Chapter 1

# Routing Policy

## JNCIS EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ Describe JUNOS software routing policy design considerations—import; export; terms; match criteria; actions; default actions
- ✓ Identify the operation of community regular expressions
- ✓ Identify the operation of AS Path regular expressions
- ✓ Evaluate the outcome of a policy using a subroutine
- ✓ Evaluate the outcome of a policy using a policy expression

Before reading this chapter, you should be very familiar with the functionality of a routing policy in the JUNOS software and when it might be appropriate to use one. You should also understand how a multiterm policy uses match criteria and actions to perform its functions. Finally, the use of route filters and their associated match types is assumed knowledge.

In this chapter, we'll explore the use of routing policies within the JUNOS software. We first examine the multiple methods of altering the processing of a policy, including policy chains, subroutines, and expressions. We then discuss the use of a routing policy to locate routes using Border Gateway Protocol (BGP) community values and Autonomous System (AS) Path information.

Throughout the chapter, we see examples of constructing and applying routing policies. We also explore some methods for verifying the effectiveness of your policies before implementing them on the router using the `test policy` command.

Routing policy basics are covered extensively in *JNCIA: Juniper Networks Certified Internet Associate Study Guide* (Sybex, 2003).

# Routing Policy Processing

One of the advantages (or disadvantages depending on your viewpoint) of the JUNOS software policy language is its great flexibility. Generally speaking, you often have four to five methods for accomplishing the same task. A single policy with multiple terms is one common method for constructing an advanced policy. In addition, the JUNOS software allows you to use a policy chain, a subroutine, a prefix list, and a policy expression to complete the same task. Each of these methods is unique in its approach and attacks the problem from a different angle. Let's examine each of these in some more detail.

## Policy Chains

We first explored the concept of a *policy chain* in the *JNCIA Study Guide*. Although it sounds very formal, a policy chain is simply the application of multiple policies within a specific section of the configuration. An example of a policy chain can be seen on the Merlot router as:

```
[edit protocols bgp]
user@Merlot# show
```

```
group Internal-Peers {
    type internal;
    local-address 192.168.1.1;
    export [ adv-statics adv-large-aggregates adv-small-aggregates ];
    neighbor 192.168.2.2;
    neighbor 192.168.3.3;
}
```

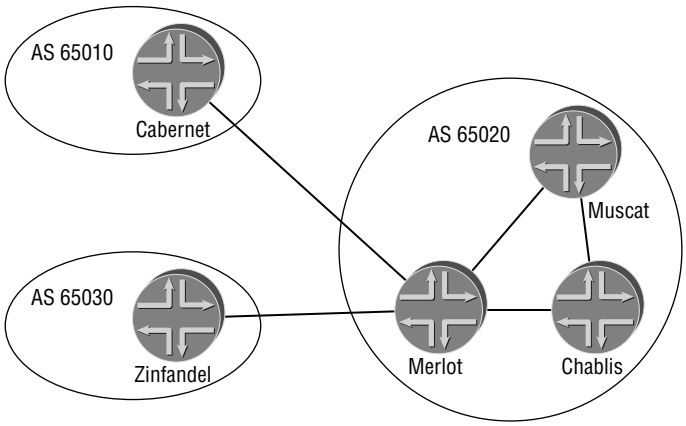
The ***adv-statics***, ***adv-large-aggregates***, and ***adv-small-aggregates*** policies, in addition to the default BGP policy, make up the policy chain applied to the BGP peers of Merlot. When we look at the currently applied policies, we find them to be rather simple:

```
[edit policy-options]
user@Merlot# show
policy-statement adv-statics {
    term statics {
        from protocol static;
        then accept;
    }
}
policy-statement adv-large-aggregates {
    term between-16-and-18 {
        from {
            protocol aggregate;
            route-filter 192.168.0.0/16 upto /18;
        }
        then accept;
    }
}
policy-statement adv-small-aggregates {
    term between-19-and-24 {
        from {
            protocol aggregate;
            route-filter 192.168.0.0/16 prefix-length-range /19-/24;
        }
        then accept;
    }
}
```

You could easily make an argument for just converting this policy chain into a single multi-term policy for the internal BGP (IBGP) peers. While this is certainly true, one of the advantages of a policy chain would be lost: the ability to reuse policies for different purposes.

Figure 1.1 displays the Merlot router with its IBGP peers of Muscat and Chablis. There are also external BGP (EBGP) connections to the Cabernet router in AS 65010 and the Zinfandel router in AS 65030. The current administrative policy within AS 65020 is to send the customer static routes only to other IBGP peers. Any EBGP peer providing transit service should only receive aggregate routes whose mask length is smaller than 18 bits. Any EBGP peer providing peering services should receive all customer routes and all aggregates whose mask length is larger than 19 bits. Each individual portion of these administrative policies is coded into a separate routing policy within the `[edit policy-opitons]` configuration hierarchy. They then provide the administrators of AS 65020 with a multitude of configuration options for advertising routes to its peers.

**FIGURE 1.1** Policy chain network map





Cabernet is providing transit service to AS 65020, which allows it to advertise their assigned routing space to the Internet at large. On the other hand, the peering service provided by Zinfandel allows AS 65020 to route traffic directly between the Autonomous Systems for all customer routes.

The EBGP peering sessions to Cabernet and Zinfandel are first configured and established:

```
[edit]
user@Merlot# show protocols bgp
group Internal-Peers {
    type internal;
    local-address 192.168.1.1;
    export [ adv-statics adv-large-aggregates adv-small-aggregates ];
    neighbor 192.168.2.2;
    neighbor 192.168.3.3;
```

```
}
group Ext-AS65010 {
    type external;
    peer-as 65010;
    neighbor 10.100.10.2;
}
group Ext-AS65030 {
    type external;
    peer-as 65030;
    neighbor 10.100.30.2;
}

[edit]
user@Merlot# run show bgp summary
Groups: 3 Peers: 4 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                12         10          0          0          0          0
Peer               AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State
192.168.2.2     65020        170        172       0       0     1:22:50 5/6/0
192.168.3.3     65020        167        170       0       0     1:21:39 5/6/0
10.100.10.2     65010         30         32       0       0       12:57 0/0/0
10.100.30.2     65030         55         57       0       0       24:49 0/0/0
```

The ***adv-large-aggregates*** policy is applied to Cabernet to advertise the aggregate routes with a subnet mask length between 16 and 18 bits. After committing the configuration, we check the routes being sent to AS 65010:

```
[edit protovols bgp]
user@Merlot# set group Ext-AS65010 export adv-large-aggregates

[edit protovols bgp]
user@Merlot# commit

[edit protocols bgp]
user@Merlot# run show route advertising-protocol bgp 10.100.10.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.0.0/16          Self                                    I
192.168.2.0/24          Self                                    I
192.168.2.16/28         Self                                    I
192.168.2.32/28         Self                                    I
```

```
192.168.2.48/28          Self                                I
192.168.2.64/28          Self                                I
192.168.3.0/24           Self                                I
192.168.3.16/28          Self                                I
192.168.3.32/28          Self                                I
192.168.3.48/28          Self                                I
192.168.3.64/28          Self                                I
```

The 192.168.0.0 /16 aggregate route is being sent as per the administrative policy, but a number of other routes with larger subnet masks are also being sent to Cabernet. Let's first verify that we have the correct policy applied:

```
[edit protocols bgp]
user@Merlot# show group Ext-AS65010
type external;
export adv-large-aggregates;
peer-as 65010;
neighbor 10.100.10.2;
```

The ***adv-large-aggregates*** policy is correctly applied. Let's see if we can find where the other routes are coming from. The `show route` command provides a vital clue:

```
[edit]
user@Merlot# run show route 192.168.3.16/28

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.3.16/28    *[BGP/170] 05:51:24, MED 0, localpref 100, from 192.168.3.3
                      AS path: I
                    > via so-0/1/1.0
```

Merlot has learned this route via its BGP session with Chablis. Since it is an active BGP route, it is automatically advertised by the BGP default policy. Remember that the default policy is always applied to the end of every policy chain in the JUNOS software. What we need is a policy to block the more specific routes from being advertised. We create a policy called ***not-larger-than-18*** that rejects all routes within the 192.168.0.0 /16 address space that have a subnet mask length greater than or equal to 19 bits. This ensures that all aggregates with a mask between 16 and 18 bits are advertised—exactly the goal of our administrative policy.

```
[edit policy-options]
user@Merlot# show policy-statement not-larger-than-18
term reject-greater-than-18-bits {
```

```
    from {
        route-filter 192.168.0.0/16 prefix-length-range /19-/32;
    }
    then reject;
}

[edit policy-options]
user@Merlot# top edit protocols bgp

[edit protocols bgp]
user@Merlot# set group Ext-AS65010 export not-larger-than-18

[edit protocols bgp]
user@Merlot# show group Ext-AS65010
type external;
export [ adv-large-aggregates not-larger-than-18 ];
peer-as 65010;
neighbor 10.100.10.2;

[edit protocols bgp]
user@Merlot# commit
commit complete

[edit protocols bgp]
user@Merlot# run show route advertising-protocol bgp 10.100.10.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.0.0/16          Self                                    I
```

It appears as if our policy chain is working correctly—only the 192.168.0.0 /16 route is advertised to Cabernet. In fact, as long as the ***not-larger-than-18*** policy appears before the BGP default policy in our policy chain we achieve the desired results.

We now shift our focus to Zinfandel, our EBGP peer in AS 65030. Our administrative policy states that this peer should receive only aggregate routes larger than 18 bits in length and all customer routes. In anticipation of encountering a similar problem, we create a policy called ***not-smaller-than-18*** that rejects all aggregates with mask lengths between 16 and 18 bits. In addition, we apply the ***adv-statics*** and ***adv-small-aggregates*** policies to announce those particular routes to the peer:

```
[edit policy-options]
user@Merlot# show policy-statement not-smaller-than-18
```

```
term reject-less-than-18-bits {
    from {
        protocol aggregate;
        route-filter 192.168.0.0/16 upto /18;
    }
    then reject;
}

[edit policy-options]
user@Merlot# top edit protocols bgp

[edit protocols bgp]
user@Merlot# set group Ext-AS65030 export adv-small-aggregates
user@Merlot# set group Ext-AS65030 export adv-statics
user@Merlot# set group Ext-AS65030 export not-smaller-than-18

[edit protocols bgp]
user@Merlot# show group Ext-AS65030
type external;
export [ adv-small-aggregates adv-statics not-smaller-than-18 ];
peer-as 65030;
neighbor 10.100.30.2;

[edit protocols bgp]
user@Merlot# commit
commit complete

[edit protocols bgp]
user@Merlot# run show route advertising-protocol bgp 10.100.30.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.1.0/24          Self                                    I
192.168.1.16/28         Self                 0                  I
192.168.1.32/28         Self                 0                  I
192.168.1.48/28         Self                 0                  I
192.168.1.64/28         Self                 0                  I
192.168.2.0/24          Self                                    I
192.168.2.16/28         Self                                    I
192.168.2.32/28         Self                                    I
```

```
192.168.2.48/28         Self                            I
192.168.2.64/28         Self                            I
192.168.3.0/24          Self                            I
192.168.3.16/28         Self                            I
192.168.3.32/28         Self                            I
192.168.3.48/28         Self                            I
192.168.3.64/28         Self                            I
192.168.20.0/24         Self                0           I
```

It looks like this policy chain is working as designed as well. In fact, after configuring our individual policies, we can use them in any combination on the router. Another useful tool for reusing portions of your configuration is a policy subroutine, so let's investigate that concept next.

## Policy Subroutines

The JUNOS software policy language is similar to a programming language. This similarity also includes the concept of nesting your policies into a *policy subroutine*. A subroutine in a software program is a section of code that you reference on a regular basis. A policy subroutine works in the same fashion—you reference an existing policy as a match criterion in another policy. The router first evaluates the subroutine and then finishes its processing of the main policy. Of course, there are some details that greatly affect the outcome of this evaluation.

