From: Eliot Lear <lear@NET.BIO.NET>

The following was written by Dr. Charles Hedrick of Rutgers University sometime in 1985. Please read it with the understanding that rule numbers are nothing more than function names. For further reference, I suggest the Sun Tutorial on Sendmail in their manuals.
-eliot

To: steve@jplgodo.UUCP
Subject: a brief tutorial on sendmail rules

A previous message suggested using "sendmail -bt" to see how sendmail is going to process an address. This is indeed a handy command for testing how an address will be processed. However the instructions given were not quite right. To see how sendmail is going to deliver mail to a given address, a reasonable thing to type is

```
        sendmail -bt
        0,4 address
```

Even this isn't quite right, but with "normal" rule sets it should work.

Because there is so much confusion about sendmail rules, the rest of this message contains a brief tutorial. My own opinion of sendmail is that it is quite a good piece of work. Many people have complained about the difficulty of understanding sendmail rule sets. However I have also worked with mailers that code address processing directly into the program. I much prefer sendmail. The real problem is not with sendmail, but with the rules. The rules normally shipped from Berkeley have lots of code that does strange Berkeley-specific things, and they are not commented. Also, typical complex rule sets are trying to handle lots of things, forwarding mail among several different mail systems with incompatible addressing conventions. A rule set to handle just old-style (non-domain) UUCP mail would be very simple and easy to understand. But real rule sets are not doing simple things, so they are not simple.

For those not familiar with sendmail, -bt invokes the rule tester. It lets you type a set of rule numbers and an address, and then shows you what the rules will do to that address. In addition, rule test mode automatically applies rule 3 before whatever rule you ask it to apply. As we will see shortly, this is a reasonable thing to do.

Before describing the rule sets, let me define two terms: "header" and "envelope". Header refers to the lines at the beginning of the message, starting with "from:", "to:", "subject:", etc. Sendmail does process these lines. E.g. with uucp mail it will add its own host name at the beginning of the from line, so that the final recipient stands some change of replying to the message. However sendmail normally does not depend upon the from and to lines to perform its actual delivery. It has more direct knowledge, passed on to it from the program that generated the mail, or if it came from another site, the mailer at that site. This information is referred to as the "envelope", since it is like the addresses on the outside of an envelope. For Arpanet mail, the envelope is passed to the next site by the MAIL FROM: and RCPT TO: commands. For UUCP mail, it is passed on as arguments to the remote rmail command. To see why there have to be separate addresses "on the envelope", consider what happens when you send mail to "john@vax, mary@sun". Two copies of the message will be dispatched, one to vax and the other to sun. The "to: " line in the headers will show both addresses. However the envelope will show only the right address that we want this copy to go to. The copy sent to vax will show "john@vax" and the copy sent to sun will show "mary@sun". If sendmail had to look at the "to: " line, it would never know which of the addresses shown there it was responsible for handling.

Anyway, here is what the rules do:

3: always done first. This turns addresses from their normal textual form into a form that the rest of the rules understand. In most cases, all it does it put < > around the name of the host that is next in line. Thus foo@bar turns into foo<@bar>. However it also does a few transformations. E.g. it turns foo!bar!user into bar!user<@foo.UUCP>. Since sendmail accepts either ! syntax or @....UUCP syntax, rule 3 standardizes on @ syntax. It also does a few other minor things. But you won't be far off if you just think of it as adding < > around the host name.

4: always done last. This turns addresses from internal form back into external form. It removes the < > around the host name, and turns foo@bar.UUCP back into bar!foo. Again, there are one or two other minor things, but you won't be too far off if you think of 4 as just removing the < > around the host name.

0: This is the rule that handles the destination address on the envelope. It is in some sense the primary rule. It returns a triple: protocol, host, user. The protocol is usually one of local, TCP, or UUCP. At the moment, it figures this out syntactically. In our rule set, hosts ending in .UUCP are handled by UUCP, the current host is local, and everything else is TCP. As domains are integrated into UUCP, obviously this rule is going to change. This rule does very little other than simply look at the format of the host name, though as usual a few other details are involved (e.g. it removes the local host. So myhost!foo!bar will be sent directly to foo).

1 and 2 are protocol-independent transformations used for sender and recipient lines in the header (i.e. from: and to: lines). In our rule sets, they don't do anything.

Each protocol has its own rules to use for sender and recipient lines in the header. E.g. UUCP rules might add the local host name to the beginning of the from line and remove it from the to line. In our rule set, the complexities in these rules are primarily caused by forwarding between UUCP and TCP. The line that defines the mailer for a protocol lists the rule to use for source and recipient, in the S= and R=.

