A-A-P Recipe Executive

Bram Moolenaar

A-A-P Recipe Executive

by Bram Moolenaar

Revision 0.100 (2002 Oct 25) Edition
Published 2002

This is the documentation for version 0.100 of the Recipe Executive, commonly known as the "aap" command. It is part of the A-A-P project.

NOTE: Currently only the tutorial has been done properly. The other chapters do contain lots of information, but this still needs to be organized and the layout is to be done.

The web site of A-A-P can be found here: http://www.a-a-p.org/

The HTML version of this manual can be read on-line: http://www.a-a-p.org/exec/index.html

The PDF version of this manual can be found here: http://www.a-a-p.org/exec/exec.pdf

Table of Contents

List of Tables

I. Tutorial

Chapter 1. Getting Started

To start using Aap you must have two applications:

- Python version 1.5 or later
- Aap

Python is often installed already. Try this:

```
python -V
```

If you get a "Command not found" error you still need to install Python. Help for this can be found on the Python web site: www.python.org/download/ (http://www.python.org/download/).

For obtaining and installing Aap look here: www.a-a-p.org/download.html (http://www.a-a-p.org/download.html).

To check if your Aap program is working, type this command:

```
aap --help
```

You should get a list of the command line arguments. Note that there are two dashes before "help".

Chapter 2. Compiling a Program

The "Hello world" example

Most programming languages start with a short example that prints a "hello world" message. This is the recipe for compiling such a program with A-A-P:

```
SOURCE = hello.c
TARGET = hello$EXESUF
```

Write this text in a file "main.aap", in the same directory as "hello.c". Now invoke Aap to compile "hello.c" into the program "hello":

```
% ls
hello.c         main.aap
% aap
Aap: Creating directory "/home/mool/tmp/build-FreeBSD4_5_RELEASE"
Aap: cc -I/usr/local/include -g -O2 -E -MM hello.c > build-FreeBSD4_5_RELEASE/hello.c.aap
Aap: cc -I/usr/local/include -g -O2 -c -o build-FreeBSD4_5_RELEASE/hello.o hello.c
Aap: cc -L/usr/local/lib -g -O2 -o hello build-FreeBSD4_5_RELEASE/hello.o
```

You see the commands Aap uses to compile the program:

1. A directory is created to write the intermediate results in. This directory is different for each platform, thus you can compile the same program for different systems without cleaning up.
2. Gcc is invoked with the "-MM" argument. This doesn't compile the program but only finds header files used. Aap will automatically detect dependencies on included files and knows that if one of the included files changed compilation needs to be done, even when the file itself didn't change. More about that below.
3. The "hello.c" program file is compiled into the "hello.o" object file.
4. The "hello.o" object file is linked to produce the "hello" program. Note that on MS-Windows this would be "hello.exe". That is why $EXESUF was used in the TARGET variable. Aap makes it empty on Unix and sets it to ".exe" on MS-Windows.

Assignments

The two lines in the example recipe are variable assignments. An assignment has the form:

```
variablename = expression
```

The variable name is the usual combination of letters, digits and underscore. It must start with a letter. Upper and lower case letters can be used and case matters. To see this in action, write this recipe in a file with the name "try.aap":

```
foo = one
Foo = two
FOO = three
:print $foo $Foo $FOO
```

Now execute the recipe:

```
% aap -f try.aap
one two three
Aap: No target on the command line and no $TARGET or "all" target in a recipe
%
```

The ":print" command prints its argument. You can see that a variable name predeced with a dollar is replaced by the value of the variable. The three variables that only differ by case each have a different value.

All Aap commands, except the assignment, start with a colon. That makes them easy to recognize.

Finally Aap complains there is nothing interesting to do...

Some characters in the expression have a special meaning. And the ":print" command also handles a few arguments in a special way. This table gives an overview of which characters you need to watch out for when using the ":print" command:

Table 2-1. Special characters in the ":print" command

":print" argument	resulting character
$$	$
“ (two backticks)	‘ (one backtick)
$#	#
$gt	>
$lt	<
$bar	|

Example:

```
:print tie $#2 “green“ $bar price: $$ 13 $lt incl vat $gt
```

Write this in the file "try.aap". Executing it results in:

```
% aap -f try.aap
tie #2 ‘green‘ | price: $ 13 < incl vat >
Aap: No target on the command line and no $TARGET or "all" target in a recipe
%
```

Several Source Files

When you have several files with source code you can specify them as a list:

```
SOURCE =
       main.c
       version.c
       help.c
TARGET = myprogram$EXESUF
```

There are three source files: `main.c`, `version.c` and `help.c`. Notice that it is not necessary to use a line continuation character, as you would have to do in a Makefile. The list ends at a line where the indent is equal to or less than what the assignment started with. The amount of indent for the continuation lines is irrelevant, so long as it's more than the indent of the first line.

Indents are very important, just like in a Python script. Make sure your tabstop is always set to the standard value of eight, otherwise you might run into trouble when mixing tabs and spaces!

To try out the example write it in the file "main.aap" in the same directory as where the three C files are. Then run Aap without arguments. You will see the commands used to compile the files and end up with the "myprogram" program. If you are less fortunate you will get an error message. Hopefully it makes clear what mistake you made.

Comments

Someone who sees this recipe would like to know what it's for. This requires adding comments. These start with a "#" character and extend until the end of the line (like in a Makefile and Python script).

It is also possible to associate a comment with a specific item:

```
# A-A-P recipe for compiling "myprogram"
SOURCE =
        main.c            # startup stuff
        version.c         # just the date stamp
        help.c            # display a help message

TARGET = myprogram$EXESUF { comment = build my program }
```

Now run Aap:

```
% aap comment
target "myprogram": build my program
%
```

The text inside curly braces is called an attribute. In this case the attribute name is "comment" and the attribute value is "build my program". An attribute can be used to attach extra information to a file name. We will encounter more attributes later on.

Dependencies

Let's go back to the "Hello world" example and find out what happens when you change a source file. Use this "hello.c" file:

```
#include <stdio.h>
#include "hello.h"
main()
```

```
{
    printf("Hello %s\n", world);
}
```

The included "hello.h" file defines "world":

```
#define world "World!"
```

If you run Aap, the "hello" program will be build as before. If you run Aap again you will notice that nothing happens. Aap remembers that "hello.c" was already compiled. Now try this:

```
% touch hello.c
% aap
%
```

If you have been using the "make" program you would expect something to happen. But Aap checks the contents of the file, not the timestamp. A signature of "hello.c" is computed and if it is still the same as before Aap knows that it does not need to be compiled, even though "hello.c" is newer than the "hello" program.

Aap uses the mechanism of dependencies. When you specify "SOURCE" and "TARGET" Aap knows that the target depends on the sources. When one of the sources changes, the commands to build the target from the sources must be executed. This can also be specified explicitly:

```
hello$EXESUF : $BDIR/hello.o
    :do build $source

$BDIR/hello.o : hello.c
    :do compile $source
```

The generic form of a dependency is:

```
target : list-of-source-files
    build-commands
```

The colon after the target is important, it separates the target from the sources. It is not required to put a space before it, but there must be a space after it. We mostly put white space before the colon, so that it is easy to spot. There could be several targets, this is handled later.

There are two dependencies in the example. In the first one the target is "hello$EXESUF", the source file is "$BDIR/hello.o" and the build command is ":do build $source". This specifies how to build the "hello$EXESUF" program from the "$BDIR/hello.o" object file. The second dependency specifies how to compile "hello.c" into "$BDIR/hello.o" with the command ":do compile $source". The "BDIR" variable holds the name of the platform-dependent directory for intermediate results, as mentioned in the first example of this chapter.

The relation between the two dependencies in the example is that the source of the first one is the target in the second one. The logic is that Aap follows the dependencies and executes the associated build commands. In this case "hello$EXESUF" depends on "$BDIR/hello.o", which then depends on "hello.c". The last dependency is handled first, thus first hello.c is compiled by the build command of the second dependency, and then linked into "hello$EXESUF" by the build command of the first dependency.

Now change the "hello.h" file by replacing "World" with ’Universe":

```
#define world "Universe!"
```

If you now run Aap with "aap hello" or "aap hello.exe" the "hello" program will be build. But you never mentioned the "hello.h" file in the recipe. How did Aap find out the change in this file matters? When Aap is run to update the "hello" program, this is what will happen:

1. The first dependency with "hello$EXESUF" as the target is found, it depends on "$BDIR/hello.o".
2. The second dependency with "$BDIR/hello.o" as the target is found. The source file "hello.c" is recognized as a C program file. It is inspected for included files. This finds the "hello.h" file. "stdio.h" is ignored, since it is a system file. "hello.h" is added to the list of files that the target depends on.
3. Each file that the target depends on is updated. In this case "hello.c" and "hello.h". No dependency has been specified for them and the files exist, thus nothing happens.
4. Aap computes signatures for "hello.c" and "hello.h". It also computes a signature for the build commands. If one of them changed since the last time the target was build, or the target was never build before, the target is considered "outdated" and the build commands are executed.
5. The second dependency is now finished, "$BDIR/hello.o" is up-to-date. Aap goes back to the first dependency.
6. Aap computes a signature for "$BDIR/hello.o". Note that this happens after the second dependency was handled, it may have changed the file. It also computes a signature for the build command. If one of them changed since the last time the target was build, or the target was never build before, the target is considered "outdated" and the build commands are executed.

Now try this: Append a comment to one of the lines in the "hello.c" file. This means the file is changed, thus when invoking Aap it will compile "hello.c". But the program is not build, because the produced intermediate file "$BDIR/hello.o" is still equal to what it was the last time. When compiling a large program with many dependencies this mechanism avoids that adding a comment may cause a snowball effect. (Note: some compilers include line numbers or a timestamp in the object file, in that case building the program will happen anyway).

By the way, the "Hello world" example for an Aap recipe is very simple:

```
all:
   :print Hello world!
```

Compiling Multiple Programs

Suppose you have a number of sources files that are used to build two programs. You need to specify which files are used for which program. Here is an example:

```
1.    COMMON = help.c util.c
2.
3.    all : foo bar
4.
5.    foo : $COMMON foo.c
```

```
6.          :do build $source
7.
8.    bar : $COMMON bar.c
9.          :do build $source
```

This recipe defines three targets: "all", "foo" and "bar". We left out the $EXESUF here to keep it simple. "foo" and "bar are programs that Aap can build from source files. But the "all" target is not a file. This is called a virtual target: A name used for a target that does not exist as a file. Aap knows the target "all" (and some others) is always used as a virtual target. For other targets you need to specify it with the "{virtual}" attribute.

The first dependency has no build commands. This only specifies that "all" depends on "foo" and "bar". Thus when Aap updates the "all" target, this dependency specifies that "foo" and "bar" need to be updated. Since the "all" target is the default target, this dependency causes both "foo" and "bar" to be updated when Aap is started without an argument. You can use "aap foo" to build "foo" only. The dependencies for "all" and "bar" will not be used then.

The two files "help.c" and "util.c" are used by both the "foo" and the "bar" program. To avoid having to type the file names twice, the "COMMON" variable is used. Note that the assignment in the first line uses "COMMON" without a dollar, while in line 5 and 8 "$COMMON" is used. This makes it possible to mix the use of a variable with plain text and file names. This recipe would do the same thing:

```
all : foo bar

foo : help.c util.c foo.c
        :do build $source

bar : help.c util.c bar.c
        :do build $source
```

You can see that "$COMMON" has been replaced by the value of the "COMMON" variable.

A variable name consists of ASCII letters, numbers and the underscore. If you want to append one of these after a variable name you must put parenthesis around the variable name, so that Aap knows where it ends:

```
MAKENAME = Make
BUILDFILE = $(MAKENAME)file
:print $BUILDFILE
```

```
% aap -f try.aap
Makefile
Aap: No target on the command line and no $TARGET or "all" target in a recipe
%
```

The "-f" argument is used to specify the recipe to be read by Aap. If you don’t give it, the recipe "main.aap" is read.

Chapter 3. Publishing a Web Site

If you are maintaining a web site it is often a good idea to edit the files on your local system. After trying out the changes you then need to upload the changed files to the web server. A-A-P can be used to identify the files that changed and upload these files only. This is called publishing.

Uploading The Files

Here is an example of a recipe:

```
FILES = index.html
        project.html
links.html
        images/logo.png
:attr {publish = scp://user@ftp.foo.org/public_html/%file%} $FILES
```

That's all. You just need to specify the files you want to publish and the URL that says how and where to upload them to. Now "aap publish" will find out which files have changed and upload them:

```
% aap publish
Aap: Uploading ['/home/mool/www/foo/index.html'] to scp://user@ftp.foo.org/public_html/index.html
Aap: scp '/home/mool/www/vim/index.html' 'user@ftp.foo.org:public_html/index.html'
Aap: Uploaded "/home/mool/www/vim/index.html" to "scp://user@ftp.foo.org/public_html/index.html"
%
```

The first time you execute the recipe all files will be uploaded. Aap will create the "images" directory for you. If you had already uploaded the files and want to avoid doing it again, first run the recipe with: "aap publish --touch". Aap will compute the signatures of the files as they are now and remember them. Only files that are changed will be uploaded from now on.

The ":attr" command uses its first argument as an attribute and further arguments as file names. It will attatch the attribute to each of the files. In this case the "publish" attribute is added, which specifies the URL where to upload a file to. In the example the "scp" protocol is used, which is a good method for uploading files to a public server. "ftp" can be used as well, but this means your password will go over the internet, which is not safe. The special item "%file%" is replaced with the name of the file being published.

Generating a HTML File

It is common for HTML files to consist of a standard header, a body with the useful info and a footer. You don't want to manually add the header and footer to each page. When the header changes you would have to make the same change in many different files. Instead, use the recipe to generate the HTML files.

Let's start with a simple example: Generate the index.html file. Put the common header, containing a logo and navigation links, in "header.part". The footer, containing contact info for the maintainer, goes in "footer.part". The useful contents of the page goes in "index_body.part". Now you can use this recipe to generate "index.html" and publish it:

```
FILES =    index.html
```

```
           images/logo.png
:attr {publish = scp://user@ftp.foo.org/public_html/%file%} $FILES

all: $FILES

publish: $FILES
    :publishall

index.html: header.part index_body.part footer.part
    :cat $source >! $target
```

Notice that only the published files are put in the "FILES" variable. These files get a "publish" attribute, which tells Aap that these are the files that need to be uploaded. The ".part" files are not published, thus they do not get the "publish" attribute.

Three dependencies follow. The "all" target is the virtual target we have seen before. It specifies that the default work for this recipe is to update the files in the "FILES" variable. This means you don’t accidentally upload the files by running "aap" without arguments. The normal way of use is to run "aap", check if the produced HTML file looks OK, then use "aap publish" to upload the file.

For "index.html" a target is specified with a build command. The ":cat" command concatenates the source files. "$source" stands for the source files used in the dependency: "header.part, "index_body.part" and "footer.part". The resulting text is written to "$target", which is the target of the dependency, thus "index.html". The ">!" is used to redirect the output of the ":cat" command and overwrite any existing result. This works just like the Unix "cat" command.

In the dependency with the "publish" target the ":publishall" command is used. This command goes through all the files which were given a "publish" attribute with the ":attr" command. Note that this does not work:

```
# This won’t work.
FILES = index.html {publish = scp://user@ftp.foo.org/public_html/%file%}
```

Using a "publish" attribute in an assignment will not make it used with the ":publishall" command.

