Programmers Guide

Craig Riecke

with

Others

Draft of January 6, 2016

Contents

Chapter 1

Quick Start

In this book, you will use Frenetic to create a full-programmable network. For the moment, let's assume you're familiar with Software Defined Networking and the OpenFlow protocol, and just dive right in. (If you're not, don't worry! We'll introduce some bedrock concepts in the next chapter and explain everything that happened here.)

1.1 Installation

There are several ways to get started with Frenetic, but the easiest is to use Frenetic VM. Frenetic itself only runs on Linux, but the Frenetic VM will run on any host system that supports VirtualBox, including Windows, Mac OS X and practically any version of Linux itself. Keeping Frenetic it in its own VM will keep your own system clean and neat. Later on, if you want to install Frenetic on a bare metal Ubuntu Linux server or network device, you can use the instructions in **??**.

- Install VirtualBox from https://www.virtualbox.org/wiki/Downloads. Use the latest version platform package appropriate for your system.
- Install Vagrant at http://www.vagrantup.com/downloads. Vagrant automates the process of building VM's from scratch, and Frenetic VM uses it to build its own environment. This is more reliable than downloading a multi-gigabyte VM file.
- Install Frenetic VM from https://github.com/frenetic-lang/frenetic-vm. You can simply use the Download Zip button and unzip to an appropriate directory on your system, like frenetic-vm. Then from a terminal or command prompt:

```
$ cd /path/to/frenetic-vm
$ vagrant up
... lots of text
```

The build process may take 15 minutes to an hour, depending on the speed of your system and Internet connection.

1.2 What Do You Get With Frenetic VM?

At the end of the process you will have a working copy of Frenetic with lots of useful open source infrastructure:

Mininet software builds a test network inside of Linux. It can simulate a topology with many switches and hosts. Writing a network application and throwing it into production is . . . well, pretty risky, but running it on Mininet first can be a good test for how it works beforehand. We'll use it throughout this book.

Wireshark captures and analyzes network traffic. It's a great debugging tool, and very necessary for sifting through large amounts of data.

Frenetic . This layer provides an easy-to-use programmable layer on top of ODL. Its main job is to shuttle OpenFlow messages between ODL and your application, and to translate the language NetKAT into OpenFlow flow tables. We'll see the differences between the two as we go.

Hmmm, that's a lot of software - what do *you* bring to the table? You write your network application in Python, using the Frenetic framework. As you'll see, it's quite easy to build a network device from scratch, and easy to grow it organically to fit your requirements. Python is fairly popular, and knowing it will give you a head start into Frenetic programming. But if you're a Python novice that's OK. As long as you know one object-oriented language fairly well, you should be able to follow the concepts. We'll introduce you to useful Python features, like list comprehensions, as we go.

1.3 An Attempt at Hello World

So let's dive right in. We'll set up a Mininet network with one switch and two hosts. First you should work from the directory where you installed Frenetic VM.

```
$ cd /path/to/frenetic-vm
```

Then start up the VM:

```
$ vagrant up
ringing machine 'default' up with 'virtualbox' provider...
==> default: Clearing any previously set forwarded ports...
```

```
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 22 => 2222 (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Setting hostname...
==> default: Mounting shared folders...
    default: /vagrant => /Users/cr396/frenetic-vm
    default: /home/vagrant/src => /Users/cr396/frenetic-vm/src
==> default: Machine already provisioned. Run `vagrant provision` or...
==> default: to force provisioning. Provisioners marked to run always...
```

Then log in to the VM. At this point your command prompt will change to `vagrant@frenetic` to distinguish it from your host machine.

```
 $ vagrant ssh
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Oct  6 10:35:06 2015 from 10.0.2.2
vagrant@frenetic:~$
```

So you are now working inside an Ubuntu-based VM. You don't really need to know Ubuntu, but just know that Mac OS and Windows commands won't necessarily work here.

Let's start up a Mininet network with one switch and two nodes.

```
vagrant@frenetic:~$ sudo mn --topo=single,2 --controller=remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2
```

```
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

The prompt changes to `mininet>` to show your working in Mininet. The error message `Unable to contact controller at 127.0.0.1:6633` looks a little ominous, but not fatal.

You now have an experimental network with two hosts named h1 and h2. To see if there's connectivity between them, use the command `h1 ping h2` which means "On host h1, ping the host h2."

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100\% packet loss, time 5014ms
pipe 3
```

The ping gets executed over and over again, but it's clearly not working. So we press CTRL-C to stop and quit out of Mininet:

```
mininet> quit
```

So by default, hosts can't talk over the network to each other. We're going to fix that by writing a *network application*. Frenetic will act as the controller on the network, and the network application tells Frenetic how to act.

