HickUP

by Daniel Bergström

System Administrator Guide

Release 1.1

HickUP: System Administrator Guide

by Daniel Bergström

Abstract

A system administrators guide to writing application definistions to HickUP.

Table of Contents

Chapter 1

Introduction

Hick

1.1. Terminology

Type
: A type identifies the environment that HickUP will do the setup for. This is usally the OS platform, see also Section 3.1, “Types”.

Packet
: A definition of a application or setting that has one or more versions of which a user can choose to install. How to write a packet definition is described in Chapter 3, *Packets*.

Chapter 2

Installing

HickUP is installed with two parts, one which can be shared between all hosts and one for the system depending binaries. Depending on where you put the base directory it will have the same structure under the shared cataloge. Below is an exampe on how this can look like, it is a setup for solaris and FreeBSD, see Figure 2.1.

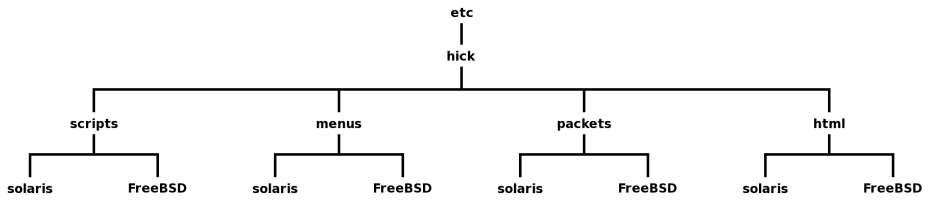

Figure 2.1. Directories for shared hick files, showing for the types solaris and FreeBSD

This directory structure will be created when installing but the OS specifict parts, like the directories solaris and FreeBSD, since it is up to you to choose how you want the HICKTYPE part to work. Examples on scripts, menus, html and info files will also be installed. As of the example package 'hickup' you willl need to copy it into the /etc/hick/packets/$HICKTYPE.

Chapter 3

Packets

In this chapter we will go through how to create a packet. A packet contains information about how to setup program, each packet have one or more versions. It is the versions which will be choosen when installing a packet. Some information can be commonly shared in the versions and can then be written outside the version. The packet also contains information on how to display the packet. It is not possible to have a packet with the same name twice, this will result in a unspecified behaviour. All names are case insensitive.

String

A string has no set length and depending on wheter or not the enclosed within quotes, "", or not. When the string is not enclosed within quotes only a word may be used, that is no spaces may occur. When the string is enclosed within quots there are special characters that needs to be considered. If you wish to include a "" or '\' in the string you must escape them with a slash '\' infront of them. It is also possible to add a new line by escaping 'n'. Table 3.1 shows a number of example strings.

For an empty string use two quotes after each other, for example "".

String	**Result**
Hello	Hello
"Hello World!"	Hello World!
"/\\ \"ALERT!\" /\\"	/\ "ALERT!" /\
""	*empty*

Table 3.1. Example of how strings are encoded

Number

A number can contain up to 9 digits and may not be preceeded with a plus or minus sign.

Group — A group is a string with a limit of 32 characters that defines where a packet definition belongs to. A packets can belong to multiple groups.

Packet — A packet is a string with a limit of 32 characters that defines the name of the packet definition.

Version — The version is a string with a limit of 16 characters.

Date — A date is set by specifying seconds since 00:00:00 UTC, January 1, 1970 (see time(2)) or by using the time and date format:

```
DD '/' MM '/' YYYY [ ',' hh ':' mm [ '.' ss ] ]
```

No spaces are allowed between the numbers and the separators. You must also add leading zeros when needed. An example on how time and date can be writte:

```
27/07/1999
27/07/1999,11:03
27/07/1999,11:03.11
```

Variable — A variable is a setting in the packet or version definition that tells hick how to install the package. The variables available are; call, conflict, create, env, html, info, link, menuItem, require, script and xres. For more information see Section 3.3, “Variables”.