Using ":rule" to Generate Several HTML Files

Your web site contains several pages, thus you need to specify how to generate each HTML page. This quickly becomes a lot of typing. We would rather specify once how to make a "xxx.html" file from a "xxx_body.part" file, and then give the list of names to use for "xxx" (if you have assocations with the name "xxx_body.part" that is your own imagination! :-). This is how it’s done:

```
FILES =    ‘glob("*.html")‘
           ‘glob("images/*.png")‘
:attr {publish = scp://user@ftp.foo.org/public_html/%file%} $FILES

all: $FILES

publish: $FILES
    :publishall

:rule %.html : header.part %_body.part footer.part
```

```
        :cat $source >! $target
```

This is very similar to the example that only generates the "index.html" file. The first difference is in the assignment of "FILES". This uses the Python function "glob()". It expands wildcards like a shell. The backticks surround the Python expression. The result is that "FILES" gets a list of all "*.html" files in the current directory and all "*.png" files in the "images" directory. We will encounter more examples of using a Python expression further on.

The second difference is that there is no specific dependency for the "index.html" file but a ":rule" command. It looks very much the same, but the word "index" has been replaced by a percent character. You could read the rule command as a dependency where the "%" stands for "anything". In the example the target is "anything.html" and in the sources we find "anything_body.part". Obviously these two occurrences of "anything" are the same word.

If you have made HTML pages, you know they contain a title. We ignored that until now. The following recipe will handle a title, stored in the file "xxx_title.part". You also need a file "start.part", which contains the HTML code that goes before the title.

```
    FILES =    ‘glob("*.html")‘
               ‘glob("images/*.png")‘
    :attr {publish = scp://user@ftp.foo.org/public_html/%file%} $FILES

    all: $FILES

    publish: $FILES
        :publishall

    :rule %.html : start.part %_title.part header.part %_body.part footer.part
        :cat $source >! $target
```

Notice that "%" is now used three times in the ":rule" command. It stands for the same word every time.

After writing this recipe you can forget what changes you made to what file. A-A-P will take care of generating and uploading those HTML files that are affected. For example, if you change "header.part", all the HTML files are generated and uploaded. If you change "index_title.part" only "index.html" will be done.

There is one catch: You must create an (empty) xxx.html file the first time, otherwise it will not be found with "*.html". And you have to be careful not to have other "xxx.html" files in this directory. You might want to explicitly specify all the HTML files instead of using "glob()".

A similar recipe is actually used to update the A-A-P website. It’s a bit more complicated, because not all pages use the same header.

Chapter 4. Distributing a Program

Open source software needs to be distributed. This chapter gives a simple example of how you can upload your files and make it easy for others to download and install your program.

Downloading

To make it easy for others to obtain the latest version of your program, you give them a recipe. That is all they need. In the recipe you describe how to download the files and compile the program. Here is an example:

```
1     ORIGIN = ftp://ftp.mysite.org/pub/theprog
2
3     :recipe {fetch = $ORIGIN/main.aap}
4
5     SOURCE = main.c
6              version.c
7     HEADER = common.h
8     TARGET = theprog
9
10    :attr {fetch = $ORIGIN/%file%} $SOURCE $HEADER
```

The first line specifies the location where all the files can be found. It is good idea to specify this only once. If you would use the text all over the recipe it is more difficult to read and it would be more work when the URL changes.

Line 3 specifies where this recipe can be obtained. After obtaining this recipe once, it can be updated with a simple command:

```
% aap refresh
Aap: Updating recipe "main.aap"
Aap: Attempting download of "ftp://ftp.mysite.org/pub/theprog/main.aap"
Aap: Downloaded "ftp://ftp.mysite.org/pub/theprog/main.aap" to "/home/mool/.aap/cache/98092140.aap"
Aap: Copied file from cache: "main.aap"
%
```

The messages from Aap are a bit verbose. This is just in case the downloading is very slow, you will have some idea of what is going on.

Lines 5 to 8 define the source and target files. This is not different from the examples that were used to compile a program.

The last line specifies where the files can be fetched from. This is done by giving the source and header files the `fetch` attribute. The `:attr` command does not cause the files to be fetched yet. When a file is used somewhere and it has a `fetch` attribute, then it is fetched. Thus files that are not used will not be fetched.

A user of your program stores this recipe as "main.aap" and runs **aap** without arguments. What will happen is:

1. Dependencies will be created from `SOURCE` and `TARGET` to build "theprog" from "main.c" and "version.c".
2. The target "theprog" depends on "main.c" and "version.c". Since these files do not exist and they do have a `fetch` attribute, they are fetched.
3. The "main.c" file is inspected for dependencies. It includes the "common.h" file, which is automatically added to the list of dependencies. Since "common.h" does not exist and has a `fetch` attribute, it is fetched as well.
4. Now that all the files are present they are compiled and linked into "theprog".

Uploading

You need to upload the files mentioned in the recipe above. This needs to be repeated each time one of the files changes. This is essentially the same as publishing a web site. This recipe will do the work:

```
:include main.aap
TARGET =

URL = scp://user@ftp.mysite.org//pub/theprog/%file%
:attr {publish = $URL} $SOURCE $HEADER main.aap

all: publish
```

Write this recipe as "publish.aap" and execute it with **`aap -f publish.aap`**. There is no need to specify the target to be build, since the last line specifies that the default target "all" depends on "publish" and the "publish" target publishes all files with a `publish` attribute.

In the first line the `:include` command is used to include the recipe "main.aap". This works like the contents of "main.aap" was present in place of the `:include` command. The "main.aap" recipe defines `SOURCE` and `HEADER`. Including this avoids having to list the source files again. But the "main.aap" recipe also defined `TARGET`, which would trigger the default rule to build a program. We don't want that here, therefore `TARGET` is made empty after including "main.aap".

Chapter 5. Building Variants

A-A-P provides a way to build two variants of the same application. You just need to specify what is different about them. A-A-P will then take care of putting the resulting files in a different directory, so that you don’t have to recompile everything when you toggle between two variants.

One Choice

Quite often you want to compile an application for release with maximal optimizing. But the optimizer confuses the debugger, thus when stepping through the program to locate a problem, you want to recompile without optimizing. Here is an example:

```
1    SOURCE = main.c version.c gui.c
2
3    :variant BUILD
4        release
5            CFLAGS = -O4
6            TARGET = myprog
7        debug
8            CFLAGS = -g
9            TARGET = myprogd
```

Write this recipe as "main.aap" and run Aap without arguments. This will build "myprog" and use a directory for the object files that ends in "-release". The release variant is the first one mentioned, that makes it the default choice.

The first argument for the `:variant` command is `BUILD`. This is the name of the variable that specifies what variant will be selected. The names of the alternatives are specified with a bit more indent in lines 4 and 7. For each alternative two commands are given, again with more indent. Note that the indent not only makes it easy for you to see the parts of the `:variant` command, they are essential for Aap to recognize them.

To select the "debug" variant the `BUILD` variable must be set to "debug". A convenient way to do this is by specifying this on the command line:

```
% aap BUILD=debug
```

This will build the "myprogd" program, using "-g" instead of "-O4" for `CFLAGS`. The object files are stored in a directory ending in "-debug". Once you finished debugging and fixed the problem in, for example, "gui.c", running Aap to build the release variant will only compile the modified file. There is no need to compile all the C files, because the object files for the "release" variant are still in the "-release" directory.

Two Choices

You can extend the `BUILD` variant with more items, for example "profile". This is useful for alternatives that exclude each other. Another possibility is to add a second `:variant` command. Let us extend the example with a selection of the user interface type.

```
1    SOURCE = main.c version.c gui.c
```

```
2
3    :variant BUILD
4        release
5            CFLAGS = -O4
6            TARGET = myprog
7        debug
8            CFLAGS = -g
9            TARGET = myprogd
10
11   GUI ?= motif
12   :variant GUI
13       console
14       motif
15            SOURCE += motif.c
16       gtk
17            SOURCE += gtk.c
18
19   CFLAGS += -DGUI=$GUI
```

The `:variant` command in line 12 uses the `GUI` variable to select one of "console", "motif" or "gtk". Together with the earlier `:variant` command this offers six alternatives: "release" with "console", "debug" with "console", "release" with "motif", etc. To build "debug" with "gtk" use this command:

```
% aap BUILD=debug GUI=gtk
```

In line 11 an optional assignment "?=" is used. This assignment is skipped if the `GUI` variable already has a value. Thus if `GUI` was given a value on the command line, as in the example above, it will keep this value. Otherwise it will get the value "motif".

Note: Environment variables are not used for variables in the recipe, like `make` does. When you happen to have a `GUI` environment variable, this will not influence the variant in the recipe. This is especially useful if you are not aware of what environment variables are set and/or which variables are used in the recipe. If you intentionally want to use an environment variable this can be specified with a Python expression (see the next chapter).

In line 15, 17 and 19 the append assignment "+=" is used. This appends the argument to an existing variable. A space is inserted if the value was not empty. For the variant "release" with "motif" the result of line 19 is that `CFLAGS` becomes "-O4 -DGUI=motif".

The "motif" and "gtk" variants each add a source file in line 15 and 17. For the console version no extra file is needed. The object files for each combination of variants end up in a different directory. Ultimately you get object files in each of the six directories ("SYS" stands for the platform being used):

directory	contains files
build-SYS-release-console	main, version, gui
build-SYS-debug-console	main, version, gui
build-SYS-release-motif	main, version, gui, motif
build-SYS-debug-motif	main, version, gui, motif
build-SYS-release-gtk	main, version, gui, gtk

directory	contains files
build-SYS-debug-gtk	main, version, gui, gtk

Compile only when needed

We happen to know that the main.c file does not depend on the GUI used. With the recipe above it will nevertheless be compiled again for every GUI version. Although this is a small thing in this example, in a bigger project it becomes more important to skip compilation when it is not needed. Here is the modified recipe:

```
1    SOURCE = main.c version.c gui.c
2
3    :variant BUILD
4        release
5            CFLAGS = -O4
6            TARGET = myprog
7        debug
8            CFLAGS = -g
9            TARGET = myprogd
10
11   :attr {var_CFLAGS = $CFLAGS} {var_BDIR = $BDIR} main.c
12
13   GUI ?= motif
14   :variant GUI
15       console
16       motif
17            SOURCE += motif.c
18       gtk
19            SOURCE += gtk.c
20
21   CFLAGS += -DGUI=$GUI
```

The only new line is line 11. The "main.c" file is given two extra attributes: `var_CFLAGS` and `var_BDIR`. What happens is that when "main.c" is being build, Aap will check for attributes of this source file that start with "var_". The values will be used to set variables with the following name to the value of the attribute. Thus `CFLAGS` gets the value of `var_CFLAGS`. This means that the variable is overruled by the attribute while building "main.c".

The `var_BDIR` attribute is set to "$BDIR" before the second `:variant` command. It does not yet have the selected GUI appended there. The list of directories used is now:

directory	contains files
build-SYS-release	main
build-SYS-debug	main
build-SYS-release-console	version, gui
build-SYS-debug-console	version, gui
build-SYS-release-motif	version, gui, motif

directory	**contains files**
build-SYS-debug-motif	version, gui, motif
build-SYS-release-gtk	version, gui, gtk
build-SYS-debug-gtk	version, gui, gtk

Chapter 6. Using Python

In various places in the recipe Python commands and expressions can be used. Python is a powerful and portable scripting language. In most recipes you will only use a few Python items. But where needed you can do just about anything with it.

Conditionals

When a recipe needs to work both on Unix and on MS-Windows you quickly run into the problem that the compiler does not use the same arguments. Here is an example how you can handle that.

```
@if OSTYPE == "posix":
    CPPFLAGS += -DFAST
@else:
    CPPFLAGS += /DFAST

all:
    :print CPPFLAGS is "$CPPFLAGS"
```

The first and third line start with the "@" character. This means a Python command follows. The other lines are normal recipe lines. You can see how these two kinds of lines can be mixed.

The first line is a simple "if" statement. The `OSTYPE` variable is compared with the string "posix". If they compare equal, the next line is executed. When the `OSTYPE` variable has a different value the line below `@else:` is executed. Executing this recipe on Unix:

```
% aap
CPPFLAGS is "-I/usr/local/include -DFAST"
%
```

`OSTYPE` has the value "posix" only on Unix and Unix-like systems. Executing the recipe on MS-Windows, where `OSTYPE` has the value "mswin":

```
C:> aap
CPPFLAGS is "/IC:\VC\include /DFAST"
C:>
```

Note that the Python conditional commands end in a colon. Don’t forget to add it, you will get an error message! The indent is used to form blocks, thus you must take care to align the "@if" and "@else" lines.

You can include more lines in a block, without the need for extra characters, such as { } in C:

```
@if OSTYPE == "posix":
    CPPFLAGS += -DFAST
    LDFLAGS += -L/usr/local
@else:
    CFLAGS += /DFAST
```

Loops

Python has a "for" loop that is very flexible. In a recipe it is often used to go over a list of items. Example:

```
1      @for name in [ "solaris", "hpux", "linux", "freebsd" ]:
2          fname = README_$name
3          @if os.path.exists(fname):
4              FILES += $fname
5      all:
6          :print $FILES
```

The first line contains a list of strings. A Python list uses square brackets. The lines 2 to 4 are executed with the `name` variable set to each value in the list, thus four times. The indent of line 5 is equal to the `@for` line, this indicates the "for" loop has ended.

Note how the `name` and `fname` variables are used without a dollar in the Python code. This might be a bit confusing at first. Try to remember that you only put a dollar before a variable name in the argument of a recipe command.

In line 2 the `fname` variable is set to "README_" plus the value of `name`. The `os.path.exists()` function in line 3 tests if a file exists. Assuming all four files exist, this is the result of executing this recipe:

```
% aap
README_solaris README_hpux README_linux README_freebsd
%
```

Python Block

When the number of Python lines gets longer, the "@" characters become annoying. It is easier to put the lines in a block. Example:

```
:python
    FILES = ”
    for name in [ "solaris", "hpux", "linux", "freebsd" ]:
        fname = "README_" + name
        if os.path.exists(fname):
            if FILES:
                FILES = FILES + ’ ’
            FILES = FILES + fname
all:
    :print $FILES
```

This does the same thing as the above recipe, but now using Python commands. As usual, the `:python` block ends where the indent is equal to or less than that of the `:python` line.

When using the `:python` command, make sure you get the assignments right. Up to the "=" character the Python assignment is the same as the recipe assignment, but what comes after it is different.

Expressions

In many places a Python expression can be used. We have already seen the use of the `glob()` function:

```
SOURCE = ‘glob("*.c")‘
```

Python users know that the `glob()` function returns a list of items. Aap automatically converts the list to a string, because all Aap variables are strings. A space is inserted in between the items and quotes are added around items that contain a space.

> It is actually a bit dangerous to get the list of source files with the `glob()` function, because a "test.c" file that you temporarily used will accidentally be included. It is often better to list the source files explicitly.

The following example turns the list of source files into a list of header files:

```
SOURCE = ‘glob("*.c")‘
HEADER = ‘aap_sufreplace(".c", ".h", SOURCE)‘
all:
    :print SOURCE is "$SOURCE"
    :print HEADER is "$HEADER"
```

Running Aap in a directory with "main.c" and "version.c"?

```
% aap
SOURCE is "version.c main.c"
HEADER is "version.h main.h"
%
```

The "aap_sufreplace()" function takes three arguments. The first argument is the suffix which is to be replaced. The middle argument is the replacement suffix. The last argument is the name of a variable that is a list of names, or a Python expression. In this example each name in `SOURCE` ending in ".c" will be changed to end in ".h".

Further Reading

Documentation about Python can be found on its web site: http://www.python.org/doc/

Chapter 7. Version Control with CVS

CVS is often used for development of Open Source Software. A-A-P provides facilities to obtain the latest version of an application and for checking in changes you made.