First, the evaluation of the subroutine simply returns a true or false Boolean result to the main policy. Because you are referencing the subroutine as a match criterion, a true result means that the main policy has a match and can perform any configured actions. A false result from the subroutine, however, means that the main policy does not have a match. Let's configure a policy called ***main-policy*** that uses a subroutine:

```
 [edit policy-options policy-statement main-policy]
user@Merlot# show
term subroutine-as-a-match {
    from policy subroutine-policy;
    then accept;
}
term nothing-else {
    then reject;
}
```

Of course, we can't commit our configuration since we reference a policy we haven't yet created. We create the ***subroutine-policy*** and check our work:

```
[edit policy-options policy-statement main-policy]
user@Merlot# commit
Policy error: Policy subroutine-policy referenced but not defined
```

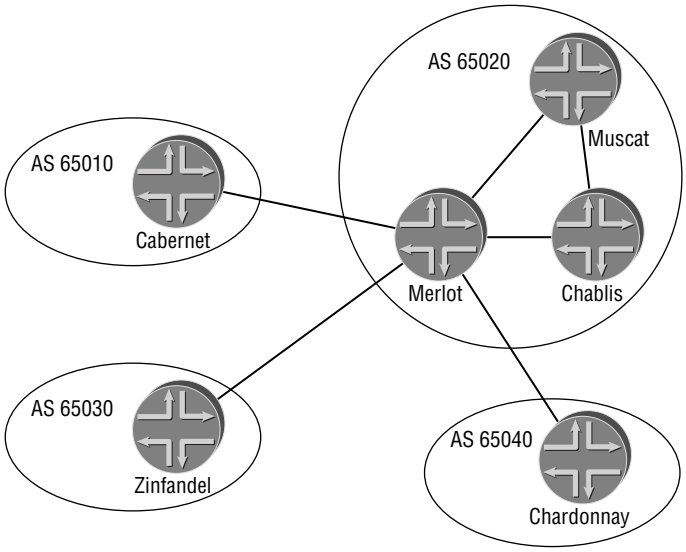
```
error: configuration check-out failed

[edit policy-options policy-statement main-policy]
user@Merlot# up

[edit policy-options]
user@Merlot# edit policy-statement subroutine-policy

[edit policy-options policy-statement subroutine-policy]
user@Merlot# set term get-routes from protocol static
user@Merlot# set term get-routes then accept

[edit policy-options policy-statement subroutine-policy]
user@Merlot# show
term get-routes {
    from protocol static;
    then accept;
}

[edit policy-options policy-statement subroutine-policy]
user@Merlot# commit
commit complete
```

The router evaluates the logic of ***main-policy*** in a defined manner. The match criterion of `from policy subroutine-policy` allows the router to locate the subroutine. All terms of the subroutine are evaluated, in order, following the normal policy processing rules. In our example, all static routes in the routing table match the subroutine with an action of `accept`. This returns a true result to the original, or calling, policy which informs the router that a positive match has occurred. The actions in the calling policy are executed and the route is accepted. All other routes in the routing table do not match the subroutine and should logically return a false result to the calling policy. The router should evaluate the second term of ***main-policy*** and reject the routes.


Keep in mind that the actions in the subroutine do not actually accept or reject a specific route. They are only translated into a true or a false result. Actions that modify a route's attribute, however, are applied to the route regardless of the outcome of the subroutine.

Figure 1.2 shows AS 65020 now connected to the Chardonnay router in AS 65040. The policy subroutine of ***main-policy*** is applied as an export policy to Chardonnay. After establishing the BGP session, we verify that Merlot has static routes to send:

**FIGURE 1.2** Policy subroutine network map




```
[edit]
user@Merlot# show protocols bgp group Ext-AS65040
type external;
peer-as 65040;
neighbor 10.100.40.2;

[edit]
user@Merlot# run show bgp summary
Groups: 4 Peers: 5 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                12         10          0          0          0          0
Peer                 AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State
192.168.2.2       65020       2284       2285       0       0    19:00:15 5/6/0
192.168.3.3       65020       2275       2275       0       0    18:55:29 5/6/0
10.100.10.2       65010       2292       2294       0       0    19:03:50 0/0/0
10.100.30.2       65030       2293       2295       0       0    19:03:46 0/0/0
10.100.40.2       65040         23         25       0       0        9:01 0/0/0

[edit]
user@Merlot# run show route protocol static terse
```

```
inet.0: 33 destinations, 37 routes (33 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 192.168.1.16/28    S   5          0             Discard
* 192.168.1.32/28    S   5          0             Discard
* 192.168.1.48/28    S   5          0             Discard
* 192.168.1.64/28    S   5          0             Discard
```

After applying the policy subroutine to Chardonnay, we check to see if only four routes are sent to the EBGP peer:

```
[edit protocols bgp]
user@Merlot# set group Ext-AS65040 export main-policy

[edit]
user@Merlot# run show route advertising-protocol bgp 10.100.40.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.1.16/28         Self                 0                  I
192.168.1.32/28         Self                 0                  I
192.168.1.48/28         Self                 0                  I
192.168.1.64/28         Self                 0                  I
192.168.2.0/24          Self                                    I
192.168.2.16/28         Self                                    I
192.168.2.32/28         Self                                    I
192.168.2.48/28         Self                                    I
192.168.2.64/28         Self                                    I
192.168.3.0/24          Self                                    I
192.168.3.16/28         Self                                    I
192.168.3.32/28         Self                                    I
192.168.3.48/28         Self                                    I
192.168.3.64/28         Self                                    I
```

The four local static routes are being sent to Chardonnay, but additional routes are being advertised as well. Let's see if we can figure out where these routes are coming from:

```
[edit]
user@Merlot# run show route 192.168.2.16/28

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
192.168.2.16/28    *[BGP/170] 19:06:01, MED 0, localpref 100, from 192.168.2.2
                      AS path: I
                    > via so-0/1/0.0
```

The 192.168.2.16 /28 route is in the routing table as an IBGP-learned route from the Muscat router. We saw a similar problem in the "Policy Chains" section earlier in the chapter when the BGP default policy was advertising "extra" routes. The default policy is affecting the outcome in this case as well, but not in the way that you might think.

The currently applied policy chain for Chardonnay is ***main-policy*** followed by the BGP default policy. The terms of ***main-policy*** account for all routes with an explicit `accept` or `reject` action, so the BGP default policy is not evaluated as a part of the policy chain. It is being evaluated, however, as a part of the subroutine, which brings up the second important concept concerning a policy subroutine. The default policy of the protocol where the subroutine is applied is always evaluated as a part of the subroutine itself. In our case, the BGP default policy is evaluated along with ***subroutine-policy*** to determine a true or false result.

The actions of the default policy within the subroutine mean that you are actually evaluating a policy chain at all times. When you combine the BGP default policy with the terms of ***subroutine-policy***, we end up with a subroutine that looks like the following:

```
policy-options {
    policy-statement subroutine-policy {
        term get-routes {
            from protocol static;
            then accept;
        }
        term BGP-default-policy-part-1 {
            from protocol bgp;
            then accept;
        }
        term BGP-default-policy-part-2 {
            then reject;
        }
    }
}
```

Using this new concept of a subroutine alters the logic evaluation of the subroutine. All static and BGP routes in the routing table return a true result to the calling policy while all other routes return a false result to the calling policy. This clearly explains the routes currently being advertised to Chardonnay. To achieve the result we desire, we need to eliminate the BGP default policy from being evaluated within the subroutine. This is easily accomplished by adding a new term to ***subroutine-policy*** as follows:

```
[edit policy-options policy-statement subroutine-policy]
user@Merlot# show
```

```
term get-routes {
    from protocol static;
    then accept;
}
term nothing-else {
    then reject;
}
```

When we check the results of this new subroutine, we see that only the local static routes are advertised to Chardonnay:

```
[edit]
user@Merlot# run show route advertising-protocol bgp 10.100.40.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.1.16/28         Self                 0                  I
192.168.1.32/28         Self                 0                  I
192.168.1.48/28         Self                 0                  I
192.168.1.64/28         Self                 0                  I
```

### Determining the Logic Result of a Subroutine

It is worth noting again that the configured actions within a subroutine do not in any way affect whether a particular route is advertised by the router. The subroutine actions are used only to determine the true or false result. To illustrate this point, assume that ***main-policy*** is applied as we saw in the "Policy Subroutines" section. In this instance, however, the policies are altered as so:

```
[edit policy-options]
user@Merlot# show policy-statement main-policy
term subroutine-as-a-match {
    from policy subroutine-policy;
    then accept;
}

[edit policy-options]
user@Merlot# show policy-statement subroutine-policy
term get-routes {
    from protocol static;
    then accept;
}
```

```
term no-BGP-routes {
    from protocol bgp;
    then reject;
}
```

We are now aware of the protocol default policy being evaluated within the subroutine, so ***subroutine-policy*** now has an explicit term rejecting all BGP routes. Because they are rejected within the subroutine, there is no need within ***main-policy*** for an explicit `then reject` term. You may already see the flaw in this configuration, but let's follow the logic.

The router evaluates the first term of ***main-policy*** and finds a match criterion of `from policy subroutine-policy`. It then evaluates the first term of the subroutine and finds that all static routes have an action of `then accept`. This returns a true result to ***main-policy***, where the ***subroutine-as-a-match*** term has a configured action of `then accept`. The static routes are now truly accepted and are advertised to the EBGP peer.

When it comes to the BGP routes in the routing table, things occur a bit differently. When the router enters the subroutine, it finds the ***no-BGP-routes*** term where all BGP routes are rejected. This returns a false result to ***main-policy***, which means that the criterion in the ***subroutine-as-a-match*** term doesn't match. This causes the routes to move to the next configured term in ***main-policy***, which has no other terms. The router then evaluates the next policy in the policy chain—the BGP default policy. The default policy, of course, accepts all BGP routes, and they are advertised to the EBGP peer. We can prove this logic with a `show route` command on Merlot:

```
user@Merlot> show route advertising-protocol bgp 10.100.40.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop          MED     Lclpref    AS path
192.168.1.16/28         Self             0                  I
192.168.1.32/28         Self             0                  I
192.168.1.48/28         Self             0                  I
192.168.1.64/28         Self             0                  I
192.168.2.0/24          Self                                I
192.168.2.16/28         Self                                I
192.168.2.32/28         Self                                I
192.168.2.48/28         Self                                I
192.168.2.64/28         Self                                I
192.168.3.0/24          Self                                I
192.168.3.16/28         Self                                I
192.168.3.32/28         Self                                I
192.168.3.48/28         Self                                I
192.168.3.64/28         Self                                I
```

## Prefix Lists

The use of the policy subroutine in the previous section was one method of advertising a set of routes by configuring a single section of code. The JUNOS software provides other methods of accomplishing the same task, and a *prefix list* is one of them. A prefix list is a listing of IP prefixes that represent a set of routes that are used as match criteria in an applied policy. Such a list might be useful for representing a list of customer routes in your AS.

A prefix list is given a name and is configured within the `[edit policy-options]` configuration hierarchy. Using Figure 1.2 as a guide, each router in AS 65020 has customer routes that fall into the 24-bit subnet defined by their loopback address. This means that Merlot, whose loopback address is 192.168.1.1 /32, assigns customer routes within the 192.168.1.0 /24 subnet. The Muscat and Chablis routers assign customer routes within the 192.168.2.0 /24 and 192.168.3.0 /24 subnets, respectively.

Merlot has been designated the central point in AS 65020 to maintain a complete list of customer routes. It configures a prefix list called ***all-customers*** as so:

```
[edit]
user@Merlot# show policy-options prefix-list all-customers
192.168.1.16/28;
192.168.1.32/28;
192.168.1.48/28;
192.168.1.64/28;
192.168.2.16/28;
192.168.2.32/28;
192.168.2.48/28;
192.168.2.64/28;
192.168.3.16/28;
192.168.3.32/28;
192.168.3.48/28;
192.168.3.64/28;
```

As you look closely at the prefix list you see that there are no match types configured with each of the routes (as you might see with a route filter). This is an important point when using a prefix list in a policy. The JUNOS software evaluates each address in the prefix list as an exact route filter match. In other words, each route in the list must appear in the routing table exactly as it is configured in the prefix list. You reference the prefix list as a match criterion within a policy like this:

```
[edit]
user@Merlot# show policy-options policy-statement customer-routes
term get-routes {
```

```
    from {
        prefix-list all-customers;
    }
    then accept;
}
term nothing-else {
    then reject;
}
```

All the routes in the ***all-customers*** prefix list appear in the current routing table:

```
[edit]
user@Merlot# run show route 192.168/16 terse

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 192.168.0.0/16     A 130                        Reject
                     B 170        100            >so-0/1/0.0     I
                     B 170        100            >so-0/1/1.0     I
* 192.168.1.0/24     A 130                        Reject
* 192.168.1.1/32     D   0                       >lo0.0
* 192.168.1.16/28    S   5          0             Discard
* 192.168.1.32/28    S   5          0             Discard
* 192.168.1.48/28    S   5          0             Discard
* 192.168.1.64/28    S   5          0             Discard
* 192.168.2.0/24     B 170        100            >so-0/1/0.0     I
* 192.168.2.2/32     O  10          1            >so-0/1/0.0
* 192.168.2.16/28    B 170        100          0 >so-0/1/0.0     I
* 192.168.2.32/28    B 170        100          0 >so-0/1/0.0     I
* 192.168.2.48/28    B 170        100          0 >so-0/1/0.0     I
* 192.168.2.64/28    B 170        100          0 >so-0/1/0.0     I
* 192.168.3.0/24     B 170        100            >so-0/1/1.0     I
* 192.168.3.3/32     O  10          1            >so-0/1/1.0
* 192.168.3.16/28    B 170        100          0 >so-0/1/1.0     I
* 192.168.3.32/28    B 170        100          0 >so-0/1/1.0     I
* 192.168.3.48/28    B 170        100          0 >so-0/1/1.0     I
* 192.168.3.64/28    B 170        100          0 >so-0/1/1.0     I
```

After applying the ***customer-routes*** policy to the EBGP peer of Zinfandel, as seen in Figure 1.2, we see that only the customer routes are advertised:

```
[edit protocols bgp]
user@Merlot# show group Ext-AS65030
type external;
export customer-routes;
peer-as 65030;
neighbor 10.100.30.2;

[edit protocols bgp]
user@Merlot# run show route advertising-protocol bgp 10.100.30.2

inet.0: 32 destinations, 36 routes (32 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.1.16/28         Self                 0                  I
192.168.1.32/28         Self                 0                  I
192.168.1.48/28         Self                 0                  I
192.168.1.64/28         Self                 0                  I
192.168.2.16/28         Self                                    I
192.168.2.32/28         Self                                    I
192.168.2.48/28         Self                                    I
192.168.2.64/28         Self                                    I
192.168.3.16/28         Self                                    I
192.168.3.32/28         Self                                    I
192.168.3.48/28         Self                                    I
192.168.3.64/28         Self                                    I
```

## Policy Expressions

In the "Policy Subroutines" section earlier in the chapter, we compared the JUNOS software policy language to a programming language. This comparison also holds true when we discuss a *policy expression*. A policy expression within the JUNOS software is the combination of individual policies together with a set of logical operators. This expression is applied as a portion of the policy chain. To fully explain how the router uses a policy expression, we need to discuss the logical operators themselves as well as the evaluation logic when each operator is used. Then, we look at some examples of policy expressions in a sample network environment.

