
Feb  1 19:00:46 smk001 kernel: z90crypt: probe_crypto_domain -> \
  Unable to find crypto domain: No devices found
Feb  1 19:01:16 smk001 kernel: z90crypt: probe_crypto_domain -> \
  Unable to find crypto domain: No devices found
Feb  1 19:01:19 smk001 su: (to root) myuser on /dev/pts/1
```

11.3 Translation Rules

Here's a small translator called `syslog.nbx` designed for the log format above.

```
# File: tutorial/Audit/syslog.nbx
([a-zA-Z]+ +\d+ \d\d:\d\d:\d\d [^ ]+ ){
  (^-- MARK --)
  (^\/USR\/SBIN\/CRON\[\d+\]: [^ ]+ CMD)
  (^last message repeated \d+ times)
  (^kernel: ){
    (^z90crypt: probe_crypto_domain -> Unable to find crypto domain: No devices found)
    (^end_request: I\/O error)
    (^dasd_erp.*:Perform logging requested)
    (^dasd:.*:ERP successful)
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 kernel: $[=]");
    }
  (^su: ){
    (^pam_unix2: session (started|finished) for user (nobody|root|wwwadm|cyrus), service su)
    (^\(to (nobody|cyrus)\) root on none)
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 su: $[=]");
    }
  (^sshd\[\d+\]: ){
    (^Accepted password for myuser from ::ffff:.* port \d+ ssh2)
    (^Accepted publickey for myuser from ::ffff:.* port \d+ ssh2)
    (^error: Could not get shadow information for NOUSER)
    (^(?'preport'.*port) \d+ ){
      :syslog. assert ("syslog","syslog","OS","","","normal",\
         "SYS0000 sshd[*]: $[preport]port * $[=]");
      }
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 sshd[*]: $[=]");
    }
  }
():syslog. assert ("syslog","syslog","OS","","","critical","SYS0000 $[-]");
```

11.4 Start the Agent

Before starting the agent, you must touch `syslog` to make sure it exists.

```
$ ./syslog.nb
2009/02/01 19:56:57 NB000I Argument [1] -d
2009/02/01 19:56:57 NB000I Argument [2] ./syslog.nb
> #!/usr/local/bin/nb -d
> # File: tutorial/Audit/syslog.nb
> set log="syslog.log",out=".";
2009/02/01 19:56:57 NB000I NodeBrain nb will log to syslog.log
> define syslog node cache(~(h(8))):(~(1h):route,appl,group,node,object,severity,text(1));
> syslog. define alarm if(text._hitState):$ -|mail.form \
    source=tutorial route="${route}" appl="${appl}" group="${group}" \
    node="${node}" severity="${severity}" text="${text}" >> mail.log
> syslog. define audit node audit("syslog","syslog.nbx",~(10s));
2009/02/01 19:56:57 NB000I Loading translator "syslog.nbx"
---------- --------
# File: tutorial/Audit/syslog.nbx
([a-zA-Z]+ +\d+ \d\d:\d\d:\d\d [^ ]+ ){
  (^-- MARK --)
  (^\/USR\/SBIN\/CRON\[\d+\]: [^ ]+ CMD)
  (^last message repeated \d+ times)
  (^kernel: ){
    (^z90crypt: probe_crypto_domain -> Unable to find crypto domain: No devices found)
    (^end_request: I\/O error)
    (^dasd_erp.*:Perform logging requested)
    (^dasd:.*:ERP successful)
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 kernel: $[=]");
    }
  (^su: ){
    (^pam_unix2: session (started|finished) for user (nobody|root|wwwadm|cyrus), service su)
    (^\(to (nobody|cyrus)\) root on none)
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 su: $[=]");
    }
  (^sshd\[\d+\]: ){
    (^Accepted password for myuser from ::ffff:.* port \d+ ssh2)
    (^Accepted publickey for myuser from ::ffff:.* port \d+ ssh2)
    (^error: Could not get shadow information for NOUSER)
    (^(?'preport'.*port) \d+ ){
      :syslog. assert ("syslog","syslog","OS","","","normal",\
         "SYS0000 sshd[*]: $[preport]port * $[=]");
      }
    :syslog. assert ("syslog","syslog","OS","","","normal","SYS0000 sshd[*]: $[=]");
    }
  }
():syslog. assert ("syslog","syslog","OS","","","critical","SYS0000 $[-]");
---------- --------
2009/02/01 19:56:57 NB000I Translator "syslog.nbx" loaded successfully.
2009/02/01 19:56:57 NB000I Source file "./syslog.nb" included. size=425
2009/02/01 19:56:57 NB000I NodeBrain nb[6749,3352] daemonizing
$
```

11.5 Grow the Log

Use a shell script to append the sample log file `syslog.sample` to the monitored `syslog` file three times.