1.4 A Repeater

You write your network application in Python, using the Frenetic framework. Mininet is currently running in our VM under its own terminal window, and we can leave it like that. We'll do our programming in another window, so start up another one and log into our VM:

```
 $ cd /path/to/frenetic-vm
 $ vagrant ssh
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-30-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Oct  6 10:35:06 2015 from 10.0.2.2
vagrant@frenetic:~$
```

Create a tutorial directory for your use:

```
vagrant@frenetic:~$ mkdir tutorial
vagrant@frenetic:~$ cd tutorial
```

Now we'll write our first network application. You can use your favorite Unix text editor - vim and nano are already installed, or you can install your own favorite one with Ubuntu's *apt-get* commands. Nano is a nice editor if your feeling indecisive:

```
vagrant@frenetic:~/tutorial$ nano repeater.py
```

Now type in the following network application:

```
1  import sys
2  sys.path.append('../src/frenetic/lang/python')
3  import frenetic
4  from frenetic.syntax import *
5
6  class RepeaterApp(frenetic.App):
7
8      def connected(self):
9          self.update( id >> SendToController("repeater_app") )
10
11     def packet_in(self, dpid, port_id, payload):
12         out_port = 2 if port_id = 1 else 1
```

```
13          self.pkt_out(dpid, payload, [ Output(Physical(out_port_id)) ] )
14
15  app = RepeaterApp()
16  app.start_event_loop()
```

Lines 1-4 are pretty much the same in every Frenetic network application. Similarly, lines 15-16 are similar in most cases. The meat of the application is an object class named RepeaterApp, whose base class is `frenetic.App`. A frenetic application can hook code into different points of the network event cycle. In our Repeater network app, the only two events we're interested in here are `connected`, which is fired when a switch connects for the first time to Frenetic, and `packet_in`, which is fired every time a packet bound for a controller arrives.

The code in `connected` merely directs the switch to send all packets to our application. The interesting code is in `packet_in` and implements a *repeater*. A repeater is the oldest type of network device, and is sometimes called a *hub*. In a repeater, if a packet enters on port 1, it should get copied out to port 2. Conversely, if a packet enters on port 2, it should get copied out to port 1. If there were more ports in our switch, we'd write a more sophisticated repeater – one that outputs the packet to all ports except the one on which it arrived (called the *ingress port*).

`pkt_out` is a method provided by Frenetic to actually send the packet out the switch. It takes three parameters: a switch, a packet, and a policy. Here are policy send the packet out to port `out_port_id`.

1.5 Running The Repeater Application

So let's get this running in a lab setup. Three programs need to be running: Mininet, Frenetic, and our new Repeater application. For now, we'll run them in three separate command lines, having typed `vagrant ssh` in each to login to the VM.

In the first terminal window, we'll start up Frenetic:

```
vagrant@frenetic:~$ cd src/frenetic
vagrant@frenetic:~src/frenetic$ ./frenetic.native http-controller \
  > --verbosity debug
 [INFO] Calling create!
 [INFO] Current uid: 1000
 [INFO] Successfully launched OpenFlow controller with pid 3062
 [INFO] Connecting to first OpenFlow server socket
 [INFO] Failed to open socket to OpenFlow server: (Unix.Unix_error...
 [INFO] Retrying in 1 second
 [INFO] Successfully connected to first OpenFlow server socket
 [INFO] Connecting to second OpenFlow server socket
 [INFO] Successfully connected to second OpenFlow server socket
```

In the second, we'll start up Mininet with the same configuration as before:

```
vagrant@frenetic:~$ sudo mn --topo=single,2 --controller=remote
*** Creating network
*** Adding controller
```

The following will appear in your Frenetic window to show a connection has been made:

```
 [INFO] switch 1 connected
[DEBUG] Setting up flow table
+----------------------+
| 1 | Pattern | Action |
|----------------------|
|              |       |
+----------------------+
```