## Logical Operators

You can use four logical operators in conjunction with a policy expression. In order of precedence, they are a logical NOT, a logical AND, a logical OR, and a group operator. You can think of the precedence order as being similar to arithmetic, where multiplication is performed before addition. In the case of the logical operators, a NOT is performed before an OR. Let's look at the function of each logical operator, as well as an example syntax:

**Logical NOT** The *logical NOT* (`!`) reverses the normal logic evaluation of a policy. A true result becomes a false and a false result becomes a true. This is encoded in the JUNOS software as `!policy-name`.

**Logical AND** The *logical AND* (`&&`) operates on two routing policies. Should the result of the first policy be a true result, then the next policy is evaluated. However, if the result of the first policy is a false result, then the second policy is skipped. This appears as `policy-1 && policy-2`.

**Logical OR** The *logical OR* (`||`) also operates on two routing policies. It skips the second policy when the first policy returns a true result. A false result from the first policy results in the second policy being evaluated. This appears as `policy-1 || policy-2`.

**Group operator** The *group operator*, represented by a set of parentheses, is used to override the default precedence order of the other logical operators. For example, a group operator is useful when you want to logically OR two policies and then AND the result with a third policy. The JUNOS software views this as `(policy-1 || policy-2) && policy-3`.


When parentheses are not used to group policy names, such as `policy-1 || policy-2 && policy-3`, the JUNOS software evaluates the expression using the default precedence order. This order requires all logical NOT operations to be performed first, then all logical AND operations, and finally all logical OR operations. For clarity, we recommend using group operators when more than two policies are included in an expression.

## Logical Evaluation

When the router encounters a policy expression, it must perform two separate steps. The logical evaluation is calculated first, followed by some actual action on the route. In this respect, the policy expression logic is similar to a policy subroutine. The two are very different, however, when it comes to using the protocol default policy. Because the policy expression occupies a single place in the normal policy chain, the protocol default policy is not evaluated within the expression. It is evaluated only as a part of the normal policy chain applied to the protocol.

When the router evaluates the individual policies of an expression, it determines whether the policy returns a true or false result. A true result is found when either the `accept` or `next policy` action is found. The `next policy` action is either encountered by its explicit configuration within the policy or when the route does not match any terms in the policy. A logical false result is encountered when the `reject` action is encountered within the policy.

After determining the logical result of the expression, the router performs some action on the route. This action results from the policy that guaranteed the logical result. This might sound a bit confusing, so let's look at some examples to solidify the concept.

### OR Operations

The normal rules of OR logic means that when either of the policies returns a true value, then the entire expression is true. When configured as `policy-1 || policy-2`, the router first evaluates `policy-1`. If the result of this policy is a true value, then the entire expression becomes true as per the OR evaluation rules. In this case, `policy-2` is not evaluated by the router. The route being evaluated through the expression has the action defined in `policy-1` applied to it since `policy-1` guaranteed the result of the entire expression.

Should the evaluation of `policy-1` return a false result, then `policy-2` is evaluated. If the result of `policy-2` is true, the entire expression is true. Should the evaluation of `policy-2` result in a false, the entire expression becomes false as well. In either case, `policy-2` has guaranteed the result of the entire expression. Therefore, the action in `policy-2` is applied to the route being evaluated through the expression.

### AND Operations

The rules of AND logic states that both of the policies must return a true value to make the entire expression true. If either of the policies returns a false value, then the entire expression becomes false. The configuration of `policy-1 && policy-2` results in the router first evaluating `policy-1`. If the result of this policy is true, then `policy-2` is evaluated since the entire expression is not yet guaranteed. Only when the result of `policy-2` is true does the expression become true. Should the evaluation of `policy-2` return a false, the entire expression then becomes false. Regardless, `policy-2` guarantees the result of the entire expression and the action in `policy-2` is applied to the route being evaluated.

Should the evaluation of `policy-1` return a false result, then the expression is guaranteed to have a false result since both policies are not true. In this case, the action in `policy-1` is applied to the route.

### NOT Operations

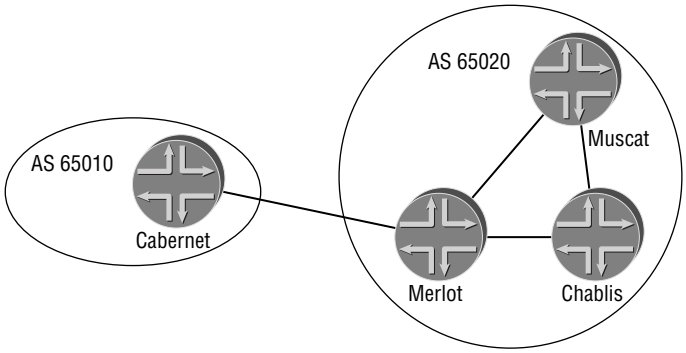
The operation of a logical NOT is performed only on a single policy. When the result of a NOT evaluation is true, the router transforms that into a false evaluation. This false result tells the router to reject the route being evaluated. The exact opposite occurs when the NOT evaluation is false. The router transforms the false into a true result and accepts the route being evaluated.

## An Example of Expressions

A policy expression in the JUNOS software occupies a single position in a protocol's policy chain, so the protocol in use is an important factor in determining the outcome of the expression. We'll use BGP as our protocol using the information in Figure 1.3.

The Merlot router in AS 65020 is peering both with its internal peers of Muscat and Chablis and with the Cabernet router in AS 65010. The customer routes within the subnets of 192.168.2.0 /24 and 192.168.3.0 /24 are being advertised from Muscat and Chablis, respectively. Two policies are configured on Merlot to locate these routes:

**FIGURE 1.3** Policy expression network map




```
[edit policy-options]
user@Merlot# show policy-statement Muscat-routes
term find-routes {
    from {
        route-filter 192.168.2.0/24 longer;
    }
    then accept;
}
term nothing-else {
    then reject;
}

[edit policy-options]
user@Merlot# show policy-statement Chablis-routes
term find-routes {
    from {
        route-filter 192.168.3.0/24 longer;
    }
    then accept;
}
```

By default, the BGP policy advertises the customer routes to Cabernet:

```
[edit]
user@Merlot# run show route advertising-protocol bgp 10.100.10.2

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.2.16/28         Self                                    I
```

```
192.168.2.32/28          Self                                I
192.168.2.48/28          Self                                I
192.168.2.64/28          Self                                I
192.168.3.16/28          Self                                I
192.168.3.32/28          Self                                I
192.168.3.48/28          Self                                I
192.168.3.64/28          Self                                I
```

### An OR Example

A logical OR policy expression is configured on the Merlot router. This means that the policy chain applied to Cabernet becomes the expression followed by the default BGP policy:

```
[edit protocols bgp]
lab@Merlot# show group Ext-AS65010
type external;
export ( Muscat-routes || Chablis-routes );
peer-as 65010;
neighbor 10.100.10.2;
```

To illustrate the operation of the expression, we select a route from each neighbor. Merlot evaluates the 192.168.2.16 /28 route against the ***Muscat-routes*** policy first. The route matches the criteria in the ***find-routes*** term, where the action is `accept`. This means that the first policy is a true result and the entire logical OR expression is also true. The configured action of `accept` in the ***Muscat-routes*** policy is applied to the route and it is sent to Cabernet. We can verify this with the `show route` command:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.2.16/28

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                   Nexthop              MED     Lclpref    AS path
192.168.2.16/28          Self                                    I
```

The 192.168.3.16 /28 route is selected from the Chablis router. As before, Merlot evaluates the ***Muscat-routes*** policy first. This route matches the ***nothing-else*** term and returns a false result to the expression. Because the expression result is not guaranteed yet, Merlot evaluates the ***Chablis-routes*** policy. The route matches the ***find-routes*** term in that policy and returns a true result to the expression. The ***Chablis-routes*** policy guaranteed the expression result, so the action of `accept` from that policy is applied to the route. Again, we verify that the route is sent to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.3.16/28

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                   Nexthop              MED     Lclpref    AS path
192.168.3.16/28          Self                                    I
```

### An AND Example

Using the same sample routes and policies, we can explore a logical AND policy expression on the Merlot router. Again, the expression occupies a single slot in the policy chain:

```
[edit protocols bgp]
lab@Merlot# show group Ext-AS65010
type external;
export ( Muscat-routes && Chablis-routes );
peer-as 65010;
neighbor 10.100.10.2;
```

Merlot first evaluates the 192.168.2.16 /28 route against the ***Muscat-routes*** policy. The route matches the criteria in the ***find-routes*** term and returns a true result to the policy expression. The expression result is not guaranteed, so the ***Chablis-routes*** policy is evaluated. The route doesn't match any terms in this policy, which means that the implicit `next policy` action is used. This action is interpreted by the expression as a true result. The expression itself is true, as both policies in the expression are true. The ***Chablis-routes*** policy guaranteed the expression result, so its action is applied to the route. The action was `next policy`, so Merlot takes the 192.168.2.16 /28 route and evaluates it against the next policy in the policy chain—the BGP default policy. The BGP default policy accepts all BGP routes, so the route is advertised to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.2.16/28

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.2.16/28         Self                                    I
```

The evaluation of the 192.168.3.16 /28 route returns a different result. Merlot evaluates the ***Muscat-routes*** policy first, where the route matches the ***nothing-else*** term. This returns a false result to the expression and guarantees a result of false for the entire expression. Since the ***Muscat-routes*** policy guaranteed the result, its action of `reject` is applied to the route and it is not advertised to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.3.16/28

user@Merlot>
```

### A NOT Example

The evaluation and use of the logical NOT operator is a little more straightforward than the OR and AND operators. As such, we apply only a single policy to the Merlot router:

```
[edit protocols bgp]
lab@Merlot# show group Ext-AS65010
```

```
type external;
export ( ! Muscat-routes );
peer-as 65010;
neighbor 10.100.10.2;
```

Merlot evaluates the 192.168.2.16 /28 route against the ***Muscat-routes*** policy, where it matches the ***find-routes*** term and returns a true result. The NOT operator converts this result to a false and applies the `reject` action to the route. It is not advertised to the Cabernet router:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.2.16/28

user@Merlot>
```

The 192.168.3.16 /28 route is evaluated by Merlot against the ***Muscat-routes*** policy, where it matches the ***nothing-else*** term. This return of a false result by the policy is converted into a true result by the NOT operator. The true evaluation implies that the `accept` action is applied to the route and it is advertised to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.3.16/28

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.3.16/28         Self                                    I
```

### A Group Example

The purpose of the logical group operator is to override the default precedence of the OR and AND operators. We can see the functionality of this operator within the network of Figure 1.3. The administrators of AS 65020 would like to advertise only certain customer routes to the EBGP peer of Cabernet. These routes are designated by the BGP community value of ***adv-to-peers*** attached to the route. We can see these routes in the local routing table:

```
user@Merlot> show route terse community-name adv-to-peers

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 192.168.2.48/28    B 170        100          0 >so-0/1/0.0      I
* 192.168.2.64/28    B 170        100          0 >so-0/1/0.0      I
* 192.168.3.48/28    B 170        100          0 >so-0/1/1.0      I
* 192.168.3.64/28    B 170        100          0 >so-0/1/1.0      I
```


We discuss the definition and use of communities within a policy in more detail in the "Communities" section later in this chapter.