Downloading

For downloading a whole module you only need to specify the location of the CVS server and the name of the module. Here is an example that obtains the A-A-P Recipe Executive:

```
   CVSROOT = :pserver:anonymous@cvs.a-a-p.sf.net:/cvsroot/a-a-p
   all:
:fetch {fetch = cvs://$CVSROOT} Exec
```

Write this recipe as "main.aap" and run **aap**. The directory "Exec" will be created and all files in the module obtained from the CVS server:

```
% aap
Aap: CVS checkout for node "Exec"
Aap: cvs -d:pserver:anonymous@cvs.a-a-p.sf.net:/cvsroot/a-a-p checkout ’Exec’
cvs server: Updating Exec
U Exec/Action.py
U Exec/Args.py
[....]
%
```

The `:fetch` command takes care of obtaining the latest version of the items mentioned as arguments. Usually the argument is one module, in this example it is "Exec". That CVS needs to be used is specified with the `fetch` attribute. This is a kind of URL, starting with "cvs://" and then the CVS root specification. In the example the `CVSROOT` variable was used. This is not required, it just makes the recipe easier to understand.

Note that this only works when you have the "cvs" command installed, otherwise you will get an error message.

If the software has been updated, you can get the latest version by running "aap" again. CVS will take care of obtaining the changed files.

Uploading

You are the maintainer of a project and want to distribute your latest changes, so that others can obtain the software with a recipe as used above. This means you need to checkin your files to the CVS server. This is done by listing the files that need to be distributed and giving them a `commit` attribute. Example:

```
CVSUSER_FOO = johndoe
CVSROOT = :ext:$CVSUSER_FOO@cvs.foo.sf.net:/cvsroot/foo
FILES =  main.c
         common.h
         version.c
:attr {commit = cvs://$CVSROOT} $FILES
```

Write this as "cvs.aap" and run **`aap -f cvs.aap revise`** . What will happen is:

1. Files that you changed since the last checkin will be checked in to the CVS server.
2. Files that you added to the list of files with a `commit` attribute will be added to the CVS module.
3. Files that you removed from the list of files with a `commit` attribute will be removed from the CVS module.

This means that you must take care the `FILES` variable lists exactly those files you want to appear in the CVS module, nothing more and nothing less. Be careful with using something like `glob("*.c")`, it might find more files that you intended.

Note: This only works when the CVS module was already setup. Read the CVS documentation on how to do this. The A-A-P user manual has useful hints as well.

In the example the `CVSUSER_FOO` variable is explicitly set, thus this recipe only works for one user. Better is to move this line to your own default recipe, e.g., "~/.aap/startup/default.aap". Then the above recipe does not explicitly contain your user name and can also be used by others.

Once you tested this recipe and it works, you can easily distribute your software with **`aap -f cvs.aap revise`**. You don't have to worry about the exact CVS commands to be used. However, don't use this when you want to checkin only some of the changes you made. And the example does not work well when others are also changing the same module.

Chapter 8. Filetypes and Actions

A-A-P can recognize what the type of a file is, either by looking at the file name or by inspecting the contents of the file. The filetype can then be used to decide how to perform an action with the file.

A New Type of File

Suppose you are using the "foo" programming language and want to use A-A-P to compile your programs. Once this is has been setup you can compile "hello.foo" into the "hello" program with a simple recipe:

```
SOURCE = hello.foo
TARGET = hello
```

You need to explain Aap how to deal with "foo" files. This is done with a recipe:

```
:filetype
    suffix foo foo

:action compile foo
    :sys foocomp $source -o $target
```

For Unix, write this recipe as "/usr/local/share/aap/startup/foo.aap" or "~/.aap/startup/foo.aap". The recipes in these "startup" directories are always read when Aap starts up.

Now try it out, using the simple recipe at the top as "main.aap":

```
% aap
Aap: foocomp hello.foo -o build-FreeBSD4_5_RELEASE/hello.o
Aap: cc -L/usr/local/lib -g -O2 -o hello build-FreeBSD4_5_RELEASE/hello.o
%
```

The "foo.aap" recipe does two things. The `:filetype` command is used to tell A-A-P to recognize your "hello.foo" file as being a "foo" file. The `:action` command is used to specify how the "foocomp" compiler is used to compile a "foo" program into an object file.

Defining a Filetype by Suffix

The `:filetype` command is followed by the line "suffix foo foo". The first word "suffix" means that recognizing is done by the suffix of the file name (the suffix is what comes after the last dot in the name). The second word is the suffix and the third word is the type. Quite often the type is equal to the suffix, but not always. Here are a few more examples of lines used with `:filetype`:

```
:filetype
    suffix fooh foo
    suffix bash sh
```

It is also possible to recognize a file by matching the name with a pattern, checking the contents of the file or using a Python script. See the user manual.

Defining a Compile Action

The lower half of "foo.aap" specifies the compile action for the "foo" filetype:

```
:action compile foo
    :sys foocomp $source -o $target
```

The `:action` command has two arguments. The first one specifies the kind of action that is being defined. In this case "compile". This action is used to make an object file from a source file. The second argument specifies the type of source file this action is used for, in this case "foo".

Below the `:action` line the build commands are specified. In this case just one, there could be more. The `:sys` command invokes an exteral program, "foocomp", and passes the arguments. In an action `$source` is expanded to the source of the action and `$target` to the target. These are obtained from the `:do` command that invokes the action. Example:

```
:do compile {target = $BDIR/main.$OBJSUF} main.foo
```

This `:do` command invokes the compile action, specified with its first argument. The target is specified as an attribute to the action, the source is the following argument "main.foo". When executing the `:do` command the filetype of "main.foo" is detected to be "foo", resulting in the compile action for "foo" to be invoked. In the build command of the action `$source` and `$target` are replaced, resulting in:

```
:sys foocomp main.foo -o $BDIR/main.$OBJSUF
```

Note that in many cases `$target` is passed implicitly from a dependency and does not appear in the `:do` command argument.

Another Use of Filetypes

When building a program you often want to include the date and time when it was build. A simple way of doing this is creating a source file "version.c" that contains the timestamp. This file needs to be compiled every time your program is build. Here is an example how this can be done:

```
1   SOURCE =
2            main.c
3            work.c
4   TARGET = prog {filetype = myprog}
5
6   :action build myprog object
7        VERSION_OBJ = $BDIR/version$OBJSUF
8        :do compile {target = $VERSION_OBJ} version.c
9        :do build {target = $target {filetype = program}} $source $VERSION_OBJ
```

The target "prog" is explicitly given a different filetype in line 4. The default filetype for a program is "program", here it is set to "myprog". This allows us to specify a different build action for "prog".

Write the recipe as "main.aap" (without the line numbers) and execute it with **`aap`**. The first time all the files will be compiled and linked together. Executing **`aap`** again will do nothing. Thus the timestamp used in "version.c" will not be updated if the files were not changed. If you now make a change in "main.c" and run **`aap`** you will see that both "main.c" and "version.c" are compiled.

The `:action` command in line 6 has three arguments. The first one "build" is the kind of action, like before. The second argument "myprog" specifies the target filetype, the third one "object" the source filetype. Thus the template is:

```
:action kind-of-action  target-filetype  source-filetype
```

This order may seem a bit strange. Remember that putting the target left of the source also happens in a dependency and an assignment.

There are three commands for the build action, lines 7 to 9. The first one assigns the name of the object file for "version.c" to `VERSION_OBJ`. "version.c" was not included in `SOURCE` at the top, it is compiled here explicitly in line 8. This is what makes sure "version.c" is compiled each time "prog" is build. The other source files will be compiled with the default rules used when `SOURCE` and `TARGET` variables are defined.

Finally the `:do build` command in line 9 invokes the build action to link all the object files together. Note that the target for this build action is explicitly defined and its "filetype" attribute is set to "program". This is required for this `:do` command to use the default action for a program target. Otherwise the action would invoke itself, since the filetype for $target is "myprog".

Chapter 9. More Than One Recipe

When you are working on a project that is split up in several directories it is convenient to use one recipe for each directory. There are several ways to split up the work and use a recipe from another recipe.

Children

A large program can be split in several parts. This makes it easy for several persons to work in parallel. You then need to allow the files in each part to be compiled separately and also want to build the complete program. A convenient way to do this is putting files in separate directories and creating a recipe in each directory. The recipe at the top level is called the parent. Here is an example that includes two recipes in subdirectories, called the children:

```
1    :child core/main.aap       # sets CORE_OBJ
2    :child util/main.aap       # sets UTIL_OBJ
3
4    all: theprog$EXESUF
5
6    theprog$EXESUF : core/$*CORE_OBJ util/$*UTIL_OBJ
7        :do build $source
```

In the first two lines the child recipes are included. These specify how the source files in each directory are to be compiled and assign the list of object files to `CORE_OBJ` and `UTIL_OBJ`. This parent recipe then defines how the object files are linked together to build the program "theprog".

In line 6 a special mechanism is used. Assume that `CORE_OBJ` has the value "main.c version.c". Then "core/$*CORE_OBJ" will expand into "core/main.c core/version.c". Thus "core/" is prepended to each item in `CORE_OBJ`. This is called rc-style expansion. You can remember it by thinking of the "*" to multiply the items.

An important thing to notice is that the parent recipe does not need to know what files are present in the subdirectories. Only the child recipes contain the list of files. Thus when a file is added, only one recipe needs to be changed. The "core/main.aap" recipe contains the list of files in the "core" directory:

```
1    SOURCE =  main.c
2              version.c
3
4    CPPFLAGS += -I../util
5
6    CORE_OBJ = ‘src2obj(globals(), SOURCE)‘
7    :export CORE_OBJ
8
9    all: $CORE_OBJ
```

Variables in a child recipe are local to that recipe. The `CPPFLAGS` variable that is changed in line 4 will remain unchanged in the parent recipe and other children. That is desired here, since finding header files in "../util" is only needed for files used in this recipe.

The `CORE_OBJ` variable we do want to be available in the parent recipe. That is done with the `:export` command in line 7.

The value of `CORE_OBJ` is set with a Python expression. The `src2obj()` function takes a list of source file names and transforms them into object file names. This takes care of changing the files in `SOURCE` to prepend $BDIR and change the file suffix to $OBJSUF.

In the last line is specified what happens when running **`aap`** without arguments in the "core" directory: The object files are build. There is no specification for how this is done, thus the default rules will be used.

All the files in the child recipe are defined without mentioning the "core" directory. That is because all parent and child recipes are executed with the current directory set to where the recipe is. Note the files in `CORE_OBJ` are passed to the parent recipe, which is in a different directory. That is why the parent recipe had to prepend "core/" when using `CORE_OBJ`.

Sharing Settings

Another mechanism to use a recipe is by including it. This is useful to put common variables and rules in a recipe that is included by several other recipes. Example:

```
CPPFLAGS += -DFOOBAR
:rule %$OBJSUF : %.foo
    :sys foocomp $source -o $target
```

This recipe adds something to `CPPFLAGS` and defines a rule to turn a ".foo" file into an object file. Suppose you want to include this recipe in all the recipes in your project. Write the above recipe as "common.aap" in the top directory of the project. Then in "core/main.aap" and "util/main.aap" put this command at the top:

```
:include ../common.aap
```

The `:include` command works like the commands in the included recipe were typed instead of the `:include` command. There is no change of directory, like with the `:child` command and there is no need to use `:export` in the included recipe.

In the toplevel recipe you need include "common.aap" as well. Suppose you include it in the first line of the recipe, before the `:child` commands. The children also include "common.aap". The `CPPFLAGS` variable would first be appended to in the toplevel recipe, then passed to the child and appended to again. To avoid this, put the `:include` command after the `:child` commands:

```
:child core/main.aap       # sets CORE_OBJ
:child util/main.aap       # sets UTIL_OBJ
:include common.aap

all: theprog$EXESUF

theprog$EXESUF : core/$*CORE_OBJ util/$*UTIL_OBJ
    :do build $source
```

You might argue that it’s easier to put the `:include` command at the top, so that the children don’t have to include "common.aap". You could do this, but then it is no longer possible to execute the child recipe by itself.

Executing a Recipe

Besides `:child` and `:include` there is a third way to use another recipe: `:execute`. This command executes a recipe. This works as if Aap was run as a separate program with this recipe, except that it is possible to export variables. Here is an example:

```
    SOURCE = main.c common.c
    TARGET = prog

    test:
        :execute test.aap test
:print $TEST_RESULT
```

This recipe uses `SOURCE` and `TARGET` as we have seen before. This takes care of building the "prog" program. For testing a separate recipe is used, called "test.aap". The first argument of the `:execute` command is the recipe name. Further arguments are handled like the arguments of the **`aap`** command. In this case the target "test" is used.

The "test.aap" recipe sets the `TEST_RESULT` variable to a message that summarizes the test results. To get this variable back to the recipe that executed "test.aap" these lines are used:

```
@if all_done:
    TEST_RESULT = All tests completed successfully.
@else:
    TEST_RESULT = Some tests failed!
:export TEST_RESULT
```

It would also be possible to use the `:child` command to reach the "test" target in it. The main difference is that other targets in "test.aap" could interfere with targets in this recipe. For example, "test.aap" could define a different "prog" target, to compile the program with specific test options. By using `:execute` we don't need to worry about this. In general, the `:child` command is useful when splitting up a tree of dependencies in parts, while `:execute` is useful for two tasks that have no common dependencies.

Fetching a Recipe

So far we assumed the included recipes were stored on the local system. It is also possible to obtain them from elsewhere. The example with children above can be extended like this:

```
ORIGIN = ftp://ftp.foo.org/recipes
:child core/main.aap {fetch = $ORIGIN/core.aap}
:child util/main.aap {fetch = $ORIGIN/util.aap}
:include common.aap {fetch = $ORIGIN/common.aap}

all: theprog$EXESUF

theprog$EXESUF : core/$*CORE_OBJ util/$*UTIL_OBJ
    :do build $source
```

The `fetch` attribute is used to specify the URL where the recipe can be obtained from. This works just like fetching source files. Notice in the example that the file name in the URL can be different from the local file name. When Aap reads this recipe and discovers that a child or included recipe does not exist, it will use the `fetch` attribute to download it. The `fetch` attribute can also be used with the `:execute` command.

Once a recipe exists locally it will be used, even when the remote version has been updated. If you explicitly want to get the latest version of the recipes used, run **`aap -R`** or **`aap fetch`**.

Chapter 10. Commands in a Pipe

A selection of commands can be connected together with a pipe. This means the output of one command is the input for the next command. It is useful for filtering text from a variable or file and writing the result in a variable or file.

Changing a timestamp

This example shows how you can change the timestamp in a file. It is done in-place.

```
all:
    :print Setting date in foobar.txt.
    :cat foobar.txt
          | :eval re.sub('Last Change: .*\n', 'Last Change: ' + DATESTR + '\n', stdin)
          >! foobar.txt
```

Lets see how this works:

```
% cat foobar.txt
This is example text for the A-A-P tutorial.
Last Change: 2002 Feb 29
The useful contents would start here.
% aap
Setting date in foobar.txt.
% cat foobar.txt
This is example text for the A-A-P tutorial.
Last Change: 2002 Oct 21
The useful contents would start here.
%
```

The last command in the example consists of three parts. First comes the `:cat` command. It reads the "foobar.txt" file and passes it throught the pipe to the next command. "cat" is short for "concatenate". This is one of the good-old Unix commands that actually does much more than the name suggests. In this example nothing is concatenated. Below you will see examples where it does.

The second part of the example is the `:eval` command. This is used to read the text coming in through the pipe and modify it with a Python expression. In this case the expression is a "re.sub()" function call. This Python function takes three arguments: A pattern, a replacement string and the text to operate on. All occurences of the pattern in the text are changed to the replacement string. The pattern "Last Change: .*\n" matches a line with the date that was inserted previously. The replacement string contains `DATESTR`, which is an Aap variable that contains today's date as a string, e.g., "2002 Oct 19". The text to operate on is `stdin`. This is the variable that holds the text that is coming in through the pipe.

The third and last part `>! foobar.txt` redirects the output of the `:eval` command back to the file "foobar.txt". Using just ">" would cause an error, since the file already exists.

Note that in a Unix shell command this pipe would not work: The "foobar.txt" would be overwritten before it was read. In Aap this does not happen, the commands in the pipe are executed one by one. That makes it easier to use, but it does mean the text is kept in memory. Don't use pipes for a file that is bigger than half the memory you have available.