```
# File: tutorial/Audit/syslog.sh
cat syslog.sample >> syslog
cat syslog.sample >> syslog
cat syslog.sample >> syslog
```

11.6 Review the Agent Log

After executing the `syslog.sh` script above, your `syslog.log` file should look like this.

```
2009/02/01 19:59:35 NB000I Agent log is syslog.log
2009/02/01 19:59:35 NM000I audit audit: Enabled audit of syslog using syslog.nbx
> syslog. assert ("syslog","syslog","OS","","","normal",\
    "SYS0000 su: (to root) myuser on /dev/pts/1");
2009/02/01 20:00:05 NB000I Rule syslog.alarm fired
: syslog. -|mail.form source=tutorial route="syslog" appl="syslog" group="OS" \
    node="" severity="normal" text="SYS0000 su: (to root) myuser on /dev/pts/1" >> mail.log
[6957] Started: -|mail.form source=tutorial route="syslog" appl="syslog" group="OS" \
    node="" severity="normal" \
    text="SYS0000 su: (to root) myuser on /dev/pts/1" >> mail.log
[6957] Exit(0)
> syslog. assert ("syslog","syslog","OS","","","normal", \
    "SYS0000 su: (to root) myuser on /dev/pts/1");
> syslog. assert ("syslog","syslog","OS","","","normal", \
    "SYS0000 su: (to root) myuser on /dev/pts/1");
> syslog. assert ("syslog","syslog","OS","","","normal", \
    "SYS0000 su: (to root) myuser on /dev/pts/1");
```

Notice that the translator only asserted the `su` log entries to the `syslog` cache. This is because the translator was coded to ignore the other entries in the sample log file. Notice also that the agent only generated an alarm once. This is because the `syslog` cache node was coded to ignore duplicates until 8:00 AM unless they are separated by 1 hour.

Although you used a system log for this tutorial, you can code a translator for application logs as well. In fact, the more unique your application log, the more likely you will need to construct your own custom log monitor.

If you started the agent for this tutorial, you should kill it now.

12 Syslog Node

> *I decided that it was not wisdom that enabled [poets] to write their poetry, but a kind of instinct or inspiration, such as you find in seers and prophets who deliver all their sublime messages without knowing in the least what they mean.*
> —Socrates (469 BC–399 BC), in "Apology," sct. 21, by Plato

If your sublime messages are delivered via the syslog protocol, you may prefer to use a Syslog node instead of an Audit node. This enables NodeBrain to respond immediately to arriving syslog UDP packets without waiting to poll a log file.

You have a couple options. You can configure your Syslog node to listen on UDP port 514 on a server that doesn't already have a syslog daemon. However, if you need (or want) more flexibility, I recommend that you use NodeBrain in combination with `syslog-ng`. In that case, you configure NodeBrain to listen on a different port and configure `syslog-ng` to forward all or selected syslog entries to NodeBrain.

The example below is configured to listen on UDP port 1514, assuming `syslog-ng` is used to forward syslog to NodeBrain.