3.1. Types

Packets is stored in directories below the packets directory. The directory names is the same as the Type names. User which does not have access to a Type directory will not be shown the Type. There is one directory which is shared between all the different Types, it is called common. For every Type read will the packets in the common directory be appended.

A hicktype could be set to the $OSTYPE, this would for different platforms give a unique value for that type. Setting $HICKTYPE to $OSTYPE is the normal use, only leaving it open for your needs on different Types.

3.2. Packet and Version

The packet can be written in any order, the order will be remembered to when its displayed for the user. You do not have to use all the settings in a packet to make it work, the minimum requirement for a packet is that it should contain a version declaration. A line beginning with a '#' will be ignored.

3.2.1. Packet

Packets are the base for everything in hick, they contain the basic information, and also at times the major information, depending on what the intentions with it is. But a packet doesn’t become anything without having atleast one version, see Section 3.2.13, “Version”. You can have variables (see Section 3.3, “Variables”) in a packet will those be added to the selected version.

```
[ GROUP [ '|' GROUP ... ] PACKET '{'
'}'
```

Between the braces '{' and '}' can you use any variable or packet options as about, created, modified, expire, subgroup, multiple hidden required, force, allow or deny, as well as versions. Below is an example on how the hickup packet looks like:

```
Miscellaneous HickUP {
  about "Hick User Profile - Utility for simple dotfile management."

  required
  created 19/01/2000

  version def 1.1a3 {
    created 19/01/2000
    link {
      bin /local/fw/hick/1.1a3/bin/hickup
      bin /local/fw/hick/1.1a3/bin/hickprj
      bin /local/fw/hick/1.1a3/bin/hickcmd
    }

    menuItem Applications HickUP... hickup
    html "<a href=\"http://www.hickup.org/index.html\">HickUp.org" Applications
  }
}
```

3.2.2. About

With the about you can give the user a short description about the program, and helpful tips on how they can start or use it. It can be override or extended by a version about message see Section 3.2.14, “AddAbout”. This is a packet specific setting.

```
about STRING
```

3.2.3. Created

Sets the date when the packet or version was created. This to make it easy for the users to see what new packets or versions have been created since the last time they checked.

```
created DATE
```

3.2.4. Modified

If you modified a already created packet or version you should set the modified settings. This so that hick knows that it needs to update this packet or version, which will not happen else. If you change only version settings you only should only then change the modified date in the version. If you change some setting in the packet, this excludes adding a new version, should you also change it there.

```
modified DATE
```

3.2.5. Expire

To notify the users that you will expire a packet or version, you set this to the date when you want to expire the packet or version. After the expire date will not it be possible to install the packet any more. Only to remove, and hick will at the next update then also automatically remove any packets which have expired. After the expire date you only need to keep information such as call scripts, since it is then possible to run that script to remove any files created with a call script see Section 3.3.1, “Call”.

```
expire DATE
```

3.2.6. SubGroup

Extends to the main groups subgroups so that you not only can have one level of group, but as many as you like. A packet can be as many subgroup as you which for each place in a subgroup you need to write a subgroup declaration. When you write subgroups it will then create the groups which does not exist. This is a packet specific setting.

```
subGroup GROUP GROUP [ '&' GROUP ... ]
```

The first GROUP is to which main GROUP it belongs, this GROUP must been defined at the packet declaration. Then following group is the new subgroup to create, any groups followed after that, separated with an '&' will add another level to the subgroups. Below is some examples on subgroup declarations:

```
subGroup Games Arcade
```

This will put the packet in the Arcade group below the Games group. To add another level to this you just write a third group name.

```
subGroup Games Arcade & Fun
```

This will then place the packet in the Fun group below the arcade and games group.