Changing a file in-place has the disadvantage that the normal dependencies don’t work, since there is no separate source and target file. Often it is better to use a file "foobar.txt.in" as source, change it like in the example above and write it as a new file. The recipe would be:

```
    foobar.txt: foobar.txt.in
        :print Setting date in $target.
        :cat $source
        | :eval re.sub(’Last Change: .*\n’, ’Last Change: ’ + DATESTR + ’\n’, stdin)
>! $target
```

Creating a file from pieces

Sometimes you need to generate a file from several pieces. Here is an example that concatenates two files and puts a generated text line in between.

```
    manual.html: body.html footer.html
        @import time
        :eval time.strftime("%A %d %B %Y", time.localtime(time.time()))
                 | :print $(lt)BR$(gt)Last updated: $stdin$br
                 | :cat body.html - footer.html >! $target
```

There are quite a few items here that need to be explained. First of all, the "@import time" line. This is a Python command to load the "time" module. So far we used modules that Aap has already loaded for you. This one isn’t, and since we use the "time" module in the next `:eval` command it needs to be loaded explicitly.

The Python function "strftime()" formats the date and time in a specified format. See the Python documentation for the details. In this case the resulting string looks like "Monday 21 October 2002".

The output of the `:eval` command is piped into a `:print` command. The variable `stdin` contains the output of the previous command. Note that "$(lt)" is used instead of "$lt". The meaning is exactly the same: the value of the `lt` variable. Without the extra parenthesis it would read "$ltBR", which would be the value of the "ltBR" variable.

The resulting text is:

<BR>Last updated: Monday 21 October 2002\n

Note that the first "BR" is the HTML code for a line break, while the "$br" at the end is the Aap variable that contains a line break (here displayed as "\n").

Finally, the `:cat` command concatenates the file "body.html", the output of the `:print` command and the file "footer.html". Thus the "-" stands for where the pipe input is used. The result is redirected to `target`, which is "manual.html".

Pipe output in a variable

The generated date in the previous example could be used elsewhere in the recipe. Since we don’t want to repeat a complicated expression the result of the `:eval` command should be redirected to a variable, like this:

```
    @import time
    :eval time.strftime("%A %d %B %Y", time.localtime(time.time()))
| :assign DATESTAMP

    manual.html: body.html footer.html
           :print $(lt)BR$(gt)Last updated: $DATESTAMP$br
                 | :cat body.html - footer.html >! $target
```

The `:assign` command takes the input from the pipe and puts it in the variable mentioned as its argument, which is "DATESTAMP" here. Actually, the same can be done with a normal assignment and a Python expression in backticks, but we intentionally wanted to show using a pipe here.

Creating a file from scratch

It is also possible to completely generate a file from scratch. Here is an example that generates a C header file:

```
1    :include config.aap
2    pathdef.c: config.aap
3       :print Creating $target
4       :print >! $target /* pathdef.c */
5       :print >> $target /* This file is automatically created by main.aap */
6       :print >> $target /* DO NOT EDIT!  Change main.aap only. */
7       :print >> $target $#include "vim.h"
8       :print >> $target char_u *default_vim_dir = (char_u *)"$VIMRCLOC";
9       :print >> $target char_u *all_cflags = (char_u *)"$CC -c -I$srcdir $CFLAGS";
```

The first `:print` command displays a message, so that it's clear "pathdef.c" is being generated. The next line contains ">!" to overwrite an existing file. It doesn't matter if the file already existed or not, it now only contains the line "/* pathdef.c */". The third and following lines contain ">>". This will cause each line to be appended to "pathdef.c".

In the example the `VIMRCLOC` and `srcdir` variables are defined in the recipe "config.aap". That is why this file is used as a source in the dependency. Also note the use of "$#" in line 7. Since "#" normally starts a comment it cannot be used directly here. "$#" is a special item that results in a "#" in the `:print` output. This is the resulting file:

```
/* pathdef.c */
/* This file is automatically created by main.aap */
/* DO NOT EDIT!  Change main.aap only. */
#include "vim.h"
char_u *default_vim_dir = (char_u *)"/usr/local/share/vim61";
char_u *all_cflags = (char_u *)"cc -c -I. -g -O2";
```

The list of ">>" redirections is quite verbose. Fortunately there is a shorter way:

```
1    :include config.aap
2    pathdef.c: config.aap
3        :print Creating $target
4        text << EOF
5            /* pathdef.c */
6            /* This file is automatically created by main.aap */
```

```
7            /* DO NOT EDIT!  Change main.aap only. */
8            $#include "vim.h"
9            char_u *default_vim_dir = (char_u *)"$VIMRCLOC";
10           char_u *all_cflags = (char_u *)"$CC -c -I$srcdir $CFLAGS";
11               EOF
12       :print $text >! $target
```

In line 4 "text << EOF" is used. This is called a block assignment. The following lines, up to the matching "EOF" line, are assigned to the variable `text`. You can use something else than "EOF" if you want to. It must be a word that does not appear inside of the text as a line on its own. White space before and after the word is ignored.

The indent of the text in the block assignment is removed. The indent of the first line is used, the same amount of indent is removed from the following lines. Thus if the second line has two more spaces worth of indent than the first line, it will have an indent of two spaces in the result. Half a tab is replace with four spaces when necessary (a tab always counts for up to eight spaces).

Chapter 11. A Ported Application

When an application already exists but for your system it requires a few tweaks, a port recipe can do the work. This can also be used for applications that work fine but you want to apply a number of patches or to add a feature. The recipe can be distributed, so that others can install the application without knowing the details. This works very much like the FreeBSD ports system.

The Port Recipe

Since A-A-P is prepared for doing all the work, usually you only need to specify the relevant information, such as where to find the files involved. Here is an example:

```
# A-A-P port recipe for Vim 6.1 plus a few patches.
AAPVERSION =        1.0

PORTNAME =          vim
LASTPATCH =         003
PORTVERSION =       6.1.$LASTPATCH
MAINTAINER =        Bram@vim.org

CATEGORIES =        editors
PORTCOMMENT =       Vim - Vi IMproved, the text editor
PORTDESCR << EOF
This is the description for the Vim package.
A very nice editor, backwards compatible to Vi.
You can find all info on http://www.vim.org.
        EOF

:recipe {fetch = http://www.a-a-p.org/ports/vim/main.aap}

WRKSRC =            vim61
BUILDCMD =          make
TESTCMD =           make test
INSTALLCMD =        make install DESTDIR=$PKGDIR
PREFIX =            /usr/local

MASTER_SITES ?=     ftp://ftp.vim.org/pub/vim
                    ftp://ftp.us.vim.org/pub/vim
PATCH_SITES =       $*MASTER_SITES/patches

DISTFILES =         unix/vim-6.1.tar.bz2

version1 =          ‘range(1, int(LASTPATCH) + 1)‘
PATCHFILES =        6.1.00$*version1

#>>> automatically inserted by "aap makesum" <<<
do-checksum:
        :checksum $DISTDIR/vim-6.1.tar.bz2 {md5 = 7fd0f915adc7c0dab89772884268b030}
        :checksum $PATCHDISTDIR/6.1.001 {md5 = 97bdbe371953b9d25f006f8b58b53532}
        :checksum $PATCHDISTDIR/6.1.002 {md5 = f56455248658f019dcf3e2a56a470080}
        :checksum $PATCHDISTDIR/6.1.003 {md5 = 0e000edba66562473a5f1e9b5b269bb8}
```

```
40    #>>> end <<<
```

Well, that is the longest example we have had so far. Let’s go through it from top to bottom.

```
 1    # A-A-P port recipe for Vim 6.1 plus a few patches.
 2    AAPVERSION =        1.0
```

`AAPVERSION` tells Aap what version of Aap this recipe was written for. If in the future the recipe format changes, this line causes Aap to interpret it as Aap version 1.0 would do.

```
 4    PORTNAME =          vim
```

Setting `PORTNAME` to the name of the port is what actually triggers Aap to read this recipe as a port recipe. It makes the other settings to be used to set up a whole range of targets and build commands. The result is that you can do **`aap install`** to install the application, for example. Note that `PORTNAME` does not include the version number.

```
 5    LASTPATCH =         003
 6    PORTVERSION =       6.1.$LASTPATCH
 7    MAINTAINER =        Bram@vim.org
 8
 9    CATEGORIES =        editors
10    PORTCOMMENT =       Vim - Vi IMproved, the text editor
11    PORTDESCR << EOF
12    This is the description for the Vim package.
13    A very nice editor, backwards compatible to Vi.
14    You can find all info on http://www.vim.org.
15            EOF
```

In lines 5 to 15 a number of informative items about the port are specified. These are used in various places. `LASTPATCH` is not a standard item, it is used here to only have to define the patchlevel in one place.

```
17    :recipe {fetch = http://www.a-a-p.org/ports/vim/main.aap}
```

The `:recipe` command specifies where to obtain the recipe itself from. We have seen this before, nothing special here.

```
19    WRKSRC =            vim61
20    BUILDCMD =          make
21    TESTCMD =           make test
```

The assignments in lines 19 to 21 specify how building is to be done. `WRKSRC` is the directory below which the source files are unpacked. The default is "$PORTNAME-$PORTVERSION". The archive used for Vim uses "vim61" instead, thus this needs to be specified. The "CMD" variables set the commands to be used to build the application. The default is to use Aap. Since Vim uses "make" this needs to be specified.

```
22    INSTALLCMD =        make install DESTDIR=$PKGDIR
23    PREFIX =            /usr/local
```

Installing a port is done by creating a binary package and installing that package. This makes it possible to copy the package to another system and install it there without the need to compile from sources.

Lines 22 and 23 specify how to do a "fake install" with Vim. This copies all the files that are to be installed to a specific directory, so that it is easy to include them in the package. `PREFIX` specifies below which directory Vim installs its files.

```
25    MASTER_SITES ?=     ftp://ftp.vim.org/pub/vim
26                        ftp://ftp.us.vim.org/pub/vim
27    PATCH_SITES =       $*MASTER_SITES/patches
```

`MASTER_SITES` and `PATCH_SITES` specify the sites where the Vim files can be downloaded from. The first is for the archives, the second for the patches. Note the use of "$*" in line 27, this causes "/patches" to be appended to each item in `MASTER_SITES` instead of appending it once at the end of the whole list.

```
29    DISTFILES =         unix/vim-6.1.tar.bz2
```

`DISTFILES` is set to the name of the archive to download. This is appended to items in `MASTER_SITES` to form the URL.

```
31    version1 =          ‘range(1, int(LASTPATCH) + 1)‘
32    PATCHFILES =        6.1.00$*version1
```

Lines 32 and 33 specify the list of patch file names. The Python function "range()" is used, this returns a list of numbers in the specified range (up to and excluding the upper number). Note the user of "int()" to turn the patch number in `LASTPATCH` into an int type, all Aap variables are strings.

for three patch files this could also have been typed, but when the number of patches grows this mechanism is easier. The example only works up to patch number 009. To make it work for numbers from 100 up to 999:

```
      version1 =          ‘range(1, 10)‘
      version2 =          ‘range(10, 100)‘
      version3 =          ‘range(100, int(LASTPATCH) + 1)‘
      PATCHFILES =        6.1.00$*version1  6.1.0$*version2  6.1.$*version3
```

```
34    #>>> automatically inserted by "aap makesum" <<<
35    do-checksum:
36            :checksum $DISTDIR/vim-6.1.tar.bz2 {md5 = 7fd0f915adc7c0dab89772884268b030}
37            :checksum $PATCHDISTDIR/6.1.001 {md5 = 97bdbe371953b9d25f006f8b58b53532}
38            :checksum $PATCHDISTDIR/6.1.002 {md5 = f56455248658f019dcf3e2a56a470080}
39            :checksum $PATCHDISTDIR/6.1.003 {md5 = 0e000edba66562473a5f1e9b5b269bb8}
40    #>>> end <<<
```

Finally the "do-checksum" target is defined. This part was not typed, but added to the recipe with **`aap makesum`**. This is done by the port recipe maintainer, when he has verified that the files are correct. When a user later uses the recipe Aap will check that the checksums match, so that problems with downloading or a cracked distribution file are found and reported.

Using CVS

The port recipe specifies which source files and patches to download, thus it has to be adjusted for each version. This is good for a stable release, but when you are releasing a new version every day it is a lot of work. Another method is possible when the files are available from a CVS server. Adding these lines to the recipe will do it:

```
CVSROOT ?=        :pserver:anonymous@cvs.vim.sf.net:/cvsroot/vim
CVSMODULES =      vim
CVSTAG =          vim-6-1-$LASTPATCH
```

The first line specifies the cvsroot to use. This is specific for the cvs program. `CVSMODULES` is the name of the module to checkout. Mostly it is just one name, but you can specify several. Specifying `CVSTAG` is optional. If it is defined, like here, a specific version of the application is obtained. When it is omitted the latest version is obtained.

II. User Manual

Chapter 12. Dependencies, Rules and Actions

Automatic dependency checking

When a source file includes other files, the targets that depend on it need to be rebuild. Thus when "foo.c" includes "foo.h" and "foo.h" is changed, the build commands to produce "foo.o" from "foo.c" must be executed, even though "foo.c" itself didn't change.

A-A-P detects these implied dependencies automatically for the types it knows about. Currently that is C and C++. For other types of files you can add your own dependency checker. For example, this is how to define a checker for the "tt" file type:

```
   foo : foo.tt
:sys tt_compiler $source > $target
   :autodepend tt
    :sys tt_checker $source > $target
```

The "tt_checker" command reads the file "$source" and writes a dependency line in the file "$target". This is a dependency like it is used in a recipe. In a makefile this has the same syntax, thus tools that produce dependencies for "make" will work. Here is an example:

```
   foo.o : foo.tt  foo.hh  include/common.hh
```

This is interpreted as a dependency on "foo.hh" and "include/common.hh". Note that "foo.o" and "foo.tt" are ignored. Tools designed for "make" produce these but they are irrelevant for A-A-P.

Since the build commands for ":autodepend" are ordinary build commands, you can use Python commands, system commands or a mix of both to do the dependency checking.

Multiple targets

When a dependency has more than one target, this means that the build commands will produce all these targets. This makes it possible to specify build commands that produce several files at the same time. Example that compiles a file and at the same time produces a documentation file:

```
   foo.o foo.html : foo.src
:sys srcit $source -o $(target[0]) --html $(target[1])
```

People used to "make" must be careful, they might expect the build commands to be executed once for each target. A-A-P doesn't work that way, because the above example would be impossible. To run commands on each target this must be explicitly specified. Example:

```
   dir1 dir2 dir3 :
    @for item in target_list:
     :mkdir $item
```

The variable "target_list" is a Python list of the target items. "source_list" is the list of source items. An extreme example of executing build commands for each combination of sources and targets:

```
   $OUTPUTFILES : $INPUTFILES
    @for trg in target_list:
```

```
    :print start of file  >! $trg
    @for src in source_list:
:sys foofilter -D$trg $src >> $trg
```

```
RECIPE STRUCTURE

Generally there are these sections in a recipe used for building an
application:

1. global settings, include recipes with (project, user) settings
2. automatic configuration
3. specify variants
4. generic build rules
5. explicit dependencies

For larger projects sections can be moved to other recipes.

1. global settings, include recipes with (project, user) settings
When the recipe is part of a project, it’s often useful to move
settings (and rules) that apply to the whole project to one file.
User preferences (e.g. configuration choices) should be in a separate
file that the user edits (using a template)
2. automatic configuration
Find out properties of the system, and handle user preferences.  This
may result in building the application in a different way.
3. specify variants
Usually debug and release, but can include many more choices (type of
GUI, small or big builds, etc.)
4. generic dependencies and build commands
Rules that define dependencies and build commands that apply to
several files.
5. explicit dependencies and build commands
Dependencies and build commands that apply to specific files.


RECIPE EXECUTING:

This is done in these steps:
1. Read the startup recipes, these define default rules and variables.  These
   recipes are used:
   - "default.aap" from the distribution
   - all recipes matching "/usr/local/share/aap/startup/*.aap"
   - all recipes matching "~/.aap/startup/*.aap"
2. Read the recipe and check for obvious errors.
3. Execute the toplevel items in the recipe.  Dependencies and rules are
   stored.
4. Apply the clever stuff to add missing dependencies and rules.
5. Update the targets.  The first of the following that exists is used:
- targets specified on the command line
- the items in $TARGET
- the "all" target
6. If the "finally" target is specified, execute its build commands.
```

When there is one item in $TARGET, no dependency for $TARGET is specified and $SOURCE is not empty, a dependency is automatically generated. This will have the form:

```
$TARGET : $SOURCE_OBJ
:sys $CC $CFLAGS $LDFLAGS -o $target $source
```

Where $SOURCE_OBJ is $SOURCE with all items changed to use $OBJSUF, if there is a suffix. Example:

```
TARGET = foo
SOURCE = main.c extra.p
```

Results in:

```
foo : main$OBJSUF extra$OBJSUF
:sys $CC $CFLAGS $LDFLAGS -o $target $source
```

This then uses the default rules to make "main$OBJSUF" from main.c and extra$OBJSUF from extra.p.