Both ***Muscat-routes*** and ***Chablis-routes*** now guarantee a true or false result within the policy through the use of the ***nothing-else*** term. We've also created a policy called ***Check-for-Community*** to look for the ***adv-to-peers*** community value.

```
[edit policy-options]
user@Merlot# show policy-statement Muscat-routes
term find-routes {
    from {
        route-filter 192.168.2.0/24 longer;
    }
    then accept;
}
term nothing-else {
    then reject;
}

[edit policy-options]
user@Merlot# show policy-statement Chablis-routes
term find-routes {
    from {
        route-filter 192.168.3.0/24 longer;
    }
    then accept;
}
term nothing-else {
    then reject;
}

[edit policy-options]
user@Merlot# show policy-statement Check-for-Community
term find-routes {
    from community adv-to-peers;
    then accept;
}
term nothing-else {
    then reject;
}
```

In human terms, we want to advertise only routes that match either the ***Muscat-routes*** or the ***Chablis-routes*** policy as well as the ***Check-for-Community*** policy. To illustrate the usefulness

of the group operator, we first apply the policies using just the OR and AND operators to create a single policy expression (that also occupies a single policy chain spot):

```
[edit protocols bgp group Ext-AS65010]
lab@Merlot# show
type external;
export ( Muscat-routes || Chablis-routes && Check-for-Community );
peer-as 65010;
neighbor 10.100.10.2;
```

If we assume that our thought process is correct, then the 192.168.2.64 /28 route from the Muscat router should be advertised to Cabernet by Merlot. Because the AND operator has precedence over the OR operator, the ***Chablis-routes*** and ***Check-for-Community*** policies are evaluated together first. The route doesn't match the ***Chablis-routes*** policy and returns a false result. This guarantees the result of the expression itself, so the action of `reject` from that policy is applied to the route and it is not advertised to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.2.64/28

user@Merlot>
```

This is clearly not the result we intended, so it appears that the group operator has some usefulness after all! Let's alter the policy expression on Merlot:

```
[edit protocols bgp group Ext-AS65010]
lab@Merlot# show
type external;
export (( Muscat-routes || Chablis-routes ) && Check-for-Community );
peer-as 65010;
neighbor 10.100.10.2;
```

The group operator now causes Merlot to evaluate the ***Muscat-routes*** and ***Chablis-routes*** policies together before evaluating the ***Check-for-Community*** policy. Using the same 192.168.2.64 /28 route, Merlot evaluates the ***Muscat-routes*** policy and gets a true result. This guarantees a true result for the first portion of the expression, so the ***Chablis-routes*** policy is skipped and Merlot evaluates the ***Check-for-Community*** policy. This policy also returns a true result based on the ***find-routes*** term, because the route does indeed have the community attached. The ***Check-for-Community*** policy guaranteed the result of the expression, so its action of `accept` is applied to the route and it is advertised to the Cabernet router:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2 192.168.2.64/28

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.2.64/28         Self                                    I
```

The logic of the group operator applies to all of the routes in the local routing table of Merlot. Only the four routes with the correct community value of ***adv-to-peers*** attached are advertised to Cabernet:

```
user@Merlot> show route advertising-protocol bgp 10.100.10.2

inet.0: 30 destinations, 32 routes (30 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
192.168.2.48/28         Self                                    I
192.168.2.64/28         Self                                    I
192.168.3.48/28         Self                                    I
192.168.3.64/28         Self                                    I
```

# Communities

A *community* is a route attribute used by BGP to administratively group routes with similar properties. We won't be discussing how to use communities in conjunction with BGP in this chapter; we cover these details in Chapter 4, "Border Gateway Protocol." Here we explore how to define a community, apply or delete a community value, and locate a route using a defined community name.

## Regular Communities

A community value is a 32-bit field that is divided into two main sections. The first 16 bits of the value encode the AS number of the network that originated the community, while the last 16 bits carry a unique number assigned by the AS. This system attempts to guarantee a globally unique set of community values for each AS in the Internet.

The JUNOS software uses a notation of `AS-number:community-value`, where each value is a decimal number. The AS values of 0 and 65,535 are reserved, as are all of the community values within those AS numbers. Each community, or set of communities, is given a name within the `[edit policy-options]` configuration hierarchy. The name of the community uniquely identifies it to the router and serves as the method by which routes are categorized. For example, a route with a community value of 65010:1111 might belong to the community named ***AS65010-routes***, once it is configured. The community name is also used within a routing policy as a match criterion or as an action. The command syntax for creating a community is:

```
policy-options {
    community name members [community-ids];
}
```

The *community-ids* field is either a single community value or multiple community values. When more than one value is assigned to a community name, the router interprets this as a logical AND of the community values. In other words, a route must have all of the configured values before being assigned the community name.

**FIGURE 1.4** Communities sample network

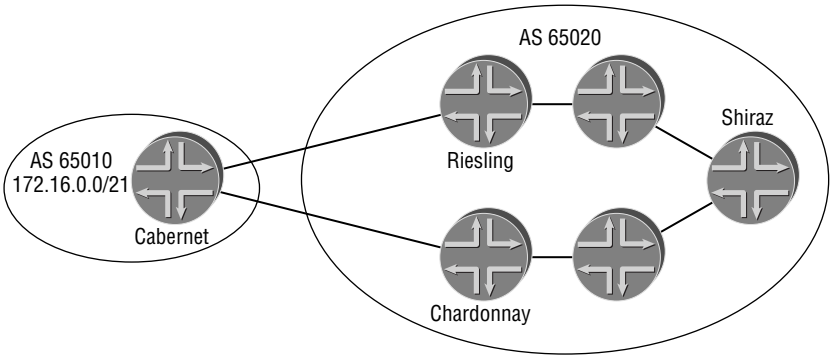



Figure 1.4 shows the Riesling, Chardonnay, and Shiraz routers as IBGP peers in AS 65020. The Cabernet router is advertising the 172.16.0.0 /21 address space from AS 65010. The specific routes received by Riesling include:

```
user@Riesling> show route receive-protocol bgp 10.100.10.1

inet.0: 28 destinations, 36 routes (28 active, 0 holddown, 0 hidden)
Prefix                  Nexthop              MED     Lclpref    AS path
172.16.0.0/24           10.100.10.1          0                  65010 I
172.16.1.0/24           10.100.10.1          0                  65010 I
172.16.2.0/24           10.100.10.1          0                  65010 I
172.16.3.0/24           10.100.10.1          0                  65010 I
172.16.4.0/24           10.100.10.1          0                  65010 I
172.16.5.0/24           10.100.10.1          0                  65010 I
172.16.6.0/24           10.100.10.1          0                  65010 I
172.16.7.0/24           10.100.10.1          0                  65010 I
```

You view the community values attached to each route, if there are any, by adding the `detail` option to the `show route` command:

```
user@Riesling> show route receive-protocol bgp 10.100.10.1 detail

inet.0: 28 destinations, 36 routes (28 active, 0 holddown, 0 hidden)
172.16.0.0/24 (2 entries, 1 announced)
```

```
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:1111 65010:1234

172.16.1.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:1111 65010:1234

172.16.2.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:1234 65010:2222

172.16.3.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:1234 65010:2222

172.16.4.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:3333 65010:4321

172.16.5.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:3333 65010:4321

172.16.6.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:4321 65010:4444
```

```
172.16.7.0/24 (2 entries, 1 announced)
     Nexthop: 10.100.10.1
     MED: 0
     AS path: 65010 I
 Communities: 65010:4321 65010:4444
```

## Match Criteria Usage

The administrators of AS 65010 attached a community value of 65010:1234 to all routes for which they would like to receive user traffic from Riesling. The community value of 65010:4321 is attached to routes for which AS 65010 would like to receive user traffic from Chardonnay. Routing policies within AS 65020 are configured using a community match criterion to effect this administrative goal. The policies change the Local Preference of the received routes to new values that alter the BGP route-selection algorithm. The policies and communities on Riesling look like this:

```
[edit]
user@Riesling# show policy-options
policy-statement alter-local-preference {
    term find-Riesling-routes {
        from community out-via-Riesling;
        then {
            local-preference 200;
        }
    }
    term find-Chardonay-routes {
        from community out-via-Chardonnay;
        then {
            local-preference 50;
        }
    }
}
community out-via-Chardonnay members 65010:4321;
community out-via-Riesling members 65010:1234;
```

A similar policy is configured on Chardonnay with the Local Preference values reversed. The policy on Riesling is applied as an import policy to alter the attributes as they are received from Cabernet:

```
[edit protocols bgp]
user@Riesling# show group Ext-AS65010
type external;
import alter-local-preference;
peer-as 65010;
neighbor 10.100.10.1;
```

We check the success of the policy on the Shiraz router. The 172.16.0.0 /24 route should use the advertisement from Riesling (192.168.1.1), while the 172.16.4.0 /24 route should use the advertisement from Chardonnay (192.168.3.3):

```
user@Shiraz> show route 172.16.0/24

inet.0: 28 destinations, 31 routes (28 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.0.0/24      *[BGP/170] 00:08:30, MED 0, localpref 200, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0

user@Shiraz> show route 172.16.4/24

inet.0: 28 destinations, 31 routes (28 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.4.0/24      *[BGP/170] 00:04:58, MED 0, localpref 200, from 192.168.3.3
                      AS path: 65010 I
                    > via so-0/1/1.0
```

It appears the policies are working as designed. We've successfully located the BGP routes using a single community value in the ***alter-local-preference*** policies. The JUNOS software also allows you to locate routes containing multiple community values. One method of accomplishing this is to create two community names and reference those names in your routing policies. Or, you create a single community name with both values and reference that single name in the policy. Let's see how these two options work on the Shiraz router.

The administrators of AS 65020 decide that they would like to reject all routes on Shiraz containing both the 65010:4321 and 65010:4444 community values. We first create three separate community names: one each for the single values and one for the combined values.

```
[edit policy-options]
user@Shiraz# show
community both-comms members [ 65010:4321 65010:4444 ];
community just-4321 members 65010:4321;
community just-4444 members 65010:4444;
```

We locate the current routes in the routing table that have these values by using the `community` or `community-name` options of the `show route` command. The `community` option allows you to enter a numerical community value and the router outputs all routes containing that value.

```
user@Shiraz> show route terse community 65010:4321

inet.0: 28 destinations, 31 routes (28 active, 0 holddown, 0 hidden)
```

```
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 172.16.4.0/24      B 170        200          0 >so-0/1/1.0      65010 I
* 172.16.5.0/24      B 170        200          0 >so-0/1/1.0      65010 I
* 172.16.6.0/24      B 170        200          0 >so-0/1/1.0      65010 I
* 172.16.7.0/24      B 170        200          0 >so-0/1/1.0      65010 I

user@Shiraz> show route terse community 65010:4444

inet.0: 28 destinations, 31 routes (28 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 172.16.6.0/24      B 170        200          0 >so-0/1/1.0      65010 I
* 172.16.7.0/24      B 170        200          0 >so-0/1/1.0      65010 I
```

It appears that the 172.16.6.0 /24 and 172.16.7.0 /24 routes have both community values attached to them. We can confirm this with the `show route detail` command to view the actual values, but we have another method at our disposal. The `community-name` option allows you to specify a configured name and have the router output the routes matching that community value. The ***both-comms*** community is configured with multiple members so that only routes currently containing both community values match this community name.

```
user@Shiraz> show route terse community-name both-comms

inet.0: 28 destinations, 31 routes (28 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 172.16.6.0/24      B 170        200          0 >so-0/1/1.0      65010 I
* 172.16.7.0/24      B 170        200          0 >so-0/1/1.0      65010 I
```

We create two different policies on Shiraz and apply them separately as an import policy for the IBGP peer group. The first policy uses the single community match criteria of ***both-comms***:

```
[edit policy-options]
user@Shiraz# show
policy-statement single-comm-match {
    term use-just-one-comm {
        from community both-comms;
        then reject;
    }
```

```
}
community both-comms members [ 65010:4321 65010:4444 ];
community just-4321 members 65010:4321;
community just-4444 members 65010:4444;

[edit protocols bgp]
user@Shiraz# set group Internal-Peers import single-comm-match

[edit]
user@Shiraz# commit and-quit
commit complete
Exiting configuration mode

user@Shiraz> show route 172.16.5/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.5.0/24      *[BGP/170] 01:27:54, MED 0, localpref 200, from 192.168.3.3
                      AS path: 65010 I
                    > via so-0/1/1.0

user@Shiraz> show route 172.16.6/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)

user@Shiraz> show route 172.16.7/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)

user@Shiraz>
```