```
#!/usr/local/bin/nb -d
# File: tutorial/Syslog/syslog.nb
-rm syslog.log
set log="syslog.log",out=".";
define syslog node cache(~(h(8))):(~(1h):route,appl,group,node,object,severity,text(1));
syslog. define alarm if(text._hitState):$ -|mail.form ...
... source=tutorial route="${route}" appl="${appl}" group="${group}" ...
... node="${node}" severity="${severity}" text="${text}" >> mail.log
syslog. define audit node syslog("syslog.nbx",1514);
```

You should reference the documentation for `syslog-ng` to see how to configure it to forward to NodeBrain. Here's an example to get you started.

```
destination nodebrain { udp("localhost" port(1514)); };
filter f_nodebrain { host("(humpty|dumpty).mydomain.com|franklin.otherdomain.com"); };
log { source(src);  filter(f_nodebrain); destination(nodebrain); };
```

Refer to the *Audit Node* tutorial above for a sample `syslog.nbx` file. Refer to the *Translator Node* tutorial for more information on coding a translator.

13 Snmptrap Node

> *Man is the only kind of varmint who sets his own trap, baits it, then steps on it.* —John Steinbeck (1902–1968)

The Snmptrap node is used to monitor SNMP traps. This is one method of configuring NodeBrain to accept alerts from monitoring tools that are capable of sending SNMP traps. NodeBrain's Snmptrap node is a bit unusual in that it does not use MIBs. Instead, each trap is converted into a NodeBrain alert using single quoted OID terms.

```
alert 'oid'="value",'oid'="value",...;
```

You must then code your rules referencing the *oid* terms. However, I recommend you define aliases for the OIDs of interest to make your rules more readable.

```
#!/usr/local/bin/nb -d
# File: tutorial/Snmptrap/snmptrap.nb
-rm snmptrap.log
set log="snmptrap.log",out=".";
# Node
define snmptrap node snmptrap:trace,dump;
# Aliases
snmptrap. define myProduct cell '1.3.6.1.6.3.1.1.4.3'="1.3.6.1.4.1.1279";
snmptrap. define address cell '1.3.6.1.4.1.1279.4';
snmptrap. define type    cell '1.3.6.1.4.1.1279.5';
# Rules
snmptrap. define r1 if(myProduct and type="hiccup");
```

The example above is only provided to illustrate the syntax for working with single quoted OID terms. You will need to adapt this example to the traps you want to monitor to construct useful rules. However, you can use this example to start collecting traps right away. The traps will show up in your `snmptrap.log` file. Then, you can figure out what you want to monitor.

Remove the `:trace,dump` from your Snmptrap node specification to reduce the amount of information in your log.

```
define snmptrap node snmptrap;
```

You may use a `silent` option to stop logging the alerts generated by the Snmptrap node.

```
define snmptrap node snmptrap:silent;
```

14 Pipe Node

> *To listen closely and reply well is the highest perfection we are able to attain in the art of conversation.* —Francois de La Rochefoucauld (1613–1680)

To construct a successful NodeBrain application, you must configure NodeBrain to listen closely and respond well. The *Operating Mode* tutorial" illustrates how NodeBrain can listen for input commands from `stdin` and both input commands and error messages from servant scripts. NodeBrain's ability to listen is controlled by a component called the *medulla.* Node modules interface with the medulla to extend NodeBrain's ability to listen to include other sources of input.

A pipe node is perhaps the simplest of listening nodes. It listens to a FIFO (named pipe) file, and you can write to the pipe using any program you like, including an `echo` command.

14.1 Defining Pipe Server Nodes

To begin this tutorial, createan agent script called `smokey.nb` with two pipe server nodes as shown here.