DEPENDENCIES:

A dependency can have several targets. How this is interpreted depends on whether commands are defined.

Without commands the dependency is used like each target depends on the list of source files. Thus this dependency:

```
t1 t2 : s1 s2 s3
```

Can be rewritten as:

```
t1 : s1 s2 s3
t2 : s1 s2 s3
```

Thus when t1 is outdated to s1, s2 or s3, this has no consequence for t2.

When commands are specified, the commands are expected to produce all these targets. Example:

```
t1 t2 : s1 s2 s3
:sys produce s1 s2 s3  > t1
:sys filter < t1  > t2
```

When either t1 or t2 is outdated relative to s1, s2 or s3, the commands are executed and both t1 and t2 are updated.

Only when no dependency is specified for a target, the rules defined with ":rule" are checked. All the matching rules without commands are used. TRICK: For the rules with commands, only the matching one with the longest

pattern is used. If there are two with an equally long pattern, this is an error.
TRICK: When the source and target in a rule are equal, it is skipped. This avoids that a rule like this becomes cyclic:

```
:rule %.jpg : path/%.jpg
:copy $source $target
```

ATTRIBUTES OVERRULING VARIABLES

Most variables like $CFLAGS and $BDIR are used for all source files. Sometimes it is useful to use a different value for a group of files. This is done with an attribute that starts with "var_". What follows is the name of the variable to be overruled. Thus attribute "var_XYZ" overrules variable "XYZ".

The overruling is done for:
- dependencies
- rules
- actions
- the default dependency added for $SOURCE and $TARGET

The attributes of all the sources are used. In case the same attribute is used twice, the last one wins.

VIRTUAL TARGETS

When a target is virtual it is always build. A-A-P does not remember if it was already done a previous time. However, it is only build once for an invocation of Aap. Ex
clean:

```
:del {r}{f} temp/*
```

To remember the signatures for a virtual target use the "remember" attribute:

```
version {virtual}{remember} : version.txt.in
:print $VERSION | :cat - $source >! version.txt
```

Now "aap version" will only execute the ":print" command if version.txt.in has changed since the last time this was done.

Using {remember} for one of the known virtual targets (e.g., "all" or "fetch") is unusual, except for "publish".

When using {remember} for a virtual target without a dependency, it will only be build once. This can be used to remember the date of the first invocation.

```
all: firsttime
firsttime {virtual}{remember}:
:print First build on $DATESTR > firstbuild.txt
```

The difference with a direct dependency on "firstbuild.txt" is that when this file is deleted, it won't be build again.

SOURCE PATH

The sources for a dependency are searched for in the directories specified with $SRCPATH. The default is ". $BDIR", which means that the sources are searched for in the current directory and in the build directory.

The "srcpath" attribute overrules using $SRCPATH for an item.
To avoid using $SRCPATH for a source, make the "srcpath: attribute empty:

```
foo.o : src/foo.c {srcpath=}
```

When setting $SRCPATH to include the value of other variables, you may want to use "$=", so that the value of the variable is not expanded right away but when $SRCPATH is used. This is especially important when appending to $SRCPATH before a ":variant’ command, since it changes $BDIR. Example:

```
SRCPATH $+= include
```

Warning: Using the search path means that the first encountered file will be used. When old files are lying around the wrong file may be picked up. Use the full path to avoid this.

When a target depends on the existence of a directory, it can be specified this way:

```
foodir/foo : foodir {directory}
:print >$target this is foo
```

The directory will be created if it doesn’t exist. The normal mode will be used (0777 with umask applied). When a different mode is required specify it with an octal value: {directory = 0700}. The number must start with a zero.

CROSS REFERENCER

A separate module in the A-A-P project is the cross referencer. It can be used to lookup where symbols are defined and used. The database that the cross referencer uses can be generated with a recipe. That makes it possible to put the right files in the database.

The standard way to create or update the database is:

```
aap reference
```

The recipe can specify a "reference" target to implement it. Example:

```
SOURCE = main.c version.c
INCLUDE = common.h

...
```

```
reference:
:sys aref -u $SOURCE $INCLUDE
```

The advantage over letting the cross referencer use all files is that backup files and old versions are skipped, and files in other locations can be included.

When there is no "reference" target, the default is to execute this command:

```
:do reference $SOURCE $?INCLUDE
```

The default for the "reference" action is to run "aref -u", thus this does almost the same as the example.

BUILD COMMAND SIGNATURE

A special kind of signature is used to check if the build commands have changed. An example:

```
foo.o : {buildcheck = $CFLAGS} foo.c
:sys $CC $CFLAGS -c $source -o $target
```

This defines a check for the value of $CFLAGS. When the value changes, the build commands will be executed.

The default buildcheck is made from the command themselves. This is with variables expanded before the commands have been executed. Thus when one of the commands is ":sys $CC $CFLAGS $source" and $CC or $CFLAGS changes, the buildcheck signature changes. The ":do" commands are also expanded into the commands for the action specified. However, this only works when the action and filetype can be estimated. The action must be specified plain, not with a variable, and the filetype used is the first of:
1. a filetype attribute specified after action
2. if the first argument doesn’t contain a "$", the file type of this argument
3. the filetype of the first source argument of the dependency.

To combine the check for the build commands themselves with specific variable values the $commands variable can be used:

```
VERSION = 1.4
foo.txt : {buildcheck = $commands $VERSION}
:del {force} $target
:print  >$target this is $target
:print >>$target version number: $VERSION
```

If you now change the value of $VERSION, change one of the ":print" commands or add one, "foo.txt" will be rebuild.

To simplify this, $xcommands can be used to check the build commands after expanding variables, thus you don’t need to specify $VERSION:

```
foo.txt : {buildcheck = $xcommands}
```

However, this only works when all $VAR in the commands can be expanded and variables used in python commands are not expanded.

To avoid checking the build commands, use an empty buildcheck. This is useful when you only want the target to exist and don’t care about the command used to create it:

```
objects : {buildcheck = }
:print "empty" > objects
```

Chapter 13. Variants

This is the first paragraph of Variants.

Here is an example how build variants can be specified:

```
:variant OPT
some
    CFLAGS = -O2
much  compiler == "gcc"
    CFLAGS = -O6
*
    CFLAGS = -O
```

"OPT" is the name of a variable. It is used to select one of the variants. Each possible value is listed in the following line and further lines with the same indent. In the example these are "some" and "much". "*" is used to accept any value, it must be the last one. The first value mentioned is the default when the variable isn’t set.

The $BDIR variable will be adjusted for the variant used. CAREFUL: this means that using $BDIR before ":variant" commands will use a different value, that might not always be what you want.

Inside the ":variant" command the value of $BDIR has already been adjusted.

When a target that is being build starts with $BDIR and $BDIR doesn’t exist, it is created.

$BDIR is relative to the recipe. When using ":child dir/main.aap" the child recipe will use a different build directory "dir/$BDIR".
Note that when building the same source file twice from recipes that are in different directories, you will get two results.

Chapter 14. Publishing

Publishing means distributing a version of your project and giving it a version number.

```
:publish file ...
```

This uses the "publish" attribute on each of the files. When it is missing the "commit" attribute is used. If both are missing this is an error.

When a file didn’t change since the last time it was published, it won’t be published again. This works with signatures, like building a target. The remote file is the target in this case.

To publish all files with a "publish" attribute use the command:

```
aap publish
```

If the "publish’ target is defined explicitly it will be executed. Otherwise, all files with the "publish" attribute are given to the ":publish" command, just like using ":publishall".

```
:publishall
:publishall {attr = val}
```

Publish all the files in the recipe (and child recipes) that
have the "publish" attribute and changed since the last time
they were published.
Note that this doesn’t fall back to the "commit" attribute
like ":publish" does.

Publishing will use all the items in the "publish" attribute. This means a file can be distributed to several sites at once. This is unlike fetching, which stops as soon as the file could be obtained from one of the items. When publishing fails for one of the sites, it is considered to have failed for all sites. When you publish again, uploading is done to all of the mentioned sites again (this may change in a future version).

DISTRIBUTING GENERATED FILES

When publishing a new version of an application, you might want to include a number of generated files. This reduces the number of tools someone needs to use when installing the application. For example, the "configure" script produced by "autoconf" from "configure.in" can be included.

To avoid these generated results to be generated again when the user runs aap, specify a special signature file and distribute this signature file together with the generated files. For example, suppose you have a directory with these files you created:

```
main.aap
```

```
prog.c
```

Additionally there is the file "version.c" that is generated by the "main.aap" recipe. It contains the date of last modification. To avoid it being generated again, include the "mysign" file in the distribution. The files to be distributed are:

```
mysign
main.aap
prog.c
version.c
```

In the "main.aap" recipe the "signfile" attribute is used for the dependency that generates version.c:

```
version.c {signfile = mysign} : prog.c
:print char *timestamp = "$TIMESTR"; >! $target
```

The result is that "version.c" will only be generated when "prog.c" has changed. When the "signfile" attribute would not have been used, "version.c" would have been generated after unpacking the distributed files.

FINALLY

The "finally" target is always executed after the other targets have been successfully build. Here is an example that uses the "finally" target to copy all files that need to be uploaded with one command.

```
SOURCE = ‘glob("*.html")‘
TARGET = remote/$*SOURCE
CFILE =
:rule remote/% : %
CFILE += $source
:export CFILE

finally:
@if CFILE:
:copy $CFILE ftp://my.org/www
```

Warning: When the ":copy" command fails, aap doesn’t know the targets were not made properly and won’t do it again next time. Using the "publish" attribute works better.

Chapter 15. Fetching

This is the first paragraph of Fetching.

```
FETCHING AND UPDATING:

A convention about using the "update" and "fetch" targets makes it easy for
users to know how to use a recipe.  The main recipe for a project should be
able to be used in three ways:

1. Without specifying a target.
This should build the program in the usual way.  Files with an
"fetch" attribute are obtained when they are missing.
2. With the "fetch" target.
This should obtain the latest version of all the files for the
program, without building the program.
3. With the "update" target.
This should fetch all the files for the program and then build it.
It’s like the previous two ways combined.

Here is an example of a recipe that works this way:

STATUS = status.txt
SOURCE = main.c version.c
INCLUDE = common.h
TARGET = myprog

$TARGET : $SOURCE
:cat $STATUS
:sys $CC $CFLAGS $LDFLAGS -o $target $source
$TARGET : $STATUS

# specify where to fetch the files from
:attr {fetch = cvs://:pserver:anonymous@cvs.myproject.sf.net:/cvsroot/myproject} $SOURCE $I
:attr {fetch = ftp://ftp.myproject.org/pub/%file%} $STATUS

When the "fetch" target is not specified in the recipe or its children, it
is automatically generated.  Its build commands will fetch all nodes with
the "fetch" attribute, except ones with a "constant" attribute set
(non-empty non-zero).  To do the same manually:

fetch:
:fetch $SOURCE $INCLUDE $STATUS

NOTE: When any child recipe defines a "fetch" target no automatic fetching
is done for any of the recipes.  This may not be what you expect.

When there is no "update" target it is automatically generated.  It will
invoke the "fetch" target and the first target in the recipe.  To do
something similar manually:

update: fetch $TARGET
```

```
Although the automatically generated "update" target uses one of $TARGET,
"all" or the first encountered dependency.

:fetchall
:fetchall {attr = val}
Fetch all the files in the recipe (and child recipes) that
have the "fetch" attribute.
Extra attributes for fetching can be specified here, they
overrule the attributes of the file itself.

aap fetch Does a ":fetchall".



THE FETCH ATTRIBUTE

The "fetch" attribute is used to specify a list of locations where the file
can be fetched from.  The word at the start defines the method used to
fetch the file:
ftp from ftp server
http from http (www) server
scp secure copy
file local file system
cvs from CVS repository
For a module that was already checked out the part
after "cvs://" may be empty, CVS will then use the
same server (CVSROOT) as when the checkout was done.
other  user defined

These kinds of locations can be used:

ftp://ftp.server.name/full/path/file
http://www.server.name/path/file
scp://host.name/path:path/file
cvs://:METHOD:[[USER][:PASSWORD]@]HOSTNAME[:[PORT]]/path/to/repository
file:~user/dir/file
file:///etc/fstab

For a local file there are two possibilities: using "file://" or "file:".
They both have the same meaning.  Note that for an absolute path, relative to
the root of the file system, you use either one or three slashes, but not two.
Thus "file:/etc/fstab" and "file:///etc/fstab" are the file "/etc/fstab".  A
relative path has two or no slashes, but keep in mind that moving the recipe
will make it invalid.  You can also use "file:~/file" or "file://~/file" for a
file in your own home directory, and "file:~jan/file" or "file://~jan/file"
for a file in the home directory of user "jan".

To add a new method, define a Python function with the name "fetch_method",
where "method" is the word at the start.  The function will be called with two
arguments:
name the url with "method://" removed
dict a dictionary with all attributes used of the URL,
dict["name"] is the full URL
node a node object.  Useful items:
   node.name short name
   node.absname full name of the file
   node.recipe_dir directory in which node.name
is valid
   node.attributes  dictionary with attributes
The function should return a non-zero number for success, zero for failure.
Here is an example:

    :python
def fetch_foo(from_name, dict, node):
    to_name = node.absname
    try:
foo_the_file(from_name, to_name, dict["foo_option"])
    except:
     return 0
    return 1

In the "fetch" attribute the string "%file%" can be used where the path of
the local target is to be inserted.  This is useful when several files have a
common directory.  Similarly "%basename%" can be used when the last item in the
path is to be used.  This removes the path from the local file name, thus can
be used when the remote directory is called differently and only the file name
is the same.  Examples:

      :attr {fetch = ftp://ftp.foo.org/pub/foo/%file%} src/include/bar.h

Gets the file "src/include/bar.h" from
"ftp://ftp.foo.org/pub/foo/src/include/bar.h".

      :attr {fetch = ftp://ftp.foo.org/pub/foo/src-2.0/include%basename%}
  src/include/bar.h

Gets the file "src/include/bar.h" from
"ftp://ftp.foo.org/pub/foo/src-2.0/include/bar.h".


CACHING:

Remote files are downloaded when used.  This can take quite a bit of time.
Therefore downloaded files are cached and only downloaded again when outdated.

The cache can be spread over several directories.  The list is specified
with the $CACHE variable.

NOTE: Using a global, writable directory makes it possible to share the cache
with other users, but only do this when you trust everybody who can login to
the system!

A cached file becomes outdated as specified with the "cache_update" attribute
or $cache_update.  The value is a number and a name.  Possible values for the
name:
```

```
day number specifies days
hour number specifies hours
min number specifies minutes
sec number specifies seconds
The default is "12 hour".

When a file becomes outdated, its timestamp is obtained.  When it differs
from when the file was last downloaded, the file is downloaded again.  When
the file changes but doesn’t get a new timestamp this will not be noticed.

When fetching files the cached files are not used (but may be updated).
```