And in the third, we'll start our repeater application:

```
vagrant@frenetic:~/tutorial$ python repeater.py
No client_id specified. Using d7650d3aebfc4bd9a708eef1382041ba
Starting the tornado event loop (does not return).
```

The following will appear in the Frenetic window.

```
 [INFO] GET /version
 [INFO] POST /d7650d3aebfc4bd9a708eef1382041ba/update_json
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] New client d7650d3aebfc4bd9a708eef1382041ba
[DEBUG] Installing policy
drop | port := pipe(repeater_app) | port := pipe(repeater_app)
[DEBUG] Setting up flow table
+---------------------------------------+
| 1 | Pattern | Action                  |
|---------------------------------------|
|              | Output(Controller(128)) |
+---------------------------------------+
```

And finally, we'll pop over to the Mininet window and try our connection test once more:

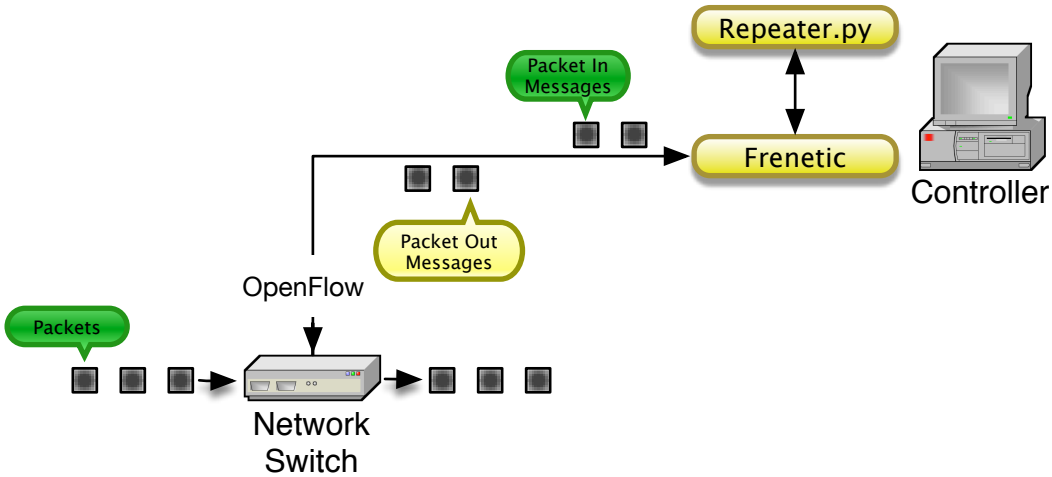
```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=149 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=97.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=88.7 ms
```

Ah, much better! Our pings are getting through. You can see evidence of this in the Frenetic window:

```
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
 [INFO] GET /d7650d3aebfc4bd9a708eef1382041ba/event
 [INFO] POST /pkt_out
[DEBUG] SENDING PKT_OUT
```

1.6 Summary

You now have a working SDN, or Software Defined Network! Like much software, it works in layers:



1. At the bottom is your switches and wires. In our lab setup, Mininet and Open-VSwitch is a substitute for this layer.

2. In the middle is Frenetic. It talks the OpenFlow protocol to the switches (or to Mininet) – this is called the Southbound interface. It also accepts its own language called NetKAT from network applications – this is called the Northbound interface.

3. At the very top is your network application, which you write in Python. It defines how packets are dealt with.

We wrote a very simple network application that emulates a network repeater. It responds to the `packet_in` event coming from the switches through Frenetic when a packet arrives at the switch. And it sends the `pkt_out` message to send the packet back out through Frenetic to the switch. Frenetic-vm makes installing and testing all the pieces straightforward. When you're done, your network application can be deployed to a real production network.

Obviously you can do much more than just simple repeating with SDN! We'll cover that next with some background on OpenFlow and NetKAT, the underlying language of Frenetic.

Chapter 2

Introduction to NetKAT

Software Defined Networking, or SDN, is a huge paradigm shift in the computing world. Traditional networking involves expensive, proprietary "boxes" from major vendors, plugging them in, configuring them, and hoping they meet your needs. But traditional networking suffers from these maladies:

- The devices are flexible only within narrow configuration parameters. Special requirements, like preventing certain kinds of devices from mobility, or configuring the spanning tree to prefer certain paths, are either impossible or expensive.
- While the devices are powered by software, there's no workable way to examine the underlying code or prove it's correct.
- Upgrades tend to be the forklift-variety, since mixing and matching old and new hardware is a dicey proposition . . . not to mention mixing hardware from different vendors.
- Configuration is not easily automated. Solutions are often proprietary, require special programming languages, and are not interchangeable. Because of this, modern data center virtualization is more difficult.
- Adding support for new protocols is slow and expensive.