The routes are no longer in the `inet.0` routing table on Shiraz. The logical AND within the community definition correctly located only the routes containing both community values. We now create a second policy, called ***double-comm-match***, using the individual community names:

```
[edit policy-options policy-statement double-comm-match]
user@Shiraz# show
term two-comms {
    from community [ just-4321 just-4444 ];
    then reject;
}
```

```
[edit policy-options policy-statement double-comm-match]
user@Shiraz# top edit protocols bgp

[edit protocols bgp]
user@Shiraz# show group Internal-Peers
type internal;
local-address 192.168.7.7;
import double-comm-match;
neighbor 192.168.1.1;
neighbor 192.168.2.2;
neighbor 192.168.3.3;
neighbor 192.168.4.4;
neighbor 192.168.5.5;
neighbor 192.168.6.6;
```

After committing our configuration, we check the success of our new policy:

```
user@Shiraz> show route 172.16.5/24

inet.0: 28 destinations, 31 routes (24 active, 0 holddown, 4 hidden)

user@Shiraz> show route 172.16.6/24

inet.0: 28 destinations, 31 routes (24 active, 0 holddown, 4 hidden)

user@Shiraz> show route 172.16.7/24

inet.0: 28 destinations, 31 routes (24 active, 0 holddown, 4 hidden)
```

As you can see, something isn't right. The 172.16.5.0 /24 route should be active in the routing table, but it is not there. In addition, we now have four hidden routes whereas we had only two hidden routes using the ***single-comm-match*** policy. Let's see what routes are now hidden:

```
user@Shiraz> show route terse hidden

inet.0: 28 destinations, 31 routes (24 active, 0 holddown, 4 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
  172.16.4.0/24      B            200          0 >so-0/1/1.0      65010 I
  172.16.5.0/24      B            200          0 >so-0/1/1.0      65010 I
```

```
  172.16.6.0/24      B            200          0 >so-0/1/1.0      65010 I
  172.16.7.0/24      B            200          0 >so-0/1/1.0      65010 I
```

Something in the ***double-comm-match*** policy is rejecting more routes than we would like. The policy currently is configured like this:

```
user@Shiraz> show configuration policy-options
policy-statement single-comm-match {
    term use-just-one-comm {
        from community both-comms;
        then reject;
    }
}
policy-statement double-comm-match {
    term two-comms {
        from community [ just-4321 just-4444 ];
        then reject;
    }
}
community both-comms members [ 65010:4321 65010:4444 ];
community just-4321 members 65010:4321;
community just-4444 members 65010:4444;
```

The highlighted portion of the policy is where our problems are arising. Listing multiple values in square brackets ([]) within the community configuration itself is a logical AND of the values. We proved this with the ***both-comms*** community. The same theory doesn't hold true within a routing policy itself, where listing multiple values within a set of square brackets results in a logical OR operation. The ***double-comm-match*** policy is actually locating routes with either the ***just-4321*** community or the ***just-4444*** community value attached. To effectively locate the correct routes using the individual community values, we actually require two policies applied in a policy chain. The first policy locates routes with one of the communities attached and moves their evaluation to the next policy in the chain. The first policy then accepts all other routes. The second policy in the chain locates routes with the second community value attached and rejects them while also accepting all routes. The relevant policies are configured as so:

```
[edit policy-options]
user@Shiraz# show policy-statement find-4321
term 4321-routes {
    from community just-4321;
    then next policy;
}
term all-other-routes {
    then accept;
```

```
}

[edit policy-options]
user@Shiraz# show policy-statement find-4444
term 4444-routes {
    from community just-4444;
    then reject;
}
term all-other-routes {
    then accept;
}
```

We apply the policies to the IBGP peer group in the proper order and verify that the correct routes are rejected:

```
[edit protocols bgp]
user@Shiraz# show group Internal-Peers
type internal;
local-address 192.168.7.7;
import [ find-4321 find-4444 ];
neighbor 192.168.1.1;
neighbor 192.168.2.2;
neighbor 192.168.3.3;
neighbor 192.168.4.4;
neighbor 192.168.5.5;
neighbor 192.168.6.6;

[edit]
user@Shiraz# commit and-quit
commit complete
Exiting configuration mode

user@Shiraz> show route 172.16.5/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.5.0/24      *[BGP/170] 02:00:58, MED 0, localpref 200, from 192.168.3.3
                      AS path: 65010 I
                    > via so-0/1/1.0
```

```
user@Shiraz> show route 172.16.6/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)

user@Shiraz> show route 172.16.7/24

inet.0: 28 destinations, 31 routes (26 active, 0 holddown, 2 hidden)

user@Shiraz>
```

The 172.16.5.0 /24 route is active in the routing table and neither the 172.16.6.0 /24 or 172.16.7.0 /24 routes are present. The router output of two hidden routes also provides a hint that the policies are working as designed. While this application might seem a bit complex, the use of regular expressions (as outlined in the "Regular Expressions" section later in this chapter) makes the routing policy configuration more straightforward.

## Modifying Current Values

Altering the current values attached to a route is the other main use of a community in a routing policy. You can perform three main actions: you can add, delete, or set a community value. Here are the details of each policy action:

`add` The policy action `then community add` *`community-name`* maintains the current list of communities on the route and adds to it the community values defined in *`community-name`*.

`delete` The policy action `then community delete` *`community-name`* also maintains the current list of communities on the route while removing all community values defined in *`community-name`*.

`set` The policy action `then community set` *`community-name`* deletes all of the current communities assigned to the route. In its place, the router installs the community values defined in *`community-name`*.

The administrators of AS 65010 in Figure 1.4 want to alter the community values on the routes they receive from Riesling. The routes currently in the routing table include:

```
user@Cabernet> show route protocol bgp

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.1.0/24     *[BGP/170] 00:07:34, localpref 100
                      AS path: 65020 I
                    > to 10.100.10.2 via at-0/1/0.100
192.168.2.0/24     *[BGP/170] 00:07:34, localpref 100
                      AS path: 65020 I
                    > to 10.100.10.2 via at-0/1/0.100
```

```
192.168.3.0/24     *[BGP/170] 00:07:34, localpref 100
                      AS path: 65020 I
                    > to 10.100.10.2 via at-0/1/0.100
```

Cabernet wants to add a community value of 65010:1 to the 192.168.1.0 /24 route. We configure the appropriate policy and apply it to Riesling after examining the current community values on the route:

```
[edit]
user@Cabernet# run show route 192.168.1/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.1.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 11:15
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
                AS path: 65020 I
                Communities: 65020:1 65020:10 65020:100 65020:1000
                Localpref: 100
                Router ID: 192.168.1.1

[edit]
user@Cabernet# show policy-options
policy-statement add-a-community {
    term add-comm {
        from {
            route-filter 192.168.1.0/24 exact;
        }
        then {
            community add comm-1;
        }
    }
}
community comm-1 members 65010:1;

[edit]
user@Cabernet# show protocols bgp
```

```
group Ext-AS65020 {
    type external;
    import add-a-community;
    peer-as 65020;
    neighbor 10.100.10.2;
}

user@Cabernet> show route 192.168.1/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.1.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 12:11
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
                AS path: 65020 I
                Communities: 65010:1 65020:1 65020:10 65020:100 65020:1000
                Localpref: 100
                Router ID: 192.168.1.1
```

The router output clearly shows the 65010:1 community value added to the 192.168.1.0 /24 route as a result of the ***add-a-community*** policy. We back out our changes and create a policy to remove the 65020:200 community value from the 192.168.2.0 /24 route. As before, we view the route before and after the policy application:

```
[edit]
user@Cabernet# run show route 192.168.2/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.2.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 18:23
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
```

```
                AS path: 65020 I
                Communities: 65020:2 65020:20 65020:200 65020:2000
                Localpref: 100
                Router ID: 192.168.1.1

[edit]
user@Cabernet# show policy-options
policy-statement delete-a-community {
    term delete-comm {
        from {
            route-filter 192.168.2.0/24 exact;
        }
        then {
            community delete comm-2;
        }
    }
}
community comm-2 members 65020:200;

[edit]
user@Cabernet# show protocols bgp
group Ext-AS65020 {
    type external;
    import delete-a-community;
    peer-as 65020;
    neighbor 10.100.10.2;
}

user@Cabernet> show route 192.168.2/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.2.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 18:53
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
```

```
                AS path: 65020 I
                Communities: 65020:2 65020:20 65020:2000
                Localpref: 100
                Router ID: 192.168.1.1
```

The ***delete-a-community*** policy removed the 65020:200 community value from the 192.168.2.0 /24 route without deleting the other existing values as we expected. We again back out our changes and use the `set` community action to remove all community values attached to the 192.168.3.0 /24 route. In their place, Cabernet adds the 65010:33 community value to the route:

```
[edit]
user@Cabernet# run show route 192.168.3/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.3.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 23:29
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
                AS path: 65020 I
                Communities: 65020:3 65020:30 65020:300 65020:3000
                Localpref: 100
                Router ID: 192.168.1.1

[edit]
user@Cabernet# show policy-options
policy-statement set-a-community {
    term set-comm {
        from {
            route-filter 192.168.3.0/24 exact;
        }
        then {
            community set comm-3;
        }
    }
}
community comm-3 members 65010:33;
```

```
[edit]
user@Cabernet# show protocols bgp
group Ext-AS65020 {
    type external;
    import set-a-community;
    peer-as 65020;
    neighbor 10.100.10.2;
}

user@Cabernet> show route 192.168.3/24 detail

inet.0: 20 destinations, 20 routes (20 active, 0 holddown, 0 hidden)
192.168.3.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.100.10.2
                Next hop: 10.100.10.2 via at-0/1/0.100, selected
                State: <Active Ext>
                Local AS: 65010 Peer AS: 65020
                Age: 23:49
                Task: BGP_65020.10.100.10.2+2698
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
                AS path: 65020 I
                Communities: 65010:33
                Localpref: 100
                Router ID: 192.168.1.1
```

As we expected, the ***set-a-community*** policy removed the existing community values and in its place inserted the 65010:33 value.

## Extended Communities

Recent networking enhancements, such as virtual private networks (VPN), have functionality requirements that can be satisfied by an attribute such as a community. (We discuss VPNs in more detail in Chapter 9, "Layer 2 and Layer 3 Virtual Private Networks.") However, the existing 4-octet community value doesn't provide enough expansion and flexibility to accommodate the requirements that would be put on it. This leads to the creation of extended communities. An *extended community* is an 8-octet value that is also divided into two main sections. The first 2 octets of the community encode a type field while the last 6 octets carry a unique set of data in a format defined by the type field.

Figure 1.5 shows the format of the extended community attribute. The individual fields are defined as:

**Type (2 octets)** The type field designates both the format of the remaining community fields as well as the actual kind of extended community being used.

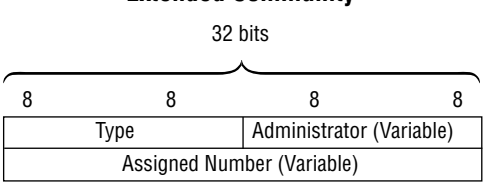
The high-order octet uses the two defined values of 0x00 and 0x01. A value of 0x00 denotes a 2-octet administrator field and a 4-octet assigned number field. The 0x01 value results in the opposite: a 4-octet administrator field and a 2-octet assigned number field.

The low-order octet determines the kind of community used. Two common values are 0x02 (a route target community) and 0x03 (a route origin community).

**Administrator (Variable)** The variable-sized administrator field contains information designed to guarantee the uniqueness of the extended community. The AS number of the network originating the community is used when 2 octets are available, and an IPv4 prefix is used when 4 octets are available. The prefix is often the router ID of the device originating the community.

**Assigned Number (Variable)** The assigned number field is also variably sized to either 2 or 4 octets. It contains a value assigned by the originating network. When combined with the administrator field, the community value is designed to be unique in the Internet.

**FIGURE 1.5** Extended community format




The JUNOS software provides the same command syntax for an extended community as a regular community. The difference is in the *community-id* value supplied. An extended community uses a notation of `type:administrator:assigned-number`. The router expects you to use the words `target` or `origin` to represent the type field. The administrator field uses a decimal number for the AS or an IPv4 address, while the assigned number field expects a decimal number no larger than the size of the field (65,535 for 2 octets or 4,294,967,295 for 4 octets).