Chapter 16. Version Control

```
The generic form of version control commands is:

:command file ...

Or:

:command {attr = val} ... file ...

The commands use the "commit" attribute to define the kind of version control
system and its location.  For example:

{commit = cvs://:ext:$CVSUSER_AAP@cvs.a-a-p.sf.net:/cvsroot/a-a-p}

These commands can be used:

:commit Update the repository for each file that was changed.
This is also done for a file that didn’t change, it’s up to
the version control software to check for an unchanged file
(it might have been changed in the repository).
Do checkin/checkout when checkin is required.
Don’t change locking of the file.
Uses "logentry" attribute when a log entry is to be done.
Otherwise a standard message is used.
Adds new files when needed.
Creates directories when needed (CVS: only one level).

:checkout Like fetch and additionally lock for editing when possible.

:checkin Like commit, but unlock file.

:unlock Remove lock on file, don’t change file in repository or
locally.

:add Add file to repository.  File must exist locally.  Implies a
"commit" of the file.

:remove Remove file from repository.  File may exist locally.  Implies
a "commit" of the file.

:tag Add a tag to the current version.  Uses the "tag" attribute.

Additionally, there is the generic command:

:verscont action {attr = val} ... file ...

This calls the Version control module as specified in the "commit" attribute
for "action" with the supplied arguments.  What happens is specific for the
VCS.
```

```
These commands work on all the files mentioned that have the "commit"
attribute:

:checkoutall
:checkoutall {attr = val}
Checkout all files in the recipe (and child recipes) that have
the "commit" attribute.  Locks the files.

:commitall
:commitall {attr = val}
Commit all the files in the recipe (and child recipes) that
have the "commit" attribute.  Files missing in the VCS will be
added.  No files will be removed.

:checkinall
:checkinall {attr = val}
Just like :commitall and also remove any locks.

:unlockall
:unlockall {attr = val}
Unlock all files in the recipe (and child recipes) that have
the "commit" attribute.

:addall [options] [directory] ...
:addall [options] {attr = val} ... [directory] ...
Inspect directories and add items that do not exist in the VCS
but are mentioned in the recipe(s) with a "commit" attribute.

OPTIONS:
{l} {local} don’t do current directory recursively
{r} {recursive} do handle arguments recursively

When no directory argument is given, the current directory is
used.  It is inspected recursively, unless the "{local}"
option was given.
When directory arguments are given, each directory is
inspected.  Recursively when the "{recursive}" option was
given.
When no "commit" attribute is specified here, it will be
obtained from any node.

:removeall [options] [directory] ...
:removeall [options] {attr = val} ... [directory] ...
Inspect directories and remove items that do exist in the VCS
but are not mentioned or do not have a "commit" attribute.
Careful: Only use this command when it is certain that all
files that should be in the VCS are explicitly mentioned and
do have a "commit" attribute!

OPTIONS:
{l} {local} don’t do current directory recursively
{r} {recursive} do handle arguments recursively
```

```
When no directory argument is given, the current directory is
used.  It is inspected recursively, unless the "{local}"
option was given.
When directory arguments are given, each directory is
inspected.  Recursively when the "{recursive}" option was
given.
When no "commit" attribute is specified here, it will be
obtained from any node.

:reviseall
:reviseall {attr = val}
Just like:
:checkinall {attr = val}
:removeall {attr = val} .

:tagall {tag = name}
Add a tag to all items with a "commit" and "tag" attribute.
The tag should be simple name without special characters (no
dot or dash).

Related to these commands are targets that are handled automatically when not
defined explicitly.  When defining a target for these, it would be highly
unexpected if it works differently.

aap commit Normally uses the files you currently have to update the
version control system.  This can be used after you are done
making a change.  Default is using ":commitall".

aap checkout Update all files from the VCS that have a "commit" attribute.
When the VCS supports locking all files will be locked.
Without locking this does the same as "aap fetch".

aap checkin Do ":checkin" for all files that have been checked out of the
VCS.  For a VCS that doesn’t do file locking this is the same
as "aap commit".

aap unlock Unlock all files that are locked in the VCS.  Doesn’t change
any files in the VCS or locally.

aap add Do ":add" for all files that appear in the recipe with a
"commit" attribute that do not appear in the VCS.

aap remove Do ":removeall": remove all files that appear in the VCS but
do not exist in the recipe with a "commit" attribute or do not
exist in the local directory.  This works in the current
directory and recursively enters all subdirectories.
Careful: files with a search path may be accidentally removed!

aap tag Do ":tagall": tag all files with a "commit" and "tag"
attribute.  The tag name should be defined properly in the
recipe, although "aap tag TAG=name" can be used if the recipe
contains something like:
:attr {tag = $TAG} $FILES
```

aap revise Same as "aap checkin" followed by "aap remove": checkin all changed files, unlock all files and remove files that don’t have the "commit" attribute.

For the above the "-l" or "--local" command line option can be used to restrict the operation to the directory of the recipe and not recurse into subdirectories.

When it’s desired to commit one directory at a time the following construct can be used:

```
SOURCE = ‘glob("*.c")‘
INCLUDE = ‘glob("include/*.h")‘
commit-src {virtual}:
:commit $SOURCE
:removeall .
commit-include {virtual}:
:commit $INCLUDE
:removeall include
```

Note that this is not possible when the sources and include files are in one directory, ":removeall" only works per directory.

Chapter 17. Using CVS

A common way to distribute sources and working together on them is using CVS. This requires a certain way of working. The basics are explained here. For more information on CVS see http://www.cvshome.org.

Obtaining a module

The idea is to hide the details from a user that wants to obtain the module. This requires making a toplevel recipe that contains the instructions. Here is an example:

```
CVSROOT = :pserver:anonymous@cvs.myproject.sf.net:/cvsroot/myproject
:child mymodule/main.aap {fetch = cvs://$CVSROOT}
fetch:
:fetch {fetch = cvs://$CVSROOT} mymodule
```

Executing this recipe will use the "fetch" target. The ":fetch" command takes care of checking out the whole module "mymodule".

Note that this toplevel recipe cannot be obtained from CVS itself, that has a chicken-egg problem.

Fetching

The child recipe "mymodule/main.aap" may be totally unaware of coming from a CVS repository. If this is the case, you can build and install with the recipe, but not fetch the files or send updates back into CVS. You need to use the toplevel recipe above to obtain the latest updates of the files. This will then update all the files in the module. However, the toplevel recipe itself will never be fetched.

To be able to fetch only some of the files of the module, the recipe must be made aware of which files are coming from CVS. This is done by using an "fetch" attribute with a URL-like specification for the CVS server: {fetch = cvs://servername/dir}. Since CVS remembers the name of the server, leaving out the server name and just using "cvs://" is sufficient. Example:

```
SOURCE = foo.c version.c
INCLUDE = common.h
TARGET = myprogram
:attr {fetch = cvs://} $SOURCE $INCLUDE
```

If you now do "aap fetch" with this recipe, the files foo.c, version.c and common.h will be updated from the CVS repository. The target myprogram isn’t updated, of course.

Note: When none of the used recipes specifies a "fetch" target, one will be generated automatically. This will go through all the nodes used in the recipe and fetch the ones that have an "fetch" attribute.

The recipe itself may also be fetched from the CVS repository:

```
:recipe {fetch = cvs://}
```

When using files that include a version number in the file name, fetching isn’t needed, since these files will never change. To reduce the overhead caused by checking for changes, give these files a "constant" attribute (with a value non-empty non-zero value). Example:

```
PATCH = patches/fix-1.034.diff {fetch = $FTPDIR} {constant}
```

To update a whole directory, omit the "fetch" attribute from individual files and use it on the directory. Example:

```
SOURCE = main.c version.c
TARGET = myprog
:attr {fetch = cvs://} .
```

Alternatively, a specific "fetch" target may be specified. The automatic updates are not used then. You can specify the "fetch" attribute right there.

```
fetch:
:fetch {fetch = cvs://} $SOURCE
```

If you decided to checkout only part of a module, and want to be able to get the rest later, you need to tell where in the module to file can be found. This is done by adding a "path" attribute to the cvs:// item in the fetch attribute. Example:

```
fetch:
:fetch {fetch = $CVSROOT {path = mymodule/foo}} foo.aap
```

What will happen is that aap will checkout "mymodule/foo/foo.aap", while standing in two directories upwards. That’s required for CVS to checkout the file correctly. Note: this only works as expected if the recipe is located in the directory "mymodule/foo"!
If the "path" attribute is omitted, A-A-P will obtain the information from the "CVS/Repository" file. This only works when something in the same directory was already checked out from CVS.

Checking in

When you have made changes to your local project files and want to upload them all into the CVS repository, you can use this command:

```
:cvsdist {file} ... {server}
```

The list of files must include _ALL_ the files that you want to appear in CVS for the current directory and below. Files that were previously not in CVS will be added ("cvs add file") and missing files are removed ("cvs remove file"). Then all files are committed ("cvs commit file").

To be able to commit changes you made into the CVS repository, you need to specify the server name and your user name on that server. Since the user name is different for everybody, you must specify it in a recipe in your ~/.aap/startup/ directory. For example:

```
CVSUSER_AAP = foobar
```

The name of the variable starts with "CVSUSER" and is followed by the name of the project. That is because you might have a different user name for each project.

The method to access the server also needs to be specified. For example, on SourceForge the "ext" method is used, which sends passwords over an SSH connection for security. The name used for the server then becomes:

```
:ext:$CVSUSER_AAP@cvs.a-a-p.sf.net:/cvsroot/a-a-p
```

You can see why this is specified in the recipe, you wouldn't want to type this for commiting each change!

DISTRIBUTING YOUR PROJECT WITH CVS

This is a short how-to that only explains how to distribute a set of files (and directories) using CVS.

1. Copy the files you want to distribute to a separate directory

    Mostly you have various files in your project for your own use that you don't want to distribute. These can be backup files and snippets of code that you want to keep for later. Since CVS imports all files it can find, best is to make a copy of the project. On Unix:

    ```
     cp -r projectdir tempdir
    ```

    Then delete all files you don't want to distribute. Be especially careful to delete "aap" directories and hidden files (starting with a dot). It's better to delete too much than too few: you can always add files later.

2. Import the project to the CVS repository

    Move to the newly created directory ("tempdir" in the example above). Import the whole tree into CVS with a single command. Example:

```
cd tempdir
     cvs -d:ext:myname@cvs.myproject.sf.net:/cvsroot/myproject import mymodule mypro-
ject start
```

    Careful: This will create at least one new directory "mymodule", which you can't delete with CVS commands.
    This will create the module "mymodule" and put all the files and directories in it. If there are any problems, read the documentation

```
    available for your CVS server.

3. Checkout a copy from CVS and merge

    Move to a directory where you want to get your project back.  Create the
    directory "myproject" with this example:

     cvs -d:ext:myname@cvs.myproject.sf.net:/cvsroot/myproject checkout mymodule

    You get back the files you imported in step 2, plus a bunch of "CVS"
    directories.  These contain the administration for the cvs program.  Move
    each of these directories back to your original project.  Example:

     mv myproject/CVS projectdir/CVS
     mv myproject/include/CVS projectdir/include/CVS

    If you have many directories, one complicated command does them all:

cd myproject
     find . -name CVS -exec mv {} ../projectdir/{} \;

4. Publish changes

    After making changes to your project and testing them, it’s time to send
    them out.  In the recipe you use for distribution, add one command that
    will publish all the files by updating the module in CVS server.  Example:

FILES = $SOURCE $INCLUDE main.aap
     :publishall {publish = :ext:$CVSUSER_MYPROJECT@cvs.myproject.sf.net:/cvsroot/myproject

    Careful: $FILES must contain all files that you want to publish in this
    directory and below.  If $FILES has extra files they will be added in CVS.
    Files missing from $FILES will be removed from CVS.
    You must assign $CVSUSER_MYPROJECT your user name on the CVS server.
    Usually you do this in one of your personal A-A-P startup files, for
    example "~/.aap/startup/main.aap".


USING SOURCEFORGE

If you are making open source software and need to find a place to distribute
it, you might consider using SourceForge.  It’s free and relatively stable.
They provide http access for your web pages, a CVS repository and a server for
downloading files.  There are news groups and maillists to support
communication.  Read more about it at http://sf.net.

Since you never know what happens with a free service, it’s a good idea to
keep all your precious work on a local system and update the files on
SourceForge from there.  If several people are updating the SourceForge site,
either make sure everyone keeps copies, or make backup copies (at least
weekly).

You can use A-A-P recipes to upload your files to the SourceForge servers.  To
```

avoid having to type passwords each time, use an ssh client and put your public keys in your home directory (for the web pages) or on your account page (for the CVS server). Read the SourceForge documentation for how to do this.

For uploading web pages you can use a recipe like this:

```
FILES = index.html
download.html
news.html
images/logo.gif
:attr {publish = scp://myname@myproject.sf.net//home/groups/m/my/myproject/htdocs/%file%} $
```

Start this recipe with the "publish" target.

For sourceforge, set environment variable CVS_RSH to "ssh". Otherwise you won't be able to login. Do "touch ~/.cvspass" to be able to use "cvs login" Upload your ssh keys to your account to avoid having to type your password each time.