With SDN, data centers can step off the proprietary network treadmill. Its a shift similar to the personal computer revolution of the 1980's. Before that, IBM and similar mainframes controlled the computer space, and required the same kinds of forklift upgrades networks do. The IBM Personal Computer opened up the architecture to competitors, who could then design and build extensions that made it more useful. This created a snowball effect, with hardware extensions like the mouse and Ethernet cards opening the way for new software like Microsoft Windows and the Netscape browser.

SDN opens network devices in a similar manner, allowing them to be extended and manipulated in interesting, user-defined ways. Although the term SDN has been hijacked to mean many things, it most often refers to OpenFlow-enabled network software and

devices. OpenFlow is an open protocol defined by the Open Network Foundation that manipulates the control plane of a network intermediary.

Frenetic is an OpenFlow controller, meaning it talks the OpenFlow protocol to network intermediaries. In turn, it exposes an API that can be used to write network programs easily. It works with any network intermediary that understands the OpenFlow 1.0 protocol – both hardware devices like the HP 2920 and software "devices" like Open vSwitch. So let's take a brief look at OpenFlow itself.

2.1 Introduction to OpenFlow

Every network device – from the lowliest repeater, to firewalls and load balancers, all the way up to the most complex router – has two conceptual layers:

The data plane performs actions on packets. It manipulates headers, copies packets to outgoing (or egress) ports, or drops packets. It consults control tables - MAC address tables, ARP caches, OSPF tables, etc. - to guide it.

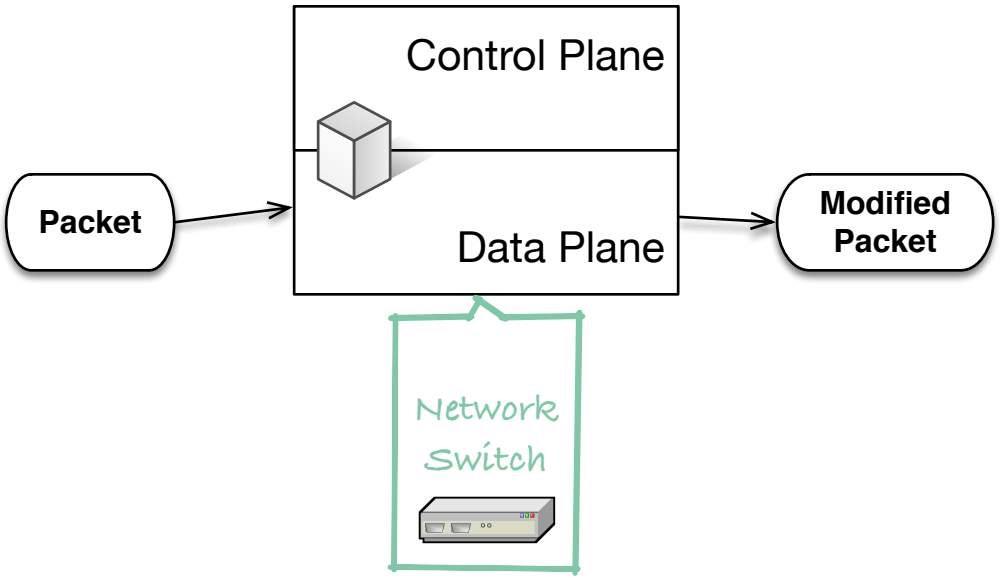
The control plane manipulates the control tables themselves. People, in turn, may manipulate the control plane through configuration software. Or packets may do it: specialized ones like OSPF neighbor exchange, ARP requests, or just examining plain ol' packets themselves. But they never actually touch the packets.

This separation is only conceptual. You'd be hard pressed to open a network device, point to a chip and say, "That's the data plane." It helps in understanding a device, though, because they have different goals:

The data plane 's job is to move data as quickly as possible. It relies more on fast table lookups than complex algorithms.