```
#!/usr/local/bin/nb -d
# File: tutorial/Pipe/smokey.nb
-rm smokey.log
setlog="smokey.log",out=".";
declare jed identity guest;
declare chief identity owner;
define corncob node pipe.server("jed@corncob");
corncob.define r1 on(a=1 and b=2);
define peace node pipe.server("chief@peace");
peace.define r1 on(a=1 and b=2);
```

The argument to a `pipe.server` is of the form "*identity*`@`*pipe*". Identities are associated with listening nodes to limit the types of commands the interpreter will accept from the node. In this example, you associate the identity `jed` with the corncob pipe, and the identity `chief` with the peace pipe. You have declared `jed` to be a guest and `chief` to be an owner. A guest can only connect and issue `show` commands—like read only access. An owner can issue any command, including shell commands, which means they have all permissions of the user that started the agent. For this reason, pipes are created with owner-only read/write permissions. However, if you declare the associated identity to be a "peer," you can give other user's write permission on the pipe. They will be able to issue assertions and alerts but not modify your rules or issue shell commands. You must still think through how the agent will respond to their assertions. For example, if you create a rule that reboots the system when `a=1`, then letting someone assert `a=1` is the same as letting them reboot the system.

14.2 Starting Pipe Server Agent

Now let's start the agent. This script is executable because the `-d` (daemon) option is specificied on the she-bang line. So you can just execute it like any executable and it will load the rules and go into the background (daemonize).

```
$ ./smokey.nb
2008/06/1017:09:16 NB000I Argument [1] -d
2008/06/1017:09:16 NB000I Argument [2] ./smokey.nb
> #!/usr/local/bin/nb -d
> # File: smokey.nb
> -rm smokey.log
[13993]Started: -rm smokey.log
[13993]Exit(0)
> set log="smokey.log",out=".";
2008/06/1017:09:16 NB000I NodeBrain nb will log to smokey.log
> declare jed identity guest;
> declare chief identity owner;
> define corncob node pipe.server("jed@corncob");
> corncob. define r1 on(a=1 and b=2);
> define peace node pipe.server("chief@peace");
> peace. define r1 on(a=1 and b=2);
2008/06/1017:09:16 NB000I Source file "./smokey.nb" included. size=323
2008/06/1017:09:16 NB000I NodeBrain nb[13992,4118] daemonizing
$
```

14.3 Sending Commands to Pipe Server Nodes

Use the `echo` command to send NodeBrain commands to the pipe servers.

```
$ echo stop > corncob
$ echo stop > peace
```

The log file shows you what happened.

```
$ cat smokey.log

N o d e B r a i n   0.8.17 (Dopey) 2014-04-10

Compiled Jul  6 2014 21:00:54 x86_64-unknown-linux-gnu

Copyright (C) 2014 Ed Trettevik <eat@nodebrain.org>
MIT or NodeBrain License
----------------------------------------------------------------

/usr/local/bin/nb -d ./smokey.nb

Date       Time     Message
---------- -------- ------------------------------------------------
2014/06/10 17:09:16 NB000I NodeBrain nb[13994:1] myuser@myhost
2014/06/10 17:09:16 NB000I Agent log is smokey.log
2014/06/10 17:09:16 NM000I pipe.server peace: Listening for FIFO connections as chief@peace
2014/06/10 17:09:16 NM000I pipe.server corncob: ...
... Listening for FIFO connections as jed@corncob
2014/06/10 17:41:11 NM000I pipe.server corncob: FIFO jed@corncob
> corncob. stop
2014/06/10 17:41:11 NB000E Identity "jed" does not have authority to issue stop command.
2014/06/10 17:41:19 NM000I pipe.server peace: FIFO chief@peace
> peace. stop
2014/06/10 17:41:19 NB000I NodeBrain nb[13994] terminating - exit code=0
$
```

Notice that when you sent a `stop` command to the corncob pipe, you didn't have the needed authority, but when you sent the same `stop` command to the peace pipe it worked.

Run `smokey.nb` again with the following commands to see what happens.