Finally, here is the exact sequence in which these rules are used. For example, the first line means that the destination specified in the envelope is processed first by rule 3, then rule 0, then rule 4.

```
envelope recipient:    3,0,4  [actually rule 4 is applied only to the
                               user name portion of what rule 0 returns]
envelope sender:       3,1,4
header recipient:      3,2,xx,4  [xx is the rule number specified in R=]
header sender:         3,1,xx,4  [xx is the rule number specified in S=]
```

I have the impression that the sender from the envelope (the return-path) may actually get processed twice, once by 3,1,4 and the second time by 3,1,xx,4. However I'm not sure about that.

Now for the format of the rules themselves. I'm just going to show some examples, since sendmail comes with a reference manual, which you can refer to. However these examples are probably enough to let you understand any set of rules that makes sense in the first place (which the normal rules do not). This example is from our UUCP definition. It a simplified version of the set of rules used to process the sender specification. As such, the major thing it has to do is to add our host name to the beginning, so that the guy at the end will know that the mail went through us.

```
S13
R$+<@$-.UUCP>           $2!$1                           u@host.UUCP => host!u
R$=U!$+                 $2                              strip local name
R$+                     $:$U!$1                         stick on our host name
```

Briefly, the first rule turns the address from the form foo<@bar.UUCP> back into bar!foo. The second rule removes our local host name, if it happens to be there already, so we don't get it twice. The third rule adds our host name to the beginning.

S13 says that this is the beginning of a new rule set, number 13.

```
R$+<@$-.UUCP>           $2!$1                           u@host.UUCP => host!u
```

R says that this is a rule. The thing immediately after it, $+<@$-.UUCP> is a pattern. If this pattern matches the address, then the rule "triggers". If the rule triggers, the address is replaced with the "right hand side", i.e. what is after the tab(s). In this rule, the right hand sie is $2!$1. The thing after the next tab(s) is a comment. This rule is used in processing UUCP addresses. As noted above, by the time we get to it, rule 3 has already been applied. So if we had a UUCP address of the form host1!host2!user, it would now be in the form host2!user<@host1.UUCP>. This does match the pattern:

```
        $+        <@$-   .UUCP>
        host2!user<@host1.UUCP>
```

$+ and $- are "wildcards" that match anything. $- will match exactly one word, while $+ will match any number. (By the way, with the increasing use of domains, this production should probably use $+.UUCP, not $-.UUCP.) Since the pattern matches, we replace this with the "right hand side" of the rule, $2!$1. $ followed by a digit means the Nth thing matched by a wildcard. In this case there were two wildcards, so

```
   $1 = host2!user
   $2 = host1
```

The final result is

```
   host1!host2!user
```

As you can see, we have simply turned UUCP addresses from the format produced by rule 3 back into normal ! format.

The second rule is

```
R$=U!$+                 $2                              strip local name
```

This is needed because there are situations in which our host name ends up on the beginning of the recipient address. Since we are about to add our host name, we don't want it to be there twice. So if it was there before, we remove it. $= is used to see if something is a member of a specified "class". U happens to be a list of our UUCP host name and any nicknames. So $=U!$+ matches any address that begins with our host name or nickname, then !, then anything else. Suppose we had topaz!host1!host2!user. The match would be

```
$=U  !$+
topaz!host1!host2!user
```

The result of the match is that

```
$1 = topaz
$2 = host1!host2!user
```

Since the right hand side of this rule is simply "$2", the result is

```
host1!host2!user
```

I.e. we have removed the topaz from the beginning. By the way, the class U used by the rule would have been defined earlier in the file by the statement

```
CUtopaz ru-topaz
```

C defines a class. U is the name of the class. The rest of the line is the list of things that will be in the class.

Finally we have the rule

```
R$+                    $:$U!$1                        stick on our host name
```

The $+ matches anything. In this case the name is host1!host2!user, so the result of the match is

```
   $1 = host1!host2!user
```

The result looks slightly obscure. $: is a tag that says to do this only once. The problem is that this rule always applies, since the pattern matches anything. Normally, rules are applied over and over, as long as they apply. In this case, the result would be an infinite loop. Putting $: at the beginning says to do it only once. $U says to use the value of the macro U. Earlier in the file we defined U as our UUCP host name, with a definition

```
DUtopaz
```

Note that there can be a class and a macro with the same name. $=U tests whether something is in the class U. $U is replaced by the value of the macro U.

So the final value of this rule, $:$U!$1, is

```
topaz!host1!host2!user
```

So this rule has managed to add our host name to the beginning, as it was supposed to. Since there are no further rules in the set (the next line is the end of file or the beginning of a new rule set), this value is returned.