The control plane works more flexibly, yet conservatively. Control tables should not change often, and when they do, appropriate checks and balances should be applied to ensure the data plane keeps working. For example, the Spanning Tree Protocol (or STP) ensures packets are routed along the shortest path with no loops. Calculating the spanning tree is the control plane's job, as its complex. But once that's calculated, the data plane can use it to forward packets quickly.

A traditional network device looks like this. The control plane is closed and contained fully within the box.



The OpenFlow protocol makes the control plane *programmable*. Rather than relying on the entire program being inside the box, you write a program that advises the control plane and runs outside the box. It's like an advisor that makes aribtrarily complex control table manipulations. The programmable piece runs on any standard computer, and is collectively called the *controller*.

DIAGRAM HERE

The controller can be written in any language and run on any computer . . . the only requirement is it must speak the OpenFlow protocol to the device. You can think of these two pieces working in a tandem through an OpenFlow conversation:

Device: I don't know what to do with this packet. It came in port 3 from Ethernet mac address 10:23:10:59:12:fb:5c.

Controller: OK. I'll memorize it. In the meantime, send it out all ports on the switch except port 3.

Device: I don't know what to do with this packet. It's bound for Ethernet mac address 10:23:10:59:12:fb:5c.

Controller: Oh yeah. I know that's on port 3. Install a rule so all packets going to that mac address go out port 3.

Device: OK!

Controller: How many packets have went out port 3, by the way?

Device: 82,120.

Device: (To itself) I just saw a packet destined for Ethernet mac address 10:23:10:59:12:fb:5c, but I have a rule for dealing with it. I'm gonna send it out port 3.

OpenFlow boils down control plane functionality to a common core, and many of the decisions can be made there. Any complex decisions that can't be handled independently by the control plane can be offloaded to the controller. In a well-designed Software Defined Network, the controller gets involved only when necessary. After all, the conversation between the device and the controller takes time, and anything that can eliminate this conversation makes the packets go faster.

So, central to the OpenFlow model is the *flow table*. Flow tables have *entries*, sometimes called *flow rules*, that are consulted for making decisions. A sample flow table might look like this:

DIAGRAM HERE

What's possible in a flow entry? There's a lot of flexibility here:

A match specifies patterns of packet header and metadata values. OpenFlow 1.3 defines 40 different fields to match: some popular ones are the Input Port, the Ethernet Destination mac address, and the TCP destination port. The match values can either be exact (like 10:23:10:59:12:fb:5c above) or wild carded (like 128.56.0.0/16 for a particular Ip subnet).

An instruction tells what to do if the match occurs. Instructions can apply actions (send a packet out a port, or write some header information, or send a packet to the controller), invoke groups (like a function call in a programming language), or set variables.

A priority defines the order that matches are consulted. When more than one entry matches a particular packet, the entry with the highest priority wins.

A cookie is a primary key for an entry. The cookie value means nothing to the device, but the controller can use it to delete or modify entries later.

In our example above, the controller installed a flow entry matching Ethernet Destination 10:23:10:59:12:fb:5c, and an instruction applying the action "Output it through Port 3".

OpenFlow's flow table model is *abstract*. An OpenFlow device is not necessarily going to find a RAM chip with the matches, instructions and priorities . . . although a pure software switch like Open vSwitch might mirror it quite closely. Instead, the controller asks the device to install entries, and the device accommodates it by placing entries in its own tables. For example, a real network device might have an L3 table that matches subnets with the various ports that have IP gateways. A programmer, knowing this table exists, can write instructions that match on those ports, and place it directly in the table accordingly.

Suppose you wanted to write your own controller from scratch. You could do that just by talking the OpenFlow protocol. Let's say you wrote this program in Node.js and

placed it on the server "controller.example.com", listening on TCP port 6653. Then you'd just point your OpenFlow network device at controller.example.com:6653. Then your program could install flow table entries into the network device over the OpenFlow protocol.

Hmmm. Sounds pretty easy, but ...

2.2 OpenFlow is Difficult

2.2.1 OpenFlow Tables Are Difficult to Program

Writing flow table entries directly is like writing programs in assembly language.