You use the defined community name for an extended community in the same manner as for a regular community. You can match on a route or modify the route attributes using the `add`, `delete`, or `set` keywords. Refer back to Figure 1.4 and the Shiraz router in AS 65020. Shiraz has local static routes representing customer networks, which have existing regular community values assigned to them. Shiraz adds extended community values to the routes before advertising them via BGP. The existing routes are:

```
[edit]
user@Shiraz# show routing-options
static {
```

```
    route 192.168.1.0/24 {
        next-hop 10.222.6.1;
        community 65020:1;
    }
    route 192.168.2.0/24 {
        next-hop 10.222.6.1;
        community 65020:2;
    }
    route 192.168.3.0/24 {
        next-hop 10.222.6.1;
        community 65020:3;
    }
    route 192.168.4.0/24 {
        next-hop 10.222.6.1;
        community 65020:4;
    }
}
```

Shiraz creates four extended communities: one for each possible combination of type, administrator field size, and assigned number field size. The communities are associated on a one-to-one basis with a route using an export policy:

```
[edit]
user@Shiraz# show policy-options
policy-statement set-ext-comms {
    term route-1 {
        from {
            route-filter 192.168.1.0/24 exact;
        }
        then {
            community add target-as;
            accept;
        }
    }
    term route-2 {
        from {
            route-filter 192.168.2.0/24 exact;
        }
        then {
            community add target-ip;
            accept;
        }
    }
```

```
    term route-3 {
        from {
            route-filter 192.168.3.0/24 exact;
        }
        then {
            community add origin-as;
            accept;
        }
    }
    term route-4 {
        from {
            route-filter 192.168.4.0/24 exact;
        }
        then {
            community add origin-ip;
            accept;
        }
    }
}
community origin-as members origin:65020:3;
community origin-ip members origin:192.168.7.7:4;
community target-as members target:65020:1;
community target-ip members target:192.168.7.7:2;

[edit]
user@Shiraz# show protocols bgp
group Internal-Peers {
    type internal;
    local-address 192.168.7.7;
    export set-ext-comms;
    neighbor 192.168.1.1;
    neighbor 192.168.2.2;
    neighbor 192.168.3.3;
    neighbor 192.168.4.4;
    neighbor 192.168.5.5;
    neighbor 192.168.6.6;
}
```

The routes are received on the Riesling router with the correct community values attached:

```
user@Riesling> show route protocol bgp 192.168/16 detail
```

```
inet.0: 32 destinations, 32 routes (32 active, 0 holddown, 0 hidden)
192.168.1.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 192.168.7.7
                Next hop: 10.222.4.2 via fe-0/0/2.0, selected
                Protocol next hop: 10.222.6.1 Indirect next hop: 85a3000 62
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 1:58       Metric2: 3
                Task: BGP_65020.192.168.7.7+1562
                Announcement bits (3): 0-KRT 1-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:1 target:65020:1
                Localpref: 100
                Router ID: 192.168.7.7

192.168.2.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 192.168.7.7
                Next hop: 10.222.4.2 via fe-0/0/2.0, selected
                Protocol next hop: 10.222.6.1 Indirect next hop: 85a3000 62
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 1:58       Metric2: 3
                Task: BGP_65020.192.168.7.7+1562
                Announcement bits (3): 0-KRT 1-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:2 target:192.168.7.7:2
                Localpref: 100
                Router ID: 192.168.7.7

192.168.3.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 192.168.7.7
                Next hop: 10.222.4.2 via fe-0/0/2.0, selected
                Protocol next hop: 10.222.6.1 Indirect next hop: 85a3000 62
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 1:58       Metric2: 3
                Task: BGP_65020.192.168.7.7+1562
```

```
                Announcement bits (3): 0-KRT 1-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:3 origin:65020:3
                Localpref: 100
                Router ID: 192.168.7.7

192.168.4.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 192.168.7.7
                Next hop: 10.222.4.2 via fe-0/0/2.0, selected
                Protocol next hop: 10.222.6.1 Indirect next hop: 85a3000 62
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 1:58       Metric2: 3
                Task: BGP_65020.192.168.7.7+1562
                Announcement bits (3): 0-KRT 1-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:4 origin:192.168.7.7:4
                Localpref: 100
                Router ID: 192.168.7.7
```

## Regular Expressions

The definition of your community within `[edit policy-options]` can contain decimal values, as we've already done, or a regular expression. A *regular expression* (regex) uses nondecimal characters to represent decimal values. This allows you the flexibility of specifying any number of community values in a single community name. When used with communities, as opposed to a BGP AS Path, the JUNOS software uses two different forms—simple and complex. Let's explore the difference between these regex types.

### Simple Community Expressions

A simple community regular expression uses either the asterisk (`*`) or the dot (`.`) to represent some value. The asterisk represents an entire AS number or an entire community value. Some examples of a regular expression using the asterisk are:

`*:1111` Matches a community with any possible AS number and a community value of 1111.

`65010:*` Matches a community from AS 65010 with any possible community value.

The dot represents a single decimal place in either the AS number or the community value. Examples of regular expressions using the dot are:

`65010:100.` Matches a community with an AS of 65010 and a community value that is four digits long, whereas the community value begins with 100. These values include 1000, 1001, 1002, …, 1009.

`65010:2...4` Matches a community from AS 65010 with a community value that is five digits long. The first digit of the community value must be 2 and the last digit must be 4. Some possible values are 23754, 21114, and 29064.

`650.0:4321` Matches a community with a community value of 4321 and an AS number that is five digits long. The fourth digit of the AS number can be any value. The AS numbers include 65000, 65010, 65020, …, 65090.

To classify as a simple regular expression, the asterisk and the dot must be used separately. Using them together (`.*`) results in a complex community regular expression. We discuss complex expressions in the "Complex Community Expressions" section later in the chapter.

Refer back to Figure 1.4 as a guide. Here, the Shiraz router is receiving routes within the 172.16.0.0 /21 address space from AS 65010. Those routes currently have the following community values assigned to them:

```
user@Shiraz> show route 172.16.0/21 detail | match Communities
                Communities: 65010:1111 65010:1234
                Communities: 65010:1111 65010:1234
                Communities: 65010:1234 65010:2222
                Communities: 65010:1234 65010:2222
                Communities: 65010:3333 65010:4321
                Communities: 65010:3333 65010:4321
                Communities: 65010:4321 65010:4444
                Communities: 65010:4321 65010:4444
```

At this point we don't know which values are attached to which routes; we only know the list of possible values within the address range. We use the `show route community` *`community-value`* command in conjunction with some simple regular expressions to accomplish this:

```
user@Shiraz> show route community *:1111

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.0.0/24      *[BGP/170] 22:41:39, MED 0, localpref 200, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
172.16.1.0/24      *[BGP/170] 22:41:39, MED 0, localpref 200, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
```

The 172.16.0.0 /24 and the 172.16.1.0 /24 routes have a community attached with a community value of 1111. The asterisk regex allows the AS number to be any value, although our previous capture tells us it is 65010. We see the actual communities by adding the `detail` option to the command:

```
user@Shiraz> show route community *:1111 detail

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
172.16.0.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-201
                Source: 192.168.1.1
                Next hop: via so-0/1/0.0, selected
                Protocol next hop: 192.168.1.1 Indirect next hop: 8570738 120
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 22:43:48   Metric: 0       Metric2: 65536
                Task: BGP_65020.192.168.1.1+179
                Announcement bits (2): 0-KRT 4-Resolve inet.0
                AS path: 65010 I
                Communities: 65010:1111 65010:1234
                Localpref: 200
                Router ID: 192.168.1.1

172.16.1.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-201
                Source: 192.168.1.1
                Next hop: via so-0/1/0.0, selected
                Protocol next hop: 192.168.1.1 Indirect next hop: 8570738 120
                State: <Active Int Ext>
                Local AS: 65020 Peer AS: 65020
                Age: 22:43:48   Metric: 0       Metric2: 65536
                Task: BGP_65020.192.168.1.1+179
                Announcement bits (2): 0-KRT 4-Resolve inet.0
                AS path: 65010 I
                Communities: 65010:1111 65010:1234
                Localpref: 200
                Router ID: 192.168.1.1
```

The routes on Shiraz with a community from AS 65010 and a community value four digits long that begins with 4 are:

```
user@Shiraz> show route community 65010:4...
```

```
inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.4.0/24      *[BGP/170] 20:32:12, MED 0, localpref 50, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
172.16.5.0/24      *[BGP/170] 20:32:12, MED 0, localpref 50, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
172.16.6.0/24      *[BGP/170] 20:32:12, MED 0, localpref 50, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
172.16.7.0/24      *[BGP/170] 20:32:12, MED 0, localpref 50, from 192.168.1.1
                      AS path: 65010 I
                    > via so-0/1/0.0
```

The JUNOS software also provides the ability to combine the asterisk and dot regular expressions. For example, Shiraz displays the routes whose community is from any AS and whose value is four digits long ending with 1:

```
user@Shiraz> show route terse community *:...1

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* 172.16.0.0/24      B 170        200          0 >so-0/1/0.0      65010 I
* 172.16.1.0/24      B 170        200          0 >so-0/1/0.0      65010 I
* 172.16.4.0/24      B 170         50          0 >so-0/1/0.0      65010 I
* 172.16.5.0/24      B 170         50          0 >so-0/1/0.0      65010 I
* 172.16.6.0/24      B 170         50          0 >so-0/1/0.0      65010 I
* 172.16.7.0/24      B 170         50          0 >so-0/1/0.0      65010 I
```

## Complex Community Expressions

A complex community regular expression allows for a more varied set of combinations than a simple expression does. The complex regex uses both a *regular expression term* in conjunction with a *regular expression operator*. The regex term is any single character within the community, including both the actual decimal digits and the simple dot (`.`) regex. The operator is an optional character that applies to a single term and usually follows that term. The JUNOS software allows you to combine multiple term-operator pairs within a single community definition. Table 1.1 displays the regular expression operators supported by the router.

**TABLE 1.1** Community Regular Expression Operators

| Operator | Description |
|---|---|
| `{m,n}` | Matches at least `m` and at most `n` instances of the term. |
| `{m}` | Matches exactly `m` instances of the term. |
| `{m,}` | Matches `m` or more instances of the term, up to infinity. |
| * | Matches 0 or more instances of the term, which is similar to `{0,}`. |
| + | Matches one or more instances of the term, which is similar to `{1,}`. |
| ? | Matches 0 or 1 instances of the term, which is similar to `{0,1}`. |
| \| | Matches one of the two terms on either side of the pipe symbol, similar to a logical OR. |
| ^ | Matches a term at the beginning of the community attribute. |
| $ | Matches a term at the end of the community attribute. |
| [] | Matches a range or an array of digits. This occupies the space of a single term within the community attribute. |
| (…) | Groups terms together to be acted on by an additional operator. |

### An Effective Use of a Simple Expression

The format and design of the community attribute means that each community should be globally unique. Router implementations, however, don't provide a sanity check on received routes looking for communities belonging to your local AS. In other words, some other network may attach a community value that "belongs" to you. To combat this, some network administrators remove all community values from each received BGP route. Of course, this is helpful only when your local administrative policy is not expecting community values from a peer. When this is not the case, you should honor the expected community values before removing the unexpected values.

A typical configuration that might accomplish the removal of all community values is shown in the ***delete-all-comms*** policy:

```
[edit policy-options]
user@Muscat# show
```

```
policy-statement delete-all-comms {
    term remove-comms {
        community delete all-comms;
    }
}
community all-comms members *:*;
```

This policy doesn't contain any match criteria, so all possible routes match the ***remove-comms*** term. The action is then to delete all communities that match the ***all-comms*** community name. The named community uses a regular expression to match all possible AS numbers and all possible community values. After applying the ***delete-all-comms*** policy as an import from its EBGP peers, the Muscat router can test its effectiveness:

```
user@Muscat> show route receive-protocol bgp 10.222.45.1 detail

inet.0: 35 destinations, 35 routes (35 active, 0 holddown, 0 hidden)
* 172.16.1.0/24 (1 entry, 1 announced)
     Nexthop: 10.222.45.1
     AS path: 65030 65020 65010 I
 Communities: 65010:1111 65010:1234

user@Muscat> show route 172.16.1/24 detail

inet.0: 35 destinations, 35 routes (35 active, 0 holddown, 0 hidden)
172.16.1.0/24 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Source: 10.222.45.1
                Next hop: 10.222.45.1 via so-0/1/1.0, selected
                State: <Active Ext>
                Local AS: 65040 Peer AS: 65030
                Age: 1:42:14
                Task: BGP_65030.10.222.45.1+179
                Announcement bits (2): 0-KRT 1-BGP.0.0.0.0+179
                AS path: 65030 65020 65010 I
                Localpref: 100
                Router ID: 192.168.4.4
```

The lack of communities in the local `inet.0` routing table proves the effectiveness of the regular expression. If the administrators of Muscat want to use communities within their own AS, they can easily apply them in a second term or another import policy.

The use of the caret (^) and dollar sign ($) operators as anchors for your community regular expression is optional. However, we recommend their use for clarity in creating and using expressions with BGP communities.