There are several more magic things that can appear in a pattern. The most important are:

$* - this is another wild card. It is similar to $+, but $+ matches anything, whereas $* matches both anything and nothing. I.e. $+

matches 1 or more tokens and $* matches 0 or more tokens. So here is a list of the wildcards I have mentioned:

```
   $*   0 or more
   $+   1 or more
   $-   exactly 1
   $=x  any member of class x
```

A typical example of $* is a production where we aren't sure whether the user name is before or after the host name:

```
R$*<@$+.UUCP>$*         $@$1<@$2.UUCP>$3
```

This production would test for the host name ending in .UUCP, and return immediately. $@ is a flag you haven't seen yet. It is simply a return statement. It causes the right hand side of this rule to be returned as the final value of this rule set.

The other magic thing I will mention is $>. This is a subroutine call. Here is an example taken from rule set 24, which is used to process recipients in TCP mail. Its purpose is to handle the situation where we might have an address like topaz!user@red. (Our host name is topaz. Red is a local host that we talk to via TCP.) I.e. someone is asking us to relay mail to red. Rule 3 will have turned this into user@red<@topaz.UUCP>. What we want to do is get rid of the topaz.UUCP and treat red as the host. (Rule set 0 would do this for the recipient on the envelope. This rule is used for the to: field in the header.) Here is the rule.

```
R$+<@$=U.UUCP>          $@$>9$1                         in case local!a@b
```

The pattern matches our example, as follows:

```
   $+      <@$=U  .UUCP>
   user@red<@topaz.UUCP>
```

Recall that $+ matches anything and $=U tests whether something is our UUCP host name or one of our nicknames. The result of the match is

```
   $1 = user@red
   $2 = topaz
```

The right hand side is $@$>9$1. The $@ is the tag saying to stop the rule set here and return this value. $>9 is a subroutine call. It says to take the right hand side, pass it to rule set 9, and then use the value of rule set 9. The actual right hand side is simply $1, which in this case is user@red. Here is rule set 9:

```
S9
R$*<$*>$*               $1$2$3                          defocus
R$+                     $:$>3$1                         make canonical
R$+                     $@$>24$1                        and do 24 again
```

The first rule simply removes < >. It is sort of a quick and dirty version of rule 4. In fact we have no < > left, since we have removed the <@topaz.UUCP>. So this rule does not trigger. (Now that I think about it, I suspect it is probably never going to trigger, and so is not needed.)

The next rule is a simple subroutine call. It matches anything ($+ matches any 1 or more token). The right hand side is $:$>3$1 The $:

says to do it only once. Since the rule matches anything, you need this, or you will have an infinite loop. The $>3 says to call rule 3 as a subroutine. The $1 is the actual right hand side. Since the left hand side matched the whole address, what this rule does is simply call rule set 3 on the whole address. Recall that rule set 3 basically locates the host name and puts < > around it. So in this case the result is user<@red>. As you can see, it was not enough to remove <@topaz.UUCP>. That leaves us with no host name. We have to call rule 3 to find the current host name and put < > around it.

The last rule is really just a goto statement. The pattern is $+, which matches anything, so it always triggers. The right hand side is $@$>24$1. The $@ is the return tag. It says to stop this rule set and return that value. $>24 says to call rule set 24. The actual right hand side is $1, so we call rule set 24 with the whole address. If you recall, this ruleset (9) was called from the middle of 24 when we found user@red<@topaz.UUCP>. So what we have done is to change this into user<@red> and say to start rule set 24 over again.

I hope you have found this exposition useful. As a final convenience, here is a "reference card" for reading rule sets. Note that this contains only operators used by the rules. There are plenty of other facilities used in the configuration section which I am not documenting here. (I'd love to see someone produce a complete reference card.)

```
wildcards:
   $*   0 or more tokens
   $+   1 or more tokens
   $-   exactly one token
   $=x  member of class x (x must be a letter, lower/upper case distinct)
   $~x  not a member of class x

macro values (usable in pattern or on right hand side)
   $x   value of macro x (x must be a letter, lower/upper case distinct)
        At least on the Pyramid, $x is replaced by the macro's value
        when the sendmail.cf file is being read in.

on the right hand side:
   $n   string matched by the Nth wildcard
   $>n  call rule set N as a subroutine
   $@   return
   $:   only do this rule once

in rule 0, defining the return value
   $#   protocol
   $@   host
   $:   user

Rutgers extensions, usable only on right hand side
   $%n  take the string matched by the Nth wildcard, look it up in
        /etc/hosts, and if found use the primary host name
   $&x  use the current value of macro x.  x must be a letter.
        upper and lower case are treated as distinct.
```