2.2.2 OpenFlow Tables Are Difficult to Compose

2.3 NetKAT Makes Network Programming Easier

2.4 Predicates

A NetKAT *predicate* is a clause used to match packets. The base language is pretty straightforward:

SwitchEq(n)	Matches packets that arrive on switch n, where n is the Datapath ID of the switch.
PortEq(n)	Matches packets that arrive on port n. Generally ports are numbered 1-m, where m is the number of interfaces, but they don't need to be consecutive.
EthSrcEq(mac)	Matches packets whose Ethernet Mac source address is mac, which is a string in the standard form $nn : nn : nn : nn : nn : nn$ where the n's are lowercase hexadecimal digits.
EthDstEq(mac)	Matches packets whose Ethernet Mac destination address is mac.
VlanEq($vlan$)	Matches packets whose VLAN is $vlan$, and integer from 1-4096. Packets without a VLAN are never matched by this predicate.
VlanPcpEq(p)	Matches packets whose VLAN Priority Code Point is p. Packets without a VLAN are never matched by this predicate.
EthTypeEq(t)	Matches packets whose Ethernet Type is t, where t is a 32 bit integer. Popular values of t are 0x800 for IP and 0x806 for ARP.
IPProtoEq(p)	Matches packets whose IP Protocol is p, a number from 0-255. Popular values are 1 for ICMP, 6 for TCP and 17 for UDP. This match only makes sense when EthTypeEq([0x800, 0x806]) (IP or ARP).
IPSrcEq($addr$, $mask$)	Matches packets whose IP source address is $addr$. If $mask$ is provided, a number from 1-32, this matches all hosts on a network whose first $mask$ bits match the host. If it's omitted, the entire address is matched – i.e. only one IP host. This match only makes sense when EthTypeEq([0x800, 0x806]) (IP or ARP).
IPDstEq($addr$, $mask$)	Matches packets whose IP destination address is $addr$. Follows same rules as IpSrcEq.
TCPSrcPortEq(p)	Matches packets whose TCP source port is p, an integer from 0-65535.
TCPDstPortEq(p)	Matches packets whose TCP destination port is p, an integer from 0-65535. Popular values are 80 for HTTP, 22 for SSH, and 443 for SSL.

If you're familiar with OpenFlow, this list should look familiar to you – it's exactly the same list of fields you can use in an OpenFlow flow table. NetKAT, however, augments these matching rules with combination operators:

$p1$ & $p2$	Matches packets that satisfy $p1$ AND $p2$
And($p1$, $p2$, ..., pn)	Matches packets that satisfy all the predicates: $p1$ AND $p2$ AND ... AND pn
$p1$ TODO $p2$	Matches packets that satisfy $p1$ OR $p2$
Or($p1$, $p2$, ..., pn)	Matches packets that satisfy one of the predicates: $p1$ OR $p2$ OR ... OR pn

2.5 Policies

2.6 Events and Hooks

Chapter 3

Learning Switch

3.1 Design

3.2 Bootup, Switch Information

3.3 Learning Ports

3.4 Timeouts and Moves

Chapter 4

Proxy ARP

4.1 The ARP Protocol

4.2 Design

4.3 Snooping on ARP Requests and Replies

4.4 Replying

4.5 Maintaining State Across Restarts

Chapter 5

Handling Vlans

5.1 Vlan Uses

5.2 Design

5.3 Maintaining Separate Vlan Tables

5.4 Tagging and Untagging

Chapter 6

Gathering Statistics

Chapter 7

SDN Development Tools

7.1 TCPDump

7.2 Wireshark

7.3 Tmux

7.4 Mininet Network Modelling

Chapter 8

Network Address Translation

8.1 Why Do We Need NAT?

8.2 Design

Chapter 9

Spanning Tree Alternatives

9.1 What's Wrong with STP Protocols?

9.2 Design

9.3 Calculating Shortest Paths

9.4 Core/Edge Separation

Chapter 10

Routing

10.1 Design

10.2 Configuring Route Tables

10.3 Using Link State

Chapter 11

Modularization

11.1 Sharing Actions in Rules and Packet Outs

11.2 Subclassing

11.3 Multiple Network Apps

11.4 Clustering

Chapter 12

Frenetic REST API

12.1 REST URL's

12.2 Incoming, Northbound Events

12.3 Messages

Chapter 13

Frenetic/NetKAT Reference

13.1 Predicates

13.2 Policies

13.3 Event Hooks

13.4 Compiler Directives

13.5 Frenetic Command Line

Chapter 14

Productionalizing

14.1 Installing Frenetic on Bare Metal Linux

14.2 Control Scripts

14.3 Logging