If a packet belongs to multiple groups, will only subGroup affect the named group. If you want to affect all groups you can just write "" at the first group. This will tell hick that it should add the settings for all groups the packet belongs to. If we have a packet belonging to Graphics and Viewers and we which to but this packet in a subGroup with the same name in both of these groups you can do it in two ways:

```
subGroup Graphics Blob
subGroup Viewers Blob
```

or you can just write:

```
subGroup "" Blob
```

3.2.7. Multiple

By default is it only allowed to select and install one version in each packet. If you want to allow the users to install more then one version of the packet you will need to set this option. Make sure that the packet does not have any internal conflicts. This can be used when you want to give the user the possibility to set settings for different programs. This is a packet specific setting.

```
multiple
```

3.2.8. Hidden

Sometimes it can be good to hide packets for all users, this can be achieved with this flag. Another possible solution is to deny all users. An example of the use it is, if you have a packet which is required, as a system setup for all users. The packet is not a choice of the user it should always be setuped, then it could be an idea to hide it. Note that if you set is on hide you will not need to select is in the project specifications to allow it, it will automaticly be installed, this can not be achieved in the same way with the deny. To hide the packet you write in the packet:

```
hidden
```

3.2.9. Required

If you must have a version of a packet, and you at the same time want to let the user choose which version you can use the require option. It works like if the user have not selected any version it will automaticlly choose the default version, if any, or take the first version specified in the packet. To make a packet required you write in the packet:

```
required
```

3.2.10. Force

At times you which some users to automatically get a certain packet installed, without letting them choose. The force setting can set so that all users or some user or group can get a packet version installed. When forcing a version will all other chosen versions be replaced with the named version. You can declare as many force as you want in a packet. It is imporent that the version which is going to be forced exists. This is a packet specific setting.

```
force VERSION
force user | group STRING VERSION
force user | group '{' [ STRING ... ] '}' VERSION
```

A few examples on how you can write a force. If having a packet which everyone should be using, you can just write it like:

```
force 1.0
```

This will then force the version 1.0 in the packet onto all the users. If only a user or group is to be forced you will have to specify whom you inted to force this on. You can give a list of users or groups if you put the names in between '{' and '}'. The user and group information is taken from the Unix system. Example on this:

```
force user { nisse kalle } 1.0
```

forcing version 1.0 onto the users nisse and kalle. If you use a group, will all members of that group get the packet forced. Example on this:

```
force group admin 1.1
```

Forces to all members of the admin group, version 1.1. If setting so a user is in more then one force will the result be unspecified.

3.2.11. Allow

To make some packets available for some users, and hiding from others, the allow can be usefull. You can setup a list of users which can access a specified version in the packet. Users who is not named by name or beloning to named group will not be snown the version. You can have as many allow directions as you wish, and they can be combind with denies, see Section 3.2.12, “Deny”. By settings the version to an empty string "" or { } will that apply for all versions in the packet.

```
allow user | group STRING VERSION
allow user | group STRING { [ VERSION ... ] }
allow user | group { [ STRING .... ] } VERSION
allow user | group { [ STRING ... ] } { [ VERSION ... ] }
```

If you want to allow the users 'daniel' and 'ankan' only to be able to choose the version 1.2 in a packet you can write it like this.

```
allow user { daniel ankan } 1.2
```

Perhaps also the users in the 'stuff' group should be granted access to 1.2 and also be exclusive access to 1.0, that could be written something like this.

```
allow group stuff { 1.2 1.0 }
```

You can combind multiple allows and denies to get the effects you wish for.