```
$ ./smokey.nb
$ echo "common. define r1 on(a=1 and b=2);" > peace
$ echo "common. assert a=1,b=3;" > peace
$ echo "common. assert b=2;" > peace
$ echo "common. show -t" > peace
$ echo "stop" > peace
$ cat smokey.log
```

15 Peer Node

> *The more elaborate our means of communication, the less we communicate.*
> —Joseph Priestly (1733–1804)

The peer module is more elaborate than some of the other modules used for communication. The goal is to communicate less, or at least to be more selective in with whom you communicate. This selectivity is accomplished with encryption and key-based authentication.

The files for this tutorial are in the `tutorial/Peer` directory. The NodeBrain scripts are executable and I have left off the `.nb` suffix for fun.

15.1 Peer Module Identity Keys

Before you communicate using the peer module, you must create keys to be used by clients and servers. To reduce complexity for this example, you'll create a single key and use it for both the client and the server as a "shared secret" key. Here's a script called `genkey` that creates a key for an identity called `buddy`.

```
#!/usr/local/bin/nb
# File: tutorial/Peer/genkey
define myService node peer.service;
myService:identify buddy;
```

The peer module provides a skill called `service` that supports some helpful commands associated with peer communication. Use the `identify` command here. This command generates a key and places it in a key store for later reference.

Unix and Linux: `~/.nb/nb_peer.keys`
Windows: `USER_PROFILE/ApplicationData/NodeBrain/nb_peer.keys`

The key store is readable only by the owning user and looks like this.

```
$ cat ~/.nb/nb_peer.keys
foo 3.3788e45e8f64776b.0.0;
bar 7.b49ad9bfb68a97b8.fab67908064b5cb3.0;
buddy 3.be5d8bc4465d3aa7.bd6107c786f72c15.0;
```

15.2 Peer Module Server Node

> *Be alert to give service. What counts a great deal in life is what we do for others.* —Anonymous

A peer server node is alert to give service, in fact, it can even give service to alerts. It provides a method for a peer client to send any command to a NodeBrain agent: alerts, assertions, new rule definitions, and so on. It accepts TCP/IP socket connections from clients and issues received commands in the context of the server node. The `server` file in the `tutorial/Peer` directory looks like this.

```
#!/usr/local/bin/nb -d
# File: tutorial/Peer/server
-rm server.log
set out=".",log="server.log";
declare buddy identity owner;
define myServer node peer.server("buddy@./socket");          # Unix domain
#define myServer node peer.server("buddy@127.0.0.1:12345");  # local
#define myServer node peer.server("buddy@0.0.0.0:12345");    # remote
myServer. define r1 on(a=1 and b=2);
```

Declare an identity `buddy` that you rank as `owner`. This is all the interpreter knows about `buddy`. Use this same name, `buddy`, in the specification of the peer server node. The peer node module requires that a key exist in the key store for this identity, which is why you created it in the previous section.

Let's start up the peer server.

```
$ ./server
2008/06/11 10:27:58 NB000I Argument [1] -d
2008/06/11 10:27:58 NB000I Argument [2] ./server
> #!/usr/local/bin/nb -d
> # File: tutorial/Peer/server
> -rm server.log
[20642] Started: -rm server.log
[20642] Exit(0)
> setout=".",log="server.log";
2008/06/11 10:27:58 NB000I NodeBrain nb will log to server.log
> declare buddy identity owner;
> define myServer node peer.server("buddy@./socket");          # Unix domain
2008/06/11 10:27:58 NB000I Peer keys loaded.
> #define myServer node peer.server("buddy@127.0.0.1:12345"); # local
> #define myServer node peer.server("buddy@0.0.0.0:12345");   # remote
> myServer. define r1 on(a=1 and b=2);
2008/06/11 10:27:58 NB000I Source file "./server" included. size=493
2008/06/11 10:27:58 NB000I NodeBrain nb[20641,19542]daemonizing
$
```

You are using a Unix domain socket in this tutorial because you don't need to communicate with remote clients. To experiment with serving remote clients, you can comment out the active `myServernode` and uncomment the remote myServer node. But first you should experiment with the client in the next section.

15.3 Peer Module Client Node

> *He who is always his own counselor will often have a fool for his client.* — Hunter S. Thompson (1937–2005)

The peer module enables NodeBrain to play the part of both server and client. This is not foolish because I am talking about different instances of NodeBrain, different processes ("skulls"), playing the roles of client and server.

The `client` file in the `tutorial/Peer` directory looks like this.