Examples of complex regular expressions include the following:

`^65000:.{2,3}$` This expression matches a community value where the AS number is 65000. The community value is any two- or three-digit number. Possible matches include 65000:123, 65000:16, and 65000:999.

`^65010:45.{2}9$` This expression matches a community value where the AS number is 65010. The community value is a five-digit number that begins with 45 and ends with 9. The third and fourth digits are any single number repeated twice. Possible matches include 65010:45119, 65010:45999, and 65010:45339.

`^65020:.*$` This expression matches a community value where the AS number is 65020. The community value is any possible combination of values from 0 through 65,535. The `.*` notation is useful for representing any value any number of times.

`^65030:84+$` This expression matches a community value where the AS number is 65030. The community value must start with 8 and include between one and four instances of 4. Matches are 65030:84, 65030:844, 65030:8444, and 65030:84444.

`^65040:234?$` This expression matches a community value where the AS number is 65040. The community value is either 23 or 234, which results in the matches being 65040:23 and 65040:234.

`^65050:1|2345$` This expression matches a community value where the AS number is 65050. The community value is either 1345 or 2345, which results in the matches being 65050:1345 and 65050:2345. You can also write the regex as `^65050:(1|2)345$` for added clarity.

`^65060:1[357]9$` This expression matches a community value where the AS number is 65060. The community value is 139, 159, or 179, which results in the matches being 65060:139, 65060:159, and 65060:179.

`^65070:1[3-7]9$` This expression matches a community value where the AS number is 65070. The community value is a three-digit number that starts with 1 and ends with 9. The second digit is any single value between 3 and 7. The matches for this regex are 65070:139, 65070:149, 65070:159, 65070:169, and 65060:179.

While we explored complex regular expressions only within the community value, the JUNOS software also allows expressions within the AS number. For example, `^65.{3}:1234$` matches any private AS number starting with 65 and a community value of 1234.

The Shiraz router in Figure 1.4 has local customer static routes it is advertising to its IBGP peers. These routes and their communities are:

```
user@Shiraz> show route protocol static detail

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
192.168.1.0/24 (1 entry, 1 announced)
        *Static Preference: 5
                Next hop: 10.222.6.1 via so-0/1/2.0, selected
                State: <Active Int Ext>
                Local AS: 65020
                Age: 1:21
                Task: RT
                Announcement bits (3): 0-KRT 3-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:1 65020:10 65020:11 65020:100 65020:111

192.168.2.0/24 (1 entry, 1 announced)
        *Static Preference: 5
                Next hop: 10.222.6.1 via so-0/1/2.0, selected
                State: <Active Int Ext>
                Local AS: 65020
                Age: 1:21
                Task: RT
                Announcement bits (3): 0-KRT 3-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:2 65020:20 65020:22 65020:200 65020:222

192.168.3.0/24 (1 entry, 1 announced)
        *Static Preference: 5
                Next hop: 10.222.6.1 via so-0/1/2.0, selected
                State: <Active Int Ext>
                Local AS: 65020
                Age: 1:21
                Task: RT
                Announcement bits (3): 0-KRT 3-BGP.0.0.0.0+179 4-Resolve inet.0
                AS path: I
                Communities: 65020:3 65020:30 65020:33 65020:300 65020:333

192.168.4.0/24 (1 entry, 1 announced)
        *Static Preference: 5
```

```
Next hop: 10.222.6.1 via so-0/1/2.0, selected
State: <Active Int Ext>
Local AS: 65020
Age: 1:21
Task: RT
Announcement bits (3): 0-KRT 3-BGP.0.0.0.0+179 4-Resolve inet.0
AS path: I
Communities: 65020:4 65020:40 65020:44 65020:400 65020:444
```

To adequately test complex regular expressions, Shiraz creates a policy called ***test-regex*** that locates routes by using a complex regular expression and rejects all other routes. The policy is configured like this:

```
[edit]
user@Shiraz# show policy-options policy-statement test-regex
term find-routes {
    from community complex-regex;
    then accept;
}
term reject-all-else {
    then reject;
}
```

The complex regular expression is currently set to match on community values beginning with either 1 or 3. Here's the configuration:

```
[edit]
user@Shiraz# show policy-options | match members
community complex-regex members "^65020:[13].*$";
```

The 192.168.1.0 /24 and 192.168.3.0/24 routes both have communities attached that should match this expression. We test the regex and its policy by using the `test policy` *`policy-name`* command:

```
user@Shiraz> test policy test-regex 0/0

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.1.0/24     *[Static/5] 00:31:44
                    > to 10.222.6.1 via so-0/1/2.0
192.168.3.0/24     *[Static/5] 00:31:44
                    > to 10.222.6.1 via so-0/1/2.0
```

```
Policy test-regex: 2 prefix accepted, 30 prefix rejected
```

The complex regular expression is altered to match on any community value containing any number of instances of the digit 2. The new expression configuration and the associated routes are shown here:

```
[edit]
user@Shiraz# show policy-options | match members
community complex-regex members "^65020:2+$";

user@Shiraz> test policy test-regex 0/0

inet.0: 32 destinations, 35 routes (32 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.2.0/24     *[Static/5] 00:40:28
                    > to 10.222.6.1 via so-0/1/2.0

Policy test-regex: 1 prefix accepted, 31 prefix rejected
```

# Autonomous System Paths

An *AS Path* is also a route attribute used by BGP. The AS Path is used both for route selection and to prevent potential routing loops. As with the communities, we won't discuss the details of using AS Paths within BGP in this chapter; those details are covered in Chapter 4. The topics concerning us in this chapter are defining regular expressions and using those expressions to locate a set of routes.

## Regular Expressions

An AS Path regular expression also uses a term-operator format similar to the complex community regular expressions. Unlike the community term, the AS Path regular expression term is an entire AS number, such as 65000 or 65432. This translates into the simple dot (`.`) regex representing an entire AS number. Table 1.2 displays the AS Path regular expression operators supported by the router.

Examples of AS Path regular expressions include:

`65000` This expression matches an AS Path with a length of 1 whose value is 65000. The expression uses a single term with no operators.

**TABLE 1.2** AS Path Regular Expression Operators

| Operator | Description |
|---|---|
| `{m,n}` | Matches at least `m` and at most `n` instances of the term. |
| `{m}` | Matches exactly `m` instances of the term. |
| `{m,}` | Matches `m` or more instances of the term, up to infinity. |
| * | Matches 0 or more instances of the term, which is similar to `{0,}`. |
| + | Matches one or more instances of the term, which is similar to `{1,}`. |
| ? | Matches 0 or 1 instances of the term, which is similar to `{0,1}`. |
| \| | Matches one of the two terms on either side of the pipe symbol, similar to a logical OR. |
| - | Matches an inclusive range of terms. |
| ^ | Matches the beginning of the AS Path. The JUNOS software uses this operator implicitly and its use is optional. |
| $ | Matches the end of the AS Path. The JUNOS software uses this operator implicitly and its use is optional. |
| (…) | Groups terms together to be acted on by an additional operator. |
| () | Matches a null value as a term. |

`65010 . 65020` This expression matches an AS Path with a length of 3 where the first AS is 65010 and the last AS is 65020. The AS in the middle of the path can be any single AS number.

`65030?` This expression matches an AS Path with a length of 0 or 1. A path length of 0 is represented by the null AS Path. If a value appears, it must be 65030.

`. (65040|65050)?` This expression matches an AS Path with a length of 1 or 2. The first AS in the path can be any value. The second AS in the path, if appropriate, must be either 65040 or 65050.

`65060 .*` This expression matches an AS Path with a length of at least 1. The first AS number must be 65060, and it may be followed by any other AS number any number of times or no AS numbers. This expression is often used to represent all BGP routes from a particular neighboring AS network.

`.* 65070` This expression matches an AS Path with a length of at least 1. The last AS number must be 65070, and it may be preceded by any other AS number any number of times or no AS numbers. This expression is often used to represent all BGP routes that originated from a particular AS network.

`.* 65080 .*` This expression matches an AS Path with a length of at least 1. The 65080 AS number must appear at least once in the path. It may be followed by or preceded by any other AS number any number of times. This expression is often used to represent all BGP routes that have been routed by a particular AS network.

`.* (64512-65535) .*` This expression matches an AS Path with a length of at least 1. One of the private AS numbers must appear at least once in the path. It may be followed by or preceded by any other AS number any number of times. This expression is useful at the edge of a network to reject routes containing private AS numbers.

`()` This expression matches an AS Path with a length of 0. The *null AS Path* represents all BGP routes native to your local Autonomous System.

**FIGURE 1.6** An AS Path sample network map

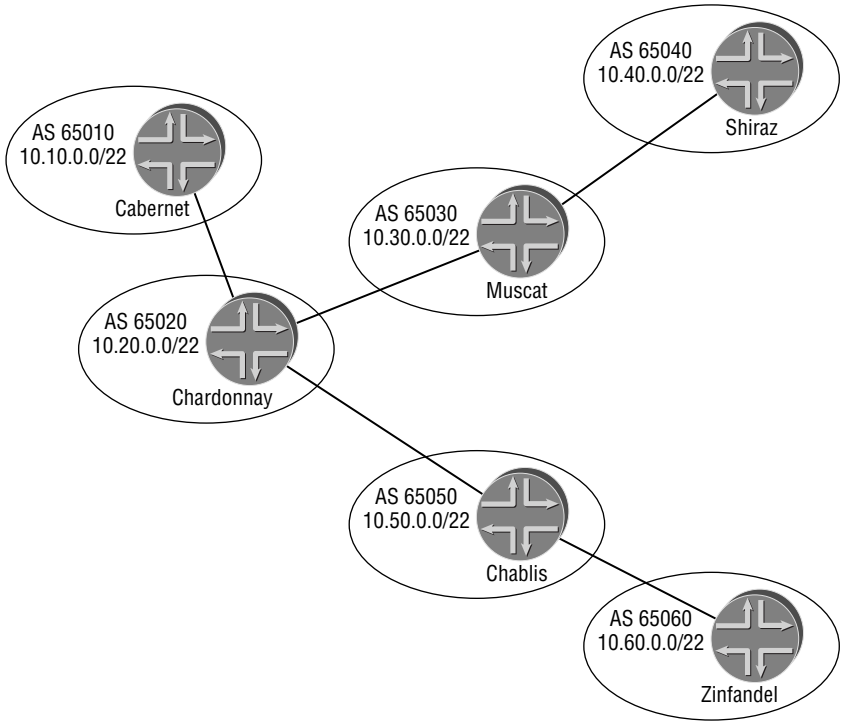



Figure 1.6 shows several Autonomous Systems connected via EBGP peering sessions. Each router is generating customer routes within their assigned address space. The Cabernet router in AS 65010 uses the `aspath-regex` option of the `show route` command to locate routes using regular expressions.

The routes originated by the Zinfandel router in AS 65060 include:

```
user@Cabernet> show route terse aspath-regex ".* 65060"
```

```
inet.0: 27 destinations, 27 routes (27 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
* 10.60.1.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
* 10.60.2.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
* 10.60.3.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
```

The routes originating in either AS 65040 or AS 65060 include:

```
user@Cabernet> show route terse aspath-regex ".* (65040|65060)"

inet.0: 27 destinations, 27 routes (27 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
* 10.40.1.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
* 10.40.2.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
* 10.40.3.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
* 10.60.1.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
* 10.60.2.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
* 10.60.3.0/24     B 170       100           >10.100.10.6   65020 65050 65060 I
```

The routes using AS 65030 as a transit network include:

```
user@Cabernet> show route terse aspath-regex ".* 65030 .+"

inet.0: 27 destinations, 27 routes (27 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
* 10.40.1.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
* 10.40.2.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
* 10.40.3.0/24     B 170       100           >10.100.10.6   65020 65030 65040 I
```

## Locating Routes

An AS Path regular expression is used within a routing policy as a match criterion to locate routes of interest. Much as you saw with communities in the "Match Criteria Usage" section earlier, you associate an expression with a name in the `[edit policy-options]` configuration hierarchy. You then use this name in the `from` section of the policy to locate your routes.