3.2.12. Deny

Deny works like allow only instead of excluding all other users then the named, it denies the named users or groups.

```
deny user | group STRING VERSION
deny user | group STRING { [ VERSION ... ] }
deny user | group { [ STRING .... ] } VERSION
deny user | group { [ STRING ... ] } { [ VERSION ... ] }
```

3.2.13. Version

Before a packet is valid you must have atleast one version, else will not the packet be selectabel in hickup. There is no limit on how many version you can have in a packet. A version can contain all variables (see Section 3.3, “Variables”) and created, modified, expire and addAbout. Syntax for a version:

```
version [ def ] VERSION '{'
'}'
```

Below is the version out of the hickup packet, might depend on where you install hickup.

```
version def 1.1a3 {
  created 19/01/2000
  addAbout "\n\nThis is a development version."
  link {
    bin /local/fw/hick/1.1a3/bin/hickup
    bin /local/fw/hick/1.1a3/bin/hickprj
    bin /local/fw/hick/1.1a3/bin/hickcmd
  }
}
```

3.2.14. AddAbout

This allows a version to have additional information to the packets about message. Or it can override the packets about message with the override option. This is a version specific setting.

```
addAbout [ override ] STRING
```

An example on how the addAbout can be used:

```
addAbout "This version adds a new feature."
```

This will add to the packets about message *This version adds a new feature.* for the version the addAbout is specified in. If you want to remove the packets about message when viewing a version you just set the override flag.

3.3. Variables

There is 11 different variables, which can be set both in the version and packet. Those set in the packet is added to the versions once installing. These variables is the information

added to the user settings. You can have as many variables as you need. All variables have the possibility to write values in a list, for example:

```
link bin /usr/bin/ls
link man /usr/man/man1/ls.1
```

these two variables can be written without having the variable name infornt of them for each. By surounding them with a '{' and '}' will allow you to write a list of commands. This could then be written:

```
link {
  bin /usr/bin/ls
  man /usr/man/man1/ls.1
}
```

In the following descriptions of the variables will the list syntax not be written out.

3.3.1. Call

Sometimes the variables is not enough to get a program setuped. With the call variable can you execute a program when running the update. It can be setup when to run the call, so that you can set it only to execute the script at first installation time, or when removed. The syntax for the call variable:

```
call [ install | update | remove [ '&' install | update | remove .... ]
     STRING STRING
```

By default is all flags setl, so that the install, update and remove flags is set. This means it will be executed at occations. The first STRING is the file to execute, remember to always specify full path. And the second STRING is any additional options to the program. Both STRING have to be specified. If no options is wanted, set it to an empty STRING. Hick will add some options to the script such as update method, install, update or remove and packet name and version followed by the additional options. All output of the script will be written to the runlog file. Below follows an example on how the call script works:

```
call install & remove /local/share/install/netscape ""
```

this will run the /local/share/install/netscape script when installing and removing the packet, and not when updating. Updating is when a installed packet have been modified and the modified date is set. If a user now choose the packet 'netscape' and version '4.6' will the following arguments be passed to the script once executed:

```
/local/share/install/netscape install "netscape" "4.6" ""
```

when the user then removes the packet or change to another version will it then run the script with the options:

```
/local/share/install/netscape remove "netscape" "4.6" ""
```

It will also run the script when removing a expired packet, see Section 3.2.5, “Expire”.

3.3.2. Conflict

Programs sometimes conflicts with each other so that if you setup for one the other will not be working correctly. The conflict settings is to let the user know that it can not install both programs and get away with it. You can conflict against a whole packet or specified versions in a packet. A conflict must be both ways so if you from a version conflict with a

whole packet, must there be a conflict back from the whole packet to that single version. In the hickup program will it warn on those occations this is not true. The syntax for the conflict variable:

```
conflict PACKET VERSION
conflict PACKET '{' [ VERSION ... ] '}'
```

To make a conflict to a whole packet you can make an empty string ("") on the version, the packet must be specified. Some examples on conflict:

```
conflict emacs ""
```

This will create a conflict to the whole emacs packet.

```
conflict gcc 1.6.3
```

Causes a conflict to GCC version 1.6.3.

```
conflict gdb { 4.16 4.17 }
```

Marks conflicts with gdb 4.16 and 4.17.