```
#!/usr/local/bin/nb
# File: tutorial/Peer/client
declare buddy identity;
define myClient node peer.client("buddy@./socket");          # Unix domain
#define myClient node peer.client("buddy@localhost:12345"); # local
#define myClient node peer.client("buddy@myhost.mydomain:12345"); # remote
myClient:assert a=1,b=2;
myClient:stop;
```

Notice you declare the identity `buddy` and specify the client just like you specified the server in the previous section. Here there is no requirement to give `buddy` local permissions.

Because you are going to run this client on the same machine as the server and under the same user account, the client and server will use the same key store. To run the client from a different account or machine, you would have to copy the `buddy` key from the server's key store to the client's key store. You could also configure different private keys for servers and clients and copy their public keys to the key stores of their peers. That approach provides better security, but quick success in this tutorial is more important, so stick with shared secret keys.

Now you can run the client and see what happens.

```
$ ./client
2008/06/11 10:28:00 NB000I Argument [1] ./client
> #!/usr/local/bin/nb
> # File: tutorial/Peer/client
> declare buddy identity;
> define myClient node peer.client("buddy@./socket");          # Unix domain
> #define myClient node peer.client("buddy@localhost:12345"); # local
> #define myClient node peer.client("buddy@myhost.mydomain:12345"); # remote
> myClient:assert a=1,b=2;
2008/06/11 10:28:00 NB000I Peer keys loaded.
2008/06/11 10:28:00 NB000I Peer b0000=buddy@./socket
2008/06/11 10:28:00 NB000I Rule myServer.r1 fired
> myClient:stop;
2008/06/11 10:28:00 NB000I Peer b0000=buddy@./socket
2008/06/11 10:28:00 NB000I Source file "./client" included. size=450
2008/06/11 10:28:00 NB000I NodeBrain nb[20644] terminating- exit code=0
$
```

The command `myClient:assert a=1,b=2` sends the command `assert a=1,b=2` to the server specified as `buddy@./socket`. Notice the message `Rule myServer.r1 fired`. There is no `myServer.r1` defined in the client. This is what happened at the server. When a peer client issues a command to a peer server, the server lets the client listen in as it reacts to the command. You can look at it from the server's point of view by displaying the agent log, in this case called `server.log`.

```
$ cat server.log

N o d e B r a i n   0.8.17 (Dopey) 2014-04-10

Compiled Jul  6 2014 21:00:54 x86_64-unknown-linux-gnu

Copyright (C) 2014 Ed Trettevik <eat@nodebrain.org>
MIT or NodeBrain License
------------------------------------------------------------------

nb -d ./server

Date       Time     Message
---------- -------- ------------------------------------------------
2014/06/11 10:27:58 NB000I NodeBrain nb[20643:1] myuser@myhost
2014/06/11 10:27:58 NB000I Agent log is server.log
2014/06/11 10:27:58 NM000I peer.server myServer: ...
... Listening for NBP connections as buddy@./socket
2014/06/11 10:28:00 NM000I peer.server myServer: buddy@./socket
> myServer. assert a=1,b=2;
2014/06/11 10:28:00 NB000I RulemyServer.r1 fired
2014/06/11 10:28:00 NM000I peer.server myServer: buddy@./socket
> myServer. stop;
2014/06/11 10:28:00 NB000I NodeBrain nb[20643] terminating - exit code=0
$
```

At 10:27:58 the server started listening for connections. At 10:28 it received a connection and issued the command `assert a=1,b=2` in the `myServer` context. This triggered rule `myServer.r1`, which has no action. In previous tutorials, you've learned enough to modify the rules in server to provide an action and try it again.

Notice the client sends a `stop` command to the server, which stops it because you've given the client full owner permissions. This means you have to restart the server each time you run the client. Only in a tutorial would you do something this silly.

15.4 Interactive Peer Module Client

If you want to experiment further with the peer module, restart the server and try the `iclient` file instead of `client`.