Chapter 18. Issue Tracking

```
BUG REPORTING

A recipe used to install an application should offer the "report" target.
This is the standard way for a user to report a problem.  The recipe should
then help the user with reporting a problem as much as possible.

An example is to send the developer an e-mail.  All useful information is put
in the message by the recipe, so that the user only has to fill in his
specific problem.  Example:

report:
tmpfile = ‘tempfname()‘
:print >$tmpfile  program version: $VERSION
:print >>$tmpfile system type: ‘os.name‘
@if os.name == "posix":
:print >>$tmpfile "      " ‘os.uname()‘
:do email {subject = ’problem in foobar’}
{to = bugs@foobar.org}
{edit}
{remove}
$tmpfile

When a web form is to be filled in, give the user hints about what information
to fill in certain fields and start a browser on the right location.  Example:

report:
:do view {async} http://www.foo.org/bugreport/
tmpfile = ‘tempfname()‘
:print >$tmpfile   use this information in the bug report:
:print >>$tmpfile  program version: $VERSION
:print >>$tmpfile  system type: ‘os.name‘
:do view {remove} $tmpfile


BUG FIXING

One a bug has been fixed, the developer needs to update the related bug
report.  The "tracker" target is the standard way for a developer to get to
the place where the status of the bug report can be changed.

Since trackers work in many different ways the recipe has to specify the
commands.  Example:

tracker:
:do view {async} http://www.foo.org/tracker?assigned_to=$USER

This is very primitive.  The developer still has to locate the bug report and
change the status and add remarks.  The above example at least lists the bug
```

```
reports for the current user.
```

Chapter 19. Using Python

Python commands can be used where A-A-P commands are not sufficient. This includes conditionally executing of commands, repeating commands for a list of items and much more.

```
PYTHON EXPRESSION:

Python expressions can be embedded in many places.  They are specified in
backticks.

The result should be a string or a list of strings.  A list is automatically
converted to a white-separated string of all list items.

A Python expression cannot be used for the variable name in an assignment.
This doesn’t work:

‘varname‘ = value

Use this instead:

@eval(varname + ’ = "value"’)

When using a function from a module, it must be imported first.  Example:

@from glob import glob
SOURCE = ‘glob(’*.c’)‘

However, for your convenience these things are imported for you already:
from glob import glob

A backtick in the Python expression has to be doubled to avoid it being
recognized as the end of the expression:

CMD = ‘my_func("“grep -l foo *.c“")‘

contains the Python expression:

my_func("‘grep -l foo *.c‘")

Note that a following attribute is only attached to the last item resulting
from the Python expression.

SOURCE = ‘glob(’*.c’)‘ {check = md5}

Can be equivalent to:

SOURCE = foo.c bar.c {check = md5}

To apply it to all items, use parenthesis

SOURCE = (‘glob(’*.c’)‘) {check = md5}
```

```
Can be equivalent to:

SOURCE = (foo.c bar.c) {check = md5}
or:
SOURCE = foo.c {check = md5} bar.c {check = md5}


Backtick expressions and $VAR can be used inside a string:

DIR = /home/foo /home/bar
:print "‘DIR + "/fi le"‘"
results in:
"/home/foo /home/bar/fi le"

Compare this to:
:print "$*DIR/fi le"
which results in:

In the result of the Python expression a $ is changed to $$, so that it’s not
used for a variable name.  The $$ is reduced to $ again when evaluating the
whole expression.


USEFUL PYTHON ITEMS:

VAR = ‘os.environ.get(’VAR’, ’default’)‘

Obtain environment variable $VAR.  If it is not set use "default".

@os.environ["PATH"] = mypath

Set an environment variable.

files = ‘glob("*.ext")‘

Obtain a list of files matching "*.ext".  A-A-P will take care of turning the
list that glob() returns into a string, using quotes where needed.

choice = ‘raw_input("Enter the value: ")‘

Prompt the user to enter a value.

tempfile = ‘tempfname()‘

Get a file name to use for temporary files.  The file will not exist.  If you
create it you need to make sure it is deleted afterwards.


AAP PYTHON FUNCTIONS:

These functions can be used in Python code:

aap_sufreplace(from, to, expr)
```

```
Returns "expr" with all occurences of the suffix
"from" changed to "to".
When "from" is empty any suffix is changed to "to".
"expr" can be a list of file names.
Example:
OBJECT = ‘aap_sufreplace("", OBJSUF, SOURCE)‘

aap_abspath(expr) Returns "expr" with all file names turned into
absolute paths.  Prepends the current directory to
each item in "expr" which isn’t an absolute path name.
Example:
:print ‘aap_abspath("foo bar")‘
results in:
/home/mool/test/foo /home/mool/test/bar

string = src2obj(dict, source)
Transform a string, which is a list of source files,
into the corresponding list of object files.  Each
item in "source" is change by prepending $BDIR and
changing or appending the suffix to $OBJSUF.
"dict" is used to obtain the value of $BDIR and
$OBJSUF from, normally "globals()" can be used:

OBJECT = ‘src2obj(globals(), SOURCE)‘

program_path(name)
Returns the path for program "name".  This uses the
$PATH environment variable or os.defpath if it isn’t
set.
On MS-Windows and OS/2 also checks with these
extensions added: ".exe", ".com", ".bat", ".cmd".
When "name" includes a suffix (a dot in the last
component) adding extensions is not done.
Returns the first program found.
Returns None when "name" could not be found.
Optional arguments:
path search path to use instead of $PATH;
when a string items are separated with
os.pathsep
pathext extensions to try; when a string
items are separated with os.pathsep
skip name of directory to skip searching
Example, search for program "foo.py" and "foo.sh":

p = ‘program_path("foo", pathext = [ ’.py’, ’.sh’ ])‘

redir_system(recdict, cmd [, use_tee])
Execute shell commands "cmd" and return two items: a
number indicating success and the stdout.  The
"recdict" argument should be obtained with
"globals()".  By default "tee" is used to display the
output as well as redirecting it.  When no output is
desired set "use_tee" to zero.  Example:
  ok, text = redir_system(globals(), "ls", 0)
  if ok:
       print "ls output: %s" % text
  else:
       print "ls failed"

list = sort_list(list)
sorts a list and returns the list.  Example:

INP = ‘sort_list(glob("*.inp"))‘

list = var2list(var) Turns a variable with a string value into a list of
items.  Attributes are ignored.  Example:

source = file1 file2 file3
@for fname in var2list(source):
:sys prog $fname

list = var2dictlist(var)
Turns a variable with a string value into a list of
dictionaries.  Each dictionary has a "name" entry for
item itself and other entries are attributes.
Example:

source = file1 {force} file2 file3
@for item in var2dictlist(source):
@    if item.get("force"):
:print forced item: ‘item["name"]‘

string = file2string(fname [, dict])
Reads the file "fname" and concatenates the lines into
one string.  Lines starting with a ’#’ are ignored.
One space is inserted in between joined lines, other
white space at the start and end of a line is removed.
When "fname" doesn’t exist or can’t be read an error
message is given and an empty string is returned.
"dict" is used to obtain the value for $MESSAGE.
The default is None.

has_target(dict, target)
Returns:
0 no dependency for "target"
1 dependency for "target" exists
2 dependency for "target" exists and has
build commands
"dict" is the dictionary of the environment, normally
globals():

@if not has_target(globals(), "fetch"):

tempfname() Returns the name of a file which does not exist and
can be used temporarily.
```

```
PYTHON BLOCK:

A block of Python commands is started with a ":python" command.  Optionally
a terminator string may be specified.  This cannot contain white space.  A
comment may follow.  If no terminator string is specified the python code ends
where the indent is equal to or less than the ":python" command.  Otherwise
the Python block continues until the terminator string is found in a line by
itself.  It may be preceded and followed by white space and a comment may
follow.

SOURCE = foo.c bar.c
:python
    for i in items:
SOURCE = SOURCE + " " + i
...
:python EOF
    INCLUDE = glob("include/*.c")
     EOF
```

Chapter 20. Porting an Application

Porting an application means starting with the original sources and changing them a little bit to make it work. When using a recipe the port will work on many platforms.

```
The presence of the variable "PORTNAME" triggers A-A-P to handle the recipe as
a port recipe.  This will happen:

1. Add the dependency:
all: dependcheck builddepend
fetch checksum extract patch configure build test
package install rundepend
2. For each of the targets in the dependency above add a dependency based on
   the variables set in the recipe.

You can define dependencies for:
do-XXX replace the body of a step
pre-XXX   do something before a step
post-XXX do something after a step

General info provided by the port recipe:
Various variables need to be set to specify properties of the port.
PORTNAME name of the port   "foobar"
PORTVERSION app version number   "3.8alpha"
PORTREVISION port patchlevel    "32"
--> foobar-3.8alpha_32
info URL http://www.foobar.org
MAINTAINER_NAME maintainer name John Doe
MAINTAINER maintainer e-mail john@foobar.org
PORTCOMMENT short description get foo into the bar
PORTDESCR long description blah blah blah
IS_INTERACTIVE requires user input yes or no

Variables that can be used by the port recipe:
WRKDIR directory all files are extracted in and the building
is done in default: "work"
DISTDIR directory where downloaded distfiles are stored
default: "distfiles"
PATCHDISTDIR directory where downloaded patches are stored
default: "patches"
PKGDIR directory where files are stored before creating a
package default: "pack"

Variables that may be set by the port recipe, defaults are set only after
reading the recipe:
WRKSRC Directory inside $WRKDIR where the unpacked sources
end up. This should be the common top directory in the
unpacked archives. default: $PORTNAME-$PORTVERSION
When using CVS it is always set to $CVSWRKSRC (also
when $WRKSRC was already set).
CVSWRKSRC Directory inside $WRKDIR where files obtained with CVS
end up. default: the first entry in
```

```
$CVSMODULES
  NO_WRKSUBDIR When not empty, $WRKSRC will be empty instead of using
the default value. default: not set
PATCHDIR Directory inside $WRKDIR where patches are to be
applied.   default: $WRKSRC
BUILDDIR Directory inside $WRKDIR where building is to be
done.   default: $WRKSRC
TESTDIR Directory inside $WRKDIR where testing is to be
done.   default: $WRKSRC

INSTALLDIR Directory inside $WRKDIR where $INSTALLCMD is to be
done.   default: $WRKSRC
CONFIGURECMD    Set to the command used to configure the application.
Usually "./configure".
Default: nothing
BUILDCMD Set to the command used to build the application.
Usually just "make".
Default: "aap".
TESTCMD Set to the command used to test the application.
Usually "make test".
Default: "aap test".
INSTALLCMD Set to the command used to do a fake install of the
application.
Default: "aap install DESTDIR=$PKGDIR


For running configure, define a pre-build target.  Example:

pre-build:
:cd $WRKDIR/$WRKSRC
:sys LANG=de ./configure --with-extra


DEPENDENCY FORMAT

Dependencies on other modules are specified with the various DEPEND_
variables.  The format of these variables is a list of items.

Items are normally white separated, which means there is an "and" relation
between them.  Alternatively "|" can be used to indicate an "or" relation.

DEPENDS = perl python require both perl and python
DEPENDS = perl | python require perl or python

Parenthesis can be used to group items.  Parenthesis must be used when
combining "or" and "and" relations.  Example:

(foo bar) | foobar Either both "foo" and "bar" or "foobar"
foo bar | foobar Illegal
foo (bar | foobar) "foo" and either "bar" or "foobar"

When a dependency is not met the first alternative will be installed, thus the
order of "or" alternatives is significant.
```

```
Each item is in one of these formats:

name-version_revision a specific version and revision
name any version
name-regexp a version specified with a regular
expression (shell style)
name>=version_revision any version at or above a specific
version and revision
name>version_revision any version above a specific version
and revision
name<=version_revision any version at or below a specific
version and revision
name<version_revision any version below a specific version
and revision
name!version_revision any version but a specific version and
revision

In the above "_revision" can be left out to ignore the revision number.  It
actually works as if there is a "*" wildcard at the end of each item.

"name" can contain wildcards.  When a part is following it is appended at the
position of the wildcard (or at -9 if it comes first).
foo-* matches foo-big, foo-big-1.2 and foo-1.2
foo-*!1.2 matches foo-big, foo-big-1.2 and skips foo-1.2

The version specifications can be concatenated, this creates an "and"
relation.  Example:

foo>=1.2<=1.4 versions 1.2 to 1.4 (inclusive)
foo>=1.2!1.8 versions 1.2 and above, excluding 1.8
xv>3.10 versions above 3.10, accepts xv-3.10a

The "!" can be followed by parenthesis containing a list of specifications.
This excludes the versions matching the specifications in the parenthesis.
Example:

foo>=1.1!(>=1.3<=1.5) versions 1.1 and higher, but not
versions 1.3 to 1.5
foo>=6.1!6.1[a-z]* version 6.1 and later but not 6.1a,
6.1rc3, etc.

When a dependency is not met the newest version that meets the description is
used.

For systems that do not allow specifying dependencies like this in a binary
pacakge, the specific package version that exists when generating the package
is used.


DEPENDENCIES FOR VARIOUS STEPS

The various variables used to specify dependencies:
```

```
DEPEND_FETCH Required for fetching files.  Also for
computing checksums.
DEPEND_EXTRACT Required for unpacking archives.
DEPEND_BUILD Required for building but not necessarily for
running; these are not included in the binary
package; items may also appear in DEPEND_RUN.
DEPEND_TEST Required for testing only; don’t include items
that are already in DEPEND_RUN.
DEPEND_RUN Required for running; these will also be
included in the generated binary package.
DEPENDS Used for DEPEND_BUILD and DEPEND_RUN when
empty.

Only the dependencies specified with DEPEND_RUN will end up in the generated
binary package.  When using a shared library, it is recommended to put a
dependency on the developer version (includes header files) in DEPEND_BUILD
and a dependency on the library itself in DEPEND_RUN.  The result is that when
installing binary packages the header files for the library don’t need to be
installed.


The "CONFLICTS" variable should be set to specify modules with which this one
conflits.  That means only one of the two packages can be installed in the
same location.  It should still be possible to install the packages in
different locations.  The format of CONFLICTS is identical to that of the
DEPENDS_ variables.

Dependencies are automatically installed, unless "AUTODEPEND" is "no".
The dependencies are normally satisfied by installing a port.  When a
satisfying port can not be found a binary package is installed.
The ports and packages are first searched for on the local system.  When not
found the internet is searched.

The order of searching can be changed with "AUTODEPEND":
binary only search for binary packages, default
locations
source only search for ports, default locations
source {path = /usr/ports http://ports.a-a-p.org}
only search for ports in /usr/ports and on the
ports.a-a-p.org web site.


STEPS

These are the individual steps for installing a ported application.  Each step
up to "install" depends on the previous one.  Thus "aap install" will do all
the preceding steps.  But the steps that have already been successfully done
will be skipped.  The "rundepend", "installtest", "clean", etc. targets do not
depend on previous steps, they can be used separately.

dependcheck: check if required dependencies can be fulfilled
This doesn’t install anything yet, it does an early check if building
```

```
and/or installing the port will probably work before starting to
download files.
This uses all the DEPEND_ variables that will actually be used.
Fails if something is not available.

fetchdepend: check dependencies for fetch and checksum
Uses DEPEND_FETCH, unless disabled with AUTODEPEND=no

fetch: get required files
if $CVSMODULES is set and $CVS is not "no", obtain files from CVS
uses $CVSROOT or cvsroot attribute in $CVSMODULES
$CVSWRKSRC is where the files will end up (default is first
entry in $CVSMODULES)
also obtain $CVSDISTFILES if defined
also obtain $CVSPATCHFILES if defined
use post-fetch to rename directories
else
if $DISTFILES is set obtain them
if $PATCHFILES is set obtain them
Use MASTER_SITES for [CVS]DISTFILES
Use PATCH_SITES for [CVS]PATCHFILES
the [CVS]DISTFILES are put in $DISTDIR
the [CVS]PATCHFILES are put in $PATCHDISTDIR
The directory can be overruled with a {distdir = dir} attribute on
individual patch files.
Files that already exist are skipped (if there is a checksum error,
delete the file(s) manually).

checksum: check if checksums are correct
The port recipe writer must add the "do-checksum" target with
":checksum" commands to verify that downloaded files are not
corrupted.  Example:

# >>> automatically inserted by "aap makesum" <<<
do-checksum:
    :checksum $DISTDIR/foo-1.1.tgz {md5 = 2341423423423423434}
    :checksum $PATCHDISTDIR/foo-patch3.gz {md5 = 3923858739234}
# >>> end <<<

The "aap makesum" command can be used to generate the lines.

extractdepend: check dependencies for extract and patch
Uses DEPEND_EXTRACT, unless disabled with AUTODEPEND=no

extract: unpack the archives
unpack archives in the right place
use $EXTRACT_ONLY if defined, otherwise $DISTFILES or
$CVSDISTFILES when CVS was used
Uses the "extract" action.  Do extract a new type of archive:
add filetype detection for this type
define an "extract" action
Extraction is done in $WRKDIR.  A subdirectory may be specified with
the "extractdir" attribute on each archive.
```

```
DISTFILES = foo-1.1.tgz foo_doc-1.1.tgz {extractdir = doc}

patch:
apply patches not applied already
$PATCHCMD defines the patch command, default "patch -p < "
The patch file name is appended, unless "%s" appears in the
string, then it’s replaced by the file name.
A "patchcmd" attribute on each patch file may specify a patch command
that overrules $PATCHCMD.
The patches are applied in $WRKDIR/$PATCHDIR (default: $WRKSRC).
A "patchdir" attribute on each patch file may overrule the value of
$PATCHDIR.

builddepend: check dependencies for configure and build
Uses DEPEND_BUILD, unless disabled with AUTODEPEND=no

configure: configuration
autoconf/imake/etc.  USE_IMAKE
may be empty

build:
run make or aap
USE_GMAKE
USE_BSDMAKE
BUILDCMD=make foo default: "aap"
Done in $WRKDIR/$BUILDDIR, default: $WRKDIR/$WRKSRC

testdepend: check test dependencies
Uses DEPEND_TEST.
check if all required items are present
try to install them automatically, unless disabled  AUTODEPEND=no
This is skipped when "SKIPTEST=yes"

test:
check if building was done properly
TESTCMD=make testall default: "aap test"
This is skipped when "SKIPTEST=yes"
Done in $WRKDIR/$TESTDIR, default: $WRKDIR/$WRKSRC

package: create binary package
Two methods to select files to include:
1. list of files below $WRKDIR, with "dest" attr where they end up
   PACKFILES = $WRKSRC/bin/prog {dest = /usr/local/bin/prog}
    $WRKSRC/man/prog.1 {dest = usr/local/man/man1/prog.1}
2. prog to fake-install into $PKGDIR, use all files there.
   INSTALLCMD = "aap install DESTDIR=$PKGDIR"
   INSTALLDIR in $WRKDIR   default is $WRKSRC
   $PKGDIR/$PREFIX is where files end up
generate packing list and other files for local package system
execute pkg_create or equivalent

install: install the binary package
in home dir or in system (require typing root password)
```

```
execute pkg_add or equivalent
Exception: This updates the "rundepend" and "installtest" targets
after updating the post-install target.  This allows doing "aap
install", which is a lot more obvious than "aap installtest".

rundepend: check runtime dependencies
Check if all required items specified with $DEPEND_RUN are present and
tries to install them automatically, unless $AUTODEPEND is "no".
This is skipped if $SKIPRUNTIME is "yes".  The pre-rundepend and
post-rundepend are still done, they should check $SKIPRUNTIME
themselves.
"aap rundepend" will _not_ cause previous steps to be updated.

installtest: test if the installed package works
This is empty by default, specify a "do-installtest" target to
actually do something.
Note that when $SKIPRUNTIME is "yes" the dependencies have not been
verified and running the application might not work.

deinstall: uninstall the binary package
Execute pkg_delete or equivalent.
Does not depend on other steps.

clean: delete all generated, unpacked, patched and CVS files
not the downloaded files.
Does not depend on other steps.

distclean: delete everything except the toplevel recipe
Does not depend on other steps.

makesum: generate checksum file
Generates a checksum file to be able the check if the fetched files
were not corrupted.
The generated file can be included into the port recipe.
Does not depend on other steps.  The files must already be present.

srcpackage: generate a package with recipe and source files
Put main recipe and all downloaded files into an archive.  The
resulting archive can be installed without downloading.
Depends on the "fetch" target.


PORT DESCRIPTION

The text to describe the port is usually a page full of plain text.  Here is an
example:

PORTDESCR << EOF
This is the description of the port.

See our website http://myport.org.
EOF
```