3.3.3. Create

Sometimes it is usefull to just add text to a text file, or just create a shell script from the packet. The create will add or create scripts. It will add to the file in no paticular order from the packets. Also is there a need of caution when creating a file, if overwritting another one. Note that you can use ~/ and full path to the file name, as well as bin/ which will be created out of the base hick dir. When adding and the file is not existing will the file be created. The synrax for the create variable:

```
create [ new | append ] [ MODE ] FILENAME CONTENTS
```

To create a new file, which is suppose to be a shell script you can do like this:

```
create new 755 bin/hello "!#/bin/sh
echo \"Hello World\\!\""
```

This will generate a script in ${HICKROOT}/.hick/${HICKTYPE}/bin/hello which writes out "Hello World!" once executed.

3.3.4. Env

Environment variables can easy be setuped with using this variable. You can append, prepend, set and unset values. The paths are written without use of any separators, this will be taken care of later. The environment variables can be created for various shell scripts and you do not have to think about that more then the name and value if the environment variable. The information where will be passed onto a script language which will generate the acctual script, see chapter 4.1. Below is the syntax for the env variable:

```
env [ prepend | append | set ] NUMBER STRING STRING
env unset NUMBER STRING
```

You can not mix between set, unset or prepend and append.

Prepend and append, prepend will be put before the current environment value and append will be put after the current environment value.

How is the priority levels working, the default priority level is set to 50. 0 - 49 will be placed before the default priority and is there for of a higher priority. Anything higher then 50 will be placed after. This is for all but append which works in a reversed order.

3.3.5. HTML

HTML is mainly to setup a HTML index from where the user can get more help on the program. A HTML page will be created for each hicktype in the users .hick directory called index-$HICKTYPE.html, it is generated with hickmk for more information how to setup the HTML layout see. .

```
html CODE TOPIC
```

The CODE is the HTML code for the item, it can be plain text or containing HTML tags, depending on how the html generating is done.

TOPIC is to what kind of link groups it belongs, this can be a empty string which will then be no group assigned. Grouping the different links can make it easier for the user to find a specific link.

```
html "<a href=\"http://www.hickup.org/index.html\">HickUP.org</a>" "Applications"
```

This will add the HickUP.org link under the applications.

3.3.6. Info

For easy maintaining the GNU info dir file you can be using the info variable. This will then create a personal top page for the installed packages with GNU info.

```
info NAME FILENAME SUBMENU ABOUT TOPIC
```

NAME
: Text which is later selected to enter the info document.

FILENAME
: Info filename to open, dose not need to include .info part.

SUBMENU
: If wanted you can jump info a submenu in the info file. Can be specified with an empty string if no submenu is wanted.

ABOUT
: The about text, this should be written in a single line even if it is very long, it will be formated to fit into the menu.

TOPIC
: A kinda of grouping to make it easier to find the documents. If left as an empty string will is belong to the head topic.

If we take the gdb's info file for example, it will look something like this:

```
Development
* Gdb: (gdb).                The GNU debugger GDB.
```

In a hick packet its written as:

```
info Gdb gdb "" "The GNU debugger GDB." Development
```

3.3.7. Link

This is perhaps the most usefull of the variables, at least what it where thought to use alot, linking of files. You can use it to create a symbolic link to an existing or non existing file, and rename the link if needed.

```
link [ normal | rename ] TO FROM
```

The TO field can be used in three different ways, using a full path, a relative path or user home directory. If a full path is used will the link be placed as where is specifies. The relative path is pointing into the users hicktype specific directory. The user home will begin in the specified users home, both ~/ and ~daniel/ can be used. Some example on this:

```
link bin /local/gnu/emacs/20.4/bin/emacs
```

This creates a link into ~/.hick/$HICKTYPE/bin called emacs pointing to /local/gnu/emacs/20.4/bin/emacs.

```
link ~/daniel-project ~daniel/project
```

This will create a link called daniel-projects in the current user home from daniel's projects directory. Even if the full path links exists it is perhaps not to useful when it comes to linking to.