```
$ ./server
$ ./iclient
```

The `iclient` script looks like this.

```
#!/usr/local/bin/nb -‘myClient:
# File: iclient
declare buddy identity;
define myClient node peer.client("buddy@./socket");         # Unix domain
```

The strange looking she-bang ("#!") line has an argument that supplies an interactive command prefix and enables an option to automatically go into interactive mode after processing all command arguments. This step causes your prompt to look like this.

```
myClient:>
```

Any command you enter is now prefixed by the value `myClient:`, causing all your commands to be directed to your peer client node, which sends them to your peer server. Enter the highlighted text when prompted to get the same results as shown below.

```
$ ./iclient
2008/06/11 16:48:39 NB000I Argument [1] -'myClient:
> #!/usr/local/bin/nb -'myClient:
> # File: tutorial/Peer/iclient
> declare buddy identity;
> define myClient node peer.client("buddy@./socket");         # Unix domain
2008/06/11 16:48:39 NB000I Source file "./iclient" included. size=209
2008/06/11 16:48:39 NB000I Reading from standard input.
---------- --------
myClient:> show r1
2008/06/11 16:48:45 NB000I Peer keys loaded.
2008/06/11 16:48:45 NB000I Peer b0000=buddy@./socket
> myServer. show r1
r1 = ! == on((a=1)& (b=2));
myClient:> assert a=1,b=2;
2008/06/11 16:49:10 NB000I Peer b0000=buddy@./socket
> myServer. assert a=1,b=2;
2008/06/11 16:49:10 NB000I Rule myServer.r1 fired
myClient:> 'foo.
foo.> '
> quit
2008/06/11 16:49:15 NB000I NodeBrain nb[23715] terminating - exit code=0
$
```

This example illustrates how a single quote at the beginning of an interactive command can be used to change the command prefix.

The little tricks illustrated in this section are features of the interpreter, not the peer module, but when combined with the peer module make it a bit easier to use NodeBrain as a primitive interactive client to a NodeBrain agent.

16 Webster Node

> *The world is governed more by appearance than realities, so that it is fully as necessary to seem to know something as to know it.* —Daniel Webster (1782–1852)

The Webster module enables NodeBrain to pretend to be a web server—not with the goal of providing a web server for web applications in general but to support little web tools associated with NodeBrain applications.

Although Webster supports x509 certificate authentication, let's skip over all that fun stuff and get it running quickly with no security. If you don't secure it with password or certificate authentication, Webster doesn't let you do anything other than display web pages, so other than exposing a port that could be vulnerable to buffer overflow errors if NodeBrain has bugs, there is no risk in running it without security.

For this tutorial, the files are in the `tutorial/Webster` subdirectory of the distribution directory. The server script `webster1` looks like this.

```
#!/usr/local/bin/nb -d
# File: tutorial/Webster/webster1
-rm webster.log
set out=".",log="webster.log";
define webster node webster;
webster. define uri cell "http://0.0.0.0:62443");
webster. define Authenticate cell "no";  # Default is "yes"
```

A Webster server is specified as *`identity@interface:port`*. You are using the `default` identity, all interfaces "0.0.0.0", and port 62443. The "443" reminds you that you'd prefer to use HTTPS in a real application. Change the port number if 62443 is used on your system.

If there are no errors in the log file, you can leave your server running and connect with your web browser using the following URL.

http://*hostname***:62443**

This tutorial continues on the page displayed by your browser.

Index