The administrators of AS 65010 would like to reject all routes originating in AS 65030. An AS Path regular expression called ***orig-in-65030*** is created and referenced in a policy called ***reject-AS65030***. The routing policy is then applied as an import policy on the Cabernet router. The relevant portions of the configuration are:

```
[edit]
user@Cabernet# show protocols bgp
export adv-statics;
group Ext-AS65020 {
    type external;
    import reject-AS65030;
    peer-as 65020;
    neighbor 10.100.10.6;
}

[edit]
user@Cabernet# show policy-options
policy-statement adv-statics {
    from protocol static;
    then accept;
}
policy-statement reject-AS65030 {
    term find-routes {
        from as-path orig-in-65030;
        then reject;
    }
}
as-path orig-in-65030 ".* 65030";
```

The Muscat router in AS 65030 is advertising the 10.30.0.0 /22 address space. After committing the configuration on Cabernet, we check for those routes in the `inet.0` routing table:

```
user@Cabernet> show route protocol bgp 10.30.0/22

inet.0: 27 destinations, 27 routes (24 active, 0 holddown, 3 hidden)

user@Cabernet>
```

No routes in that address range are present in the routing table. Additionally, we see that Cabernet has three hidden routes, which indicates a successful rejection of incoming routes. We verify that the hidden routes are in fact from the Muscat router:

```
user@Cabernet> show route hidden terse
```

```
inet.0: 27 destinations, 27 routes (24 active, 0 holddown, 3 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
  10.30.1.0/24     B           100           >10.100.10.6   65020 65030 I
  10.30.2.0/24     B           100           >10.100.10.6   65020 65030 I
  10.30.3.0/24     B           100           >10.100.10.6   65020 65030 I
```

The Cabernet router also wants to reject routes originating in AS 65040 and the Shiraz router. A new AS Path expression called ***orig-in-65040*** is created and added to the current import routing policy:

```
user@Cabernet> show configuration policy-options
policy-statement adv-statics {
    from protocol static;
    then accept;
}
policy-statement reject-AS65030 {
    term find-routes {
        from as-path [ orig-in-65030 orig-in-65040 ];
        then reject;
    }
}
as-path orig-in-65040 ".* 65040";
as-path orig-in-65030 ".* 65030";
```

The router interprets the configuration of multiple expressions in the ***reject-AS65030*** policy as a logical OR operation. This locates routes originating in either AS 65030 or AS 65040. We verify the effectiveness of the policy on the Cabernet router:

```
user@Cabernet> show route 10.30.0/22

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)

user@Cabernet> show route 10.40.0/22

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)

user@Cabernet> show route hidden terse

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
  10.30.1.0/24     B           100           >10.100.10.6   65020 65030 I
  10.30.2.0/24     B           100           >10.100.10.6   65020 65030 I
  10.30.3.0/24     B           100           >10.100.10.6   65020 65030 I
  10.40.1.0/24     B           100           >10.100.10.6   65020 65030 65040 I
  10.40.2.0/24     B           100           >10.100.10.6   65020 65030 65040 I
  10.40.3.0/24     B           100           >10.100.10.6   65020 65030 65040 I
```

Once again, it appears we've successfully used an expression to locate and reject advertised BGP routes. Should the administrators in AS 65010 continue this process, they can reject routes from multiple regular expressions. A potential configuration readability issue does arise, however, when multiple expressions are referenced in a policy. The output of the router begins to wrap after reaching the edge of your terminal screen, and reading a policy configuration might become more difficult. To alleviate this potential issue, the JUNOS software allows you to group expressions together into an *AS Path group*.

An AS Path group is simply a named entity in the `[edit policy-options]` hierarchy within which you configure regular expressions. The Cabernet router has configured a group called ***from-65030-or-65040***. Its configuration looks like this:

```
[edit]
user@Cabernet# show policy-options | find group
as-path-group from-65030-or-65040 {
    as-path from-65030 ".* 65030";
    as-path from-65040 ".* 65040";
}
```

The group currently contains two expressions—***from-65030*** and ***from-65040***—which locate routes originating in each respective AS. The router combines each expression in the AS Path group together using a logical OR operation. In this fashion, it is identical to referencing each expression separately in a policy term. The group is used in a routing policy term to locate routes, and its configuration is similar to a normal regular expression:

```
[edit]
user@Cabernet# show policy-options
policy-statement adv-statics {
    from protocol static;
    then accept;
}
policy-statement reject-AS65030 {
    term find-routes {
        from as-path [ orig-in-65030 orig-in-65040 ];
        then reject;
    }
```

```
}
policy-statement reject-65030-or-65040 {
    term find-routes {
        from as-path-group from-65030-or-65040;
        then reject;
    }
}
as-path orig-in-65040 ".* 65040";
as-path orig-in-65030 ".* 65030";
as-path-group from-65030-or-65040 {
    as-path from-65030 ".* 65030";
    as-path from-65040 ".* 65040";
}
```

After replacing the current BGP import policy with the ***reject-65030-or-65040*** policy, we find that the same routes are rejected on the Cabernet router:

```
user@Cabernet> show configuration protocols bgp
export adv-statics;
group Ext-AS65020 {
    type external;
    import reject-65030-or-65040;
    peer-as 65020;
    neighbor 10.100.10.6;
}

user@Cabernet> show route 10.30.0/22

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)

user@Cabernet> show route 10.40.0/22

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)

user@Cabernet> show route hidden terse

inet.0: 27 destinations, 27 routes (21 active, 0 holddown, 6 hidden)
+ = Active Route, - = Last Active, * = Both

A Destination      P Prf  Metric 1  Metric 2  Next hop      AS path
  10.30.1.0/24     B           100           >10.100.10.6   65020 65030 I
  10.30.2.0/24     B           100           >10.100.10.6   65020 65030 I
```

```
10.30.3.0/24      B          100          >10.100.10.6    65020 65030 I
10.40.1.0/24      B          100          >10.100.10.6    65020 65030 65040 I
10.40.2.0/24      B          100          >10.100.10.6    65020 65030 65040 I
10.40.3.0/24      B          100          >10.100.10.6    65020 65030 65040 I
```

# Summary

In this chapter, you saw how the JUNOS software provides multiple methods for processing routing policies. We explored policy chains in depth and discovered how a policy subroutine works. We then looked at how to advertise a set of routes using a prefix list. Finally, we discussed the concept of a policy expression using logical Boolean operators. This complex system allows you the ultimate flexibility in constructing and advertising routes.

We concluded our chapter with a discussion of two BGP attributes, communities and AS Paths, and some methods of interacting with those attributes with routing policies. Both attributes are used as match criteria in a policy, and community values are altered as a policy action. Regular expressions are an integral part of locating routes, and we examined the construction of these expressions with respect to both communities and AS Paths.

# Exam Essentials

**Be able to identify the default processing of a policy chain.** Multiple polices can be applied to a particular protocol to form a policy chain. The router evaluates the chain in a left-to-right fashion until a terminating action is reached. The protocol's default policy is always implicitly evaluated at the end of each chain.

**Know how to evaluate a policy subroutine.** A common policy configuration is referenced from within another policy as a match criterion. The router processes the subroutine and the protocol's default policy to determine a true or false result. This result is returned to the original policy where a true result is a match and a false result is not a match for the term.

**Understand the logical evaluation of a policy expression.** Logical Boolean operations of AND, OR, and NOT are used to combine multiple policies. Each expression occupies one space in a policy chain. The router first evaluates the expression to determine a true or false result and then uses that result to take various actions.

**Know how to evaluate a prefix list.** A prefix list is a set of routes which is applied to a routing policy as a match criterion. The prefix list is evaluated as a series of exact route matches.

**Be able to construct a community regular expression.** A regular expression is a pattern-matching system consisting of a term and an operator. The term for a community is a single

character, which can be combined with an operator. An expression is used to locate routes as a match criterion in a policy and to modify the list of communities attached to a BGP route.

**Be able to construct an AS Path regular expression.** Regular expressions can also be built to locate routes using the BGP AS Path attribute. The term for an AS Path expression is an entire AS number, not an individual character. The expression is used as a routing policy match criterion either by itself or within an AS Path group.

# Review Questions

1. Which policy is always evaluated last in a policy chain?
   - **A.** The first configured import policy
   - **B.** The last configured import policy
   - **C.** The first configured export policy
   - **D.** The last configured export policy
   - **E.** The protocol default policy

2. What is a possible result of evaluating ***called-policy*** when the router encounters a configuration of `from policy called-policy`?
   - **A.** The route is accepted by ***called-policy***.
   - **B.** The route is rejected by ***called-policy***.
   - **C.** A true or false result is returned to the original policy.
   - **D.** Nothing occurs by this evaluation.

3. The policy called ***outer-policy*** is applied as an export policy to BGP. What happens to the 10.10.10.0 /24 static route when it is evaluated by this policy?

```
outer-policy {
    term find-routes {
        from policy inner-policy;
        then accept;
    }
    term reject-all-else {
        then reject;
    }
}
inner-policy {
    term find-routes {
        from protocol static;
        then reject;
    }
}
```

   - **A.** It is accepted by ***outer-policy***.
   - **B.** It is rejected by ***outer-policy***.
   - **C.** It is accepted by ***inner-policy***.
   - **D.** It is rejected by ***inner-policy***.

4. Which route filter match type is assumed when a policy evaluates a prefix list?

   A. `exact`

   B. `longer`

   C. `orlonger`

   D. `upto`

5. The policy expression of `(policy-1 && policy-2)` is applied as an export within BGP. Given the following policies, what happens when the local router attempts to advertise the 172.16.1.0 /24 BGP route?

```
policy-1 {
    term accept-routes {
        from {
            route-filter 172.16.1.0/24 exact;
        }
        then accept;
    }
}
policy-2 {
    term reject-routes {
        from {
            route-filter 172.16.1.0/24 exact;
        }
        then reject;
    }
}
```

   A. It is accepted by ***`policy-1`***.

   B. It is rejected by ***`policy-2`***.

   C. It is accepted by the BGP default policy.

   D. It is rejected by the BGP default policy.

6. The policy expression of `(policy-1 || policy-2)` is applied as an export within BGP. Given the following policies, what happens when the local router attempts to advertise the 172.16.1.0 /24 BGP route?

```
policy-1 {
    term accept-routes {
        from {
            route-filter 172.16.1.0/24 exact;
        }
        then accept;
    }
```

```
}
policy-2 {
    term reject-routes {
        from {
            route-filter 172.16.1.0/24 exact;
        }
        then reject;
    }
}
```

**A.** It is accepted by ***policy-1***.

**B.** It is rejected by ***policy-2***.

**C.** It is accepted by the BGP default policy.

**D.** It is rejected by the BGP default policy.

**7.** The regular expression `^6[45][5-9]..:.{2,4}$` matches which community value(s)?

**A.** 6455:123

**B.** 64512:1234

**C.** 64512:12345

**D.** 65536:1234

**8.** The regular expression `^*:2+345?$` matches which community value(s)?

**A.** 65000:12345

**B.** 65010:2234

**C.** 65020:22345

**D.** 65030:23455

**9.** The regular expression `64512 .+` matches which AS Path?

**A.** Null AS Path

**B.** 64512

**C.** 64512 64567

**D.** 64512 64567 65000

**10.** The regular expression `64512 .*` matches which AS Path?

**A.** Null AS Path

**B.** 64512

**C.** 64513 64512

**D.** 65000 64512 64567

# Answers to Review Questions

1. E. The default policy for a specific protocol is always evaluated last in a policy chain.

2. C. The evaluation of a policy subroutine only returns a true or false result to the calling policy. A route is never accepted or rejected by a subroutine policy.

3. B. The policy subroutine returns a false result to ***outer-policy*** for the 10.10.10.0 /24 static route. The ***find-routes*** term in that policy then doesn't have a match, so the route is evaluated by the ***reject-all-else*** term. This term matches all routes and rejects them. This is where the route is actually rejected.

4. A. A routing policy always assumes a match type of `exact` when it is evaluating a prefix list as a match criterion.

5. B. The result of ***policy-1*** is true, but the result of ***policy-2*** is false. This makes the entire expression false, and ***policy-2*** guaranteed its result. Therefore, the action of `then reject` in ***policy-2*** is applied to the route and it is rejected.

6. A. The result of ***policy-1*** is true, which makes the entire expression true. Because ***policy-1*** guaranteed its result, the action of `then accept` in ***policy-1*** is applied to the route and it is accepted.

7. B. The first portion of the expression requires a five-digit AS value to be present. Option A doesn't fit that criterion. While Option D does, it is an invalid AS number for a community. The second portion of the expression requires a value between two and four digits long. Of the remaining choices, only Option B fits that requirement.

8. B and C. The first portion of the expression can be any AS value, so all options are still valid at this point. The second portion of the expression requires that it begin with one or more instances of the value 2. Option A begins with 1, so it is not correct. Following that must be 3 and 4, which each of the remaining options have. The final term requires a value of 5 to be present zero or one times. Options B and C fit this requirement, but Option D has two instances of the value 5. Therefore, only Options B and C are valid.

9. C. The expression requires an AS Path length of at least 2, which eliminates Options A and B. The second AS in the path may be repeated further, but a new AS number is not allowed. Option D lists two different AS values after 64512, so it does not match the expression. Only Option C fits all requirements of the regex.

10. B. The expression requires an AS Path length of at least 1, which must be 64512. Other AS values may or may not appear after 64512. Only Option B fits this criterion.