3.3.8. MenuItem

To make it a bit easier can you create menu items and letting hickup take care of the generating of the menus, see . This is done by setting up the most basic, as which menu does this belong to and item name, and command to execute. You can also send extra options which will be passed to the menu script. The priority is for telling which menu item should be before another, normaly this is at 50. If a number of items are at the same priority the order of those items will be in no particular order.

```
menuItem [ PRIORITY ] MENU ITEM COMMAND [ '{' [ NAME VALUE ... ] '}' ]
```

For adding like hickup into the menus you only need to write:

```
menuItem Applications HickUP... hickup
```

But for more advanced menus, as a whole list of menus, you perhaps want some more options, as separators and such. It depends alot on how the script for each window manager is written how well things are supported. see .

```
menuItem 49 Programs "[separator]" "" { isSeparator 1 }
```

This is an example on how a separator can be made, you can pass as many variables for each menu item as you want. Things that could be passed is where to find menu icons, the script for each menu later decides if it wants to use it or not.

3.3.9. Require

Programs sometimes depends on another program or shared library to be found, and to make it easier for the user to know what’s needed you can setup a list of required packets and versions. Required packets can be setup with 'and' and 'or' statements, is various ways.

```
require PACKET VERSION | '{' [ VERSION ... ] '}'[ '|' PACKET VERSION |
        '{' [ VERSION ... ] '}' ... ]
```

If the version is left empty will it require that any version of that packet is installed. A normal require might look something like this:

```
require Xpm ""
require zlib 1.1.3
```

This will prompt that the packet Xpm, of any version is needed as well as zlib 1.1.3. Only sometimes the user can choose between two shared libraries like lesstif and motif, this can be solved by writting:

```
require lesstif "" | motif ""
```

You can use as many or '|' as you need. The or '|' can be written in another way if you have a few versions in a packet required, as if you need some version of GTK+ before the 1.2.x version.

```
require GTK+ { 1.0.0 1.0.2/ 1.0.6 }
```

This is the same as writting a require looking as:

```
require GTK+ 1.0.0 | GTK+ 1.0.2 | GTK+ 1.0.6
```

3.3.10. Script

Scripts are a bit different then the create, see Section 3.3.3, “Create”. Instead of creating a file, or appending to one without any order, can you with script create a script file which is a bit more controled. Also this will allow you to have scripts for multiple shell types for when the user logins, and wants some options setuped. With the priority level you can make sure one script will be added before another. This is much depending on the avaliabel script types, see .

```
script [ PRIORITY ] TYPE CONTENTS
```

For example can you use it to setup shell script options, as this for tcsh:

```
script tcshrc "limit coredumpsize 0"
```

But also for many other things, as different login script for getting X server started right. As well as a script to be runned after hickup, since hickup can not always handle all cases needed yet. These files will then be placed in the hick directory $HICKROOT/scripts/$HICK-TYPE.

3.3.11. XRes

X resources can be usefull to setup for some programs, there are three ways of doing this, one is to use the link, see Section 3.3.7, “Link”, to make a link into a app-default directory. The other two is adding into the Xresource-$HICKTYPE file, which is then merge at startup.

```
xres [ inline ] NAME VALUE
xres filename FILENAME
```

You can either add entry by entry or just add a whole file into the resource file, when doing entry by entry you do not need to add the ':' at the end of the name this will be handled by hickup.

Chapter 4

Projects

This chapter will describe how you setup a project environment. To let the program know where to find projects are done by passing as an argument to the program. It will then lookup the specified projects, and if it has read access it will allow the user to choose programs for the project. If also write access are allowed is a edit tab shown and the user can by that also setup what programs can be choosen. All programs and versions are by default not allowed.

4.1. Command line argument

There are two ways of letting hickup know which projects are availible, one is by entering them all on the command like, the other is by pointing out a file containing the projects. In both cases the project specifications are written in the same format:

```
NAME:SPEC:HOME [ :NAME:SPEC:HOME ... ]
```

An example on how to write a project argument, this is for the project tools, and for the user daniel. The project lies under /home/tools and all the project settings are stored there. The argument string would look something like this:

NAME
: This is the text which will be displaied on the tab in the hickup viewer.

SPEC
: Where the project data files are stored. The project data files are the project specification file and the project information. The files which are use by hick is prjspec.hick and prjinfo.hick.

HOME
: Home directory project and user, this will be where the .hick directory will all the user data over selected packets will be stored. Should be uniq for each user, if not a global project for all users are wanted.

```
tools:/home/tools/.users:/home/tools/.users/daniel
```

One idea is to have a wrapper script calling hickup, so it will take care of all project data.

4.2. Creating a Project

We will now go through how to create a new project, the project we will setup is called tools. The way which will be shown is not the only solution on how to create a project, it is pretty much up to how you want it to work.

The project we are going to create is made so that we create a new user with it is home as everyone else on the system, like */home/tools*. To that we also create a new group in */etc/group* which will contain the members in the group.

After we have created the user and the group, we create the project data directory, which will be under the projects home directory called */home/tools/.users*. In this directory will we place the project specific files, but also the users personal hick setup.

Even if we now start hickup with the project setup wont hickup care about this project since no project specification file exists, so the next step is to create it. To do this you only need to create an empty file, this can be achived with the command **touch**.

```
# cd /home/tools/.users
# touch prjspec.hick
```

It will now recognize this project as a avalible project for the users, only no packets are allowed to be choosed, since all packets are by default in a project not allowed. The next step is then to setup the project specification, this can be done with either hickup or an text editor. For the syntax of the project file see Section 4.3, “Specification File”. When doing it in hickup you only start hickup with the project argument containing the project.

```
# hickup -p tools:/home/tools/.users:/home/tools/.users/$USER
```

If wanted you can also add project information text, this can be a good way of letting the project members know a little more about the project. The information file is a plain text file called *prjinfo.hick*.

4.3. Specification File

With a project you perhaps want to create a development environment where all the project members use some specified tools, and others which they can freely choose inbetween. This is where the project specification comes in, it allows the project manager to setup which programs are avalible to use in the project, but also which programs must be used. This can simply be achived with enabe a specific packet or version onto the project members, or even force them to use a specified version. A line begining with an '#' will be ignored. The syntax for the specification file is the following:

```
# one line comment
enable TYPE PACKET VERSION
enable TYPE PACKET '{' [ VERSION ... ] '}'
enable TYPE '{'
  [ PACKET VERSION ... ]
  [ PACKET '{' [ VERSION ... ] '}' ... ]
'}'
force TYPE PACKET VERSION
force TYPE PACKET '{' VERSION [ VERSION ... ] '}'
force TYPE '{'
  [ PACKET VERSION ... ]
```

```
  [ PACKET '{' VERSION [ VERSION ... ] '}' ... ]
'}'
```

4.4. Enable

Enables a packet or a version to be avalible to select. If the version is left empty will it affect all versions in the named packet, this can be done by writing "" or { }. If on the other hand one or more versions, but not all is only wanted to be enabled the versions is then specified. An examples on how this can be written:

```
enable solaris {
   gcc 2.7.2.2
   make ""
   autoconf { 2.12 2.11 }
}
```

This will allow the user to choose gcc 2.7.2.2 and no other version of gcc. Make any version and autoconf version 2.12 and 2.11.

4.5. Force

If something really is needed of a specified version, the force directive can be used, this will install the version on the user as if it where a force in a packet, see chapter 3.2.10. It works simmular to the enable only you have to specify at least one version. When wanting to specify more then one version the packet must support multiple choices or else will the installed version be one of the named. An example on how this can be written:

```
force solaris {
  automake 1.3
  "tcsh options" { noclobber "coredump unlimited" }
}
```

This will install the automake 1.3 and set the 'tcsh options' noclobber and 'coredump unlimited' options. The user wont be abel to choose any other existing options.