In the rare situation that "EOF" actually appears in the text you can use anything else, such as "THEEND".

USING AUTOCONF

The autoconf system is often used to configure C programs to be able to compile them on any Unix system. This section explains how to use autoconf with A-A-P in a nice way.

A recipe that uses the generated configure script can start like this:

```
$BDIR/config.h $BDIR/config.aap : \
 configure config.arg config.h.in config.aap.in
:sys ./configure ‘file2string("config.arg")‘
:move {force} config.h $BDIR/config.h
:move {force} config.aap $BDIR/config.aap
config.arg:
:touch {exist} config.arg

:update $BDIR/config.aap
:include $BDIR/config.aap
```

What happens here is that the "config.aap" target is updated before any of the building is done. This is required, because running the configure script will generate or update the "config.aap" file that influences how the building is done.

The arguments for configure are stored in the "config.arg" file. This makes it easy to run configure again with the same arguments. There should be a "config.txt" file that explains all the possible configure arguments, with examples that can be copied into "config.arg". Example:

```
# Select the library to be used for terminal access.  When omitted a
# series of libraries will be tried.  Useful values:
--with-tlib=curses
--with-tlib=termcap
--with-tlib=termlib
```

The user can now copy one of the example lines to his "config.arg" file. Example:

```
# select specific terminal library
--with-tlib=termcap
```

Comment lines can be used, they must start with a "#". Note: a comment after an argument doesn’t work, it will be seen as an argument.

When updating to a new version of the program, the same "config.arg" file can still be used. A "diff" between the old and the new "config.txt" will show what configure arguments have changed.

"config.aap" and "config.h" are put in $BDIR, because they depend on the current system. They might also depend on the variant to be build. In that case the ":variant" statement must be before the use of $BDIR. However, if the variant is selected by running configure, the variant must come later. "config.aap" and "config.h" are then updated when selecting another variant.

For a developer there also needs to be a method to generate the configure script from configure.in. This needs to be done even before configure is run. Prepending this to the example above should work:

```
configure {distributed} : configure.in
:del {force} config.cache config.status
:sys autoconf
:update configure
```

The "{distributed}" attribute on the target indicates that the "configure" file is included in the distribution. When "configure" was not built (there is no old signature for it) but it does exist, it doesn’t need to be build. This is useful to avoid running autoconf after unpacking sources that already include the up-to-date configure script. But when "configure.in" changes after executing the recipe once, configure will be built, because the signatures will be remembered when executing the recipe. When changing configure.in before executing the recipe (or after deleting the recipe file manually) no building will be done, the change will not be noticed.

The "config.cache" and "config.status" files are deleted, because they may become invalid when generating a new configure script.

Chapter 21. Debugging a Recipe

The log file shows what happened while A-A-P was executing. Often you can figure out what went wrong by looking at the messages. The log file is named "AAP/log". It is located in the directory of the main recipe. If you executed aap again and now want to see the previous log, it is named "AAP/log1". Older logs are "AAP/log2", "AAP/log3", etc. This goes up to "AAP/log9".

```
MESSAGES

The kind of messages given can be changed with the MESSAGE variable.  It is a
list of message types for which the message is actually displayed:
all everything
error error messages
warning warnings
depend dependencies, the reasoning about what to build
info general info (file copy/delete, up/downloads)
extra extra info (why something was done)
system system (shell) commands
changedir changing directories

The command line arguments "-v" and "-s" can be used to make the most often
used selections:

argument MESSAGE
(nothing) error,info,system
-v --verbose all
-s --silent error

Other values can be assigned at the command line.  For example, to only see
error and dependency messages:

aap MESSAGE=error,depend  (other arguments)

Don’t forget that excluding "error" means that no error messages are given!

No matter what messages are displayed, all messages are written in the log
file.  This can be used afterwards to see what actually happened.  The name of
the log file is "aap/log".  This is located relative to the main recipe.
Older log files are also remembered.  The previous log is "aap/log1".  The
oldest log is "aap/log9".
```

Chapter 22. Customizing Filetype Detection and Actions

See the tutorial for a simple example how to define a new filetype.

FILETYPE DETECTION

A-A-P detects the type of a file automatically. This is used to decide what tools can be used for a certain file.

To manually set the file type of an item add the "filetype" attribute. This overrules the automatic detection.
Example:

```
foo.o : foo.x {filetype = cpp}
```

Most detection is already built in. If this is not sufficient for your work, filetype detection instructions can be used to change the way file type detection works. These instructions can either be in a file or directly in the recipe:

```
:filetype  filename
:filetype
suffix p pascal
script .*bash$ sh
```

The advantage of using a file is that it will also be possible to use it when running the filetype detection as a separate program. The advantage of using the instructions directly in the recipe is that you don’t have to create another file.

For the syntax of the file see filetype.txt.

It is sometimes desired to handle a group of files in a different way. For example, to use a different optimizer setting when compiling C files. An easy way to do this is to give these files a different filetype. Then define a compile action specifically for this filetype. Example:

```
:attr {filetype = c_opt} bar.c foo.c
:action compile c_opt
CFLAGS = -O3
:do compile $source
```

The detected filetypes never contain an underscore. A-A-P knows that the underscore separates the normal filetype from the additional part. When no action is found for the whole filetype, it is tried by removing the "_" and what comes after it. (This isn’t fully implemented yet!)

Care should be taken that an action should not invoke itself. For example, to always compile "version.c" when linking a program:

```
:attr {filetype = my_prog} $TARGET
:action build my_prog object
:do compile {target = version$OBJSUF} version.c
:do build {target = $target {filetype = program}}
$source version$OBJSUF
```

Without "{filetype = program}" on the :do build" command, the action would invoke itself, since the filetype for $TARGET is "my_prog".

EXECUTING ACTIONS

The user can select how to perform an action for each type of file. For example, the user may specify that the "view" action on a file of type "html" uses Netscape, and the "edit" action on a "html" file uses Vim.

To start an application:

```
:do action filename
```

This starts the "action" application for file "filename".
Attributes for the commands can be specified after the action:

```
:do action {arg = -r} filename
```

The filetype is automatically detected from the file name and possibly its contents. Extra suffixes like ".in" and ".gz" are ignored.
To overrule the automatic filetype detection, specify the filetype as an attribute on the file:

```
:do action filename {filetype = text}
```

Multiple filename arguments can be used:

```
:do action file1 file2 file3
```

For some actions a target can or must be specified. This is done as an attribute on the action:

```
:do action {target = outfile} infile1 infile2
```

Attributes of the filenames other than "filetype" are not used!

The "async" attribute can be used to start the application without waiting for it to finish. However, this only works for system commands and when multi-tasking is possible. Example:

```
:do email {async} {remove} {to = piet} {subject = done building} logmessage
```

The "remove" attribute specifies that the files are to be deleted after the command is done, also when it fails.

```
When the filetype contains a "_" and no action can be found for it, the search
for an action is done with the part before the "_".  This is useful for
specifying a variant on a filetype where some commands are different and the
rest is the same.

"action" is usually one of these:

view Look at the file.  The "readonly" attribute is
normally set, it can be reset to allow editing.
edit Edit the file.
email Send the file by e-mail.  Relevant attributes:
    edit edit before sending
    subject subject of the message
    to destination
build Build the file(s), resulting in a program file.  The
"target" attribute is needed to specify the output
file.

compile Build the file(s), resulting in object file(s).
The "target" attribute may be used to specify the
output file.  When "target" is not specified the
action may use a default, usually $BDIR/root$OBJSUF.

extract Unpack an archive.  Requires the program for unpacking
to be available.  Unpacks in the current directory.

preproc Run the preprocessor on the file.  The "target"
attribute can be used to specify the output file.
When it is missing the output file name is formed from
the input file name by using the ".i" suffix.

reference Run a cross referencer, creating or updating a symbol
database.

strip Strip symbols from a program.  Used to make it smaller
after installing.

Examples:

:do view COPYRIGHT {filetype = text}
:do edit configargs.xml
:do email {edit} bugreport
:do build {target = myprog} main.c version.c
:do compile foo.cpp


Default actions:

These are defined in the default.aap recipe.  TODO: update this info

:action edit default
Uses the environment variable $VISUAL or $EDITOR.  Falls back to "vi"
```

```
for Posix systems or "notepad" otherwise

:action view html,xml,php
Uses the environment variable $BROWSER.  Searches a list of known
browsers if it’s not set.

:action email default
Uses the environment variable $MAILER.  Falls back to "mail".

:action build default default
     Does: ":sys $CC $LDFLAGS $CFLAGS -o $target $source $?LIBS"

:action compile object c,cpp
     Does ":sys $CC $CPPFLAGS $CFLAGS -c -o $target" $source

:action preproc default c,cpp
     Does ":sys $CC $CPPFLAGS -E $source > $target"


Specifying actions:

Applications for an action-filetype pair can be specified with this command:

:action action filetype
commands

This associates the commands with action "action" and filetype "filetype".
The commands are A-A-P commands, just like what is used in the build commands
in a dependency.

It is also possible to specify an action for turning a file of one filetype
into another filetype.  This is like a ":rule" command, but using filetypes
instead of patterns.

:action action out-filetype in-filetype
commands

Actually, specifying an action with one filetype is like using "default" for
the out-filetype.

Several variables are set and can be used by the commands:
$fname The first file name of the ":do" command.
$source All the file names of the ":do" command.
$filetype The detected or specified filetype for the input.
$targettype The detected or specified filetype for the output.
$action The name of the action for which the commands are
executed.

Furthermore, all attributes of the action are turned into variables.  Thus
when ":do action {arg = -x} file " is used, $arg is set to "-x".  Example for
using an optional "arg" attribute:

:action view foo
```

```
:sys fooviewer $?arg $source
```

In Python code check if the {edit} attribute is specified:

```
:action email default
# when $subject and/or $to is missing editing is required
@if not globals().get("subject") or not globals().get("to"):
    edit = 1
@if globals().get("edit"):
    :do edit $fname
:sys mail -s $subject $to < $fname
```

"action" and "ftype" can also be a comma-separated list of filetypes. There can't be any white space though. Examples:

```
:action view html,xml,php
:sys netscape --remote $source
:action edit,view text,c,cpp
:sys vim $source
```

"filetype" can be "default" to specify an action used when there is no action specifically for the filetype.

Note that the "async" attribute of the ":do" command may cause the ":sys" command to work asynchronously. That is done because the "async" attribute is turned into the "async" variable for the ":action" commands. If you don't want a specific ":sys" command to work asynchronously, reset "async":

```
:action view foo
tt = $?async
async =
:sys foo_prepare
async = $tt
:sys foo $source
```

However, since two consecutive ":sys" commands are executed together, this should do the same thing:

```
:action view foo
:sys foo_prepare
:sys foo $source
```

Quite often the program to be used depends on whether it is available on the system. For example, viewing HTML files can be done with netscape, mozilla, konquerer or another browser. We don't want to search the system for all kinds of programs when starting, it would make the startup time quite long. And we don't want to search each time the action is invoked, the result of the first search can be used. This construct can be used to achieve this:

```
BROWSER =
:action view html
    @if not BROWSER:
:progsearch BROWSER netscape mozilla konquerer
```

```
    :sys $BROWSER $source
    :export BROWSER
```

The generic form form :progsearch is:

```
:progsearch varname progname ...
```

":export" needs to be used for variables set in the commands and need to be used after the action is finished.

The first argument is the variable name to which to assign the resulting program name. When none of the programs is found an error message is given. Note that shell aliases are not found, only executable programs.

More examples:

```
:action compile c,cpp
:sys $CC $CPPFLAGS $CFLAGS -c $source
:action build object
:sys $CC $LDFLAGS $CFLAGS -o $target $source
```

Chapter 23. Customizing Automatic Depedencies

For various file types A-A-P can scan a source file for header files it includes. This implies that when a target depends on the source file, it also depends on the header files it includes. For example, a C language source file uses #include lines to include code from header files. The object file generated from this C source needs to be rebuild when one of the header files changes. Thus the inclusing of the header file has an implied dependency.

Aap defines a series of standard dependency checks. You don’t need to do anything to use them.

The filetype used for the dependency check is detected automatically. For files with an ignored suffix like ".in" and ".gz" no dependency checking is done.

To avoid all automatic dependency checks, set the variable "autodepend" to "off":

```
autodepend = off
```

To avoid automatic dependencies for a specific file, set the attribute "autodepend" to "off":

```
foo.o : foo.c {autodepend = off}
```

You can add your own dependency checks. This is done with the ":action" command. Its arguments are "depend" and the file types for which the check works. A block of commands follows, which is expected to inspect $source and produce the detected dependencies in $target, which has the form of a dependency. Example:

```
:action depend c,cpp
:sys $CC $CFLAGS -MM $source > $target
```

The build commands are expected to generate a file that specifies the dependency:

```
foo.o : foo.c foo.h
```

The first item (before the colon) is ignored. The items after the colon are used as implied dependencies. The source file itself may appear, this is ignored. Thus these results have the same meaning:

```
foo.xyz : foo.c foo.h
foo.o : foo.h
```

Comments starting with "#" are ignored. Line continuation with "\" is supported. Only the first (continued) line is read.

Aap will take care of executing the dependency check when the source changes or when the command changes (e.g., the value of $CFLAGS). This can be changed with a "buildcheck" attribute after "depend".

```
:action depend {buildcheck = $CFLAGS} c
:sys $CC $CFLAGS -MM $source > $target
```

Aap expects the dependency checker to only inspect the source file. If it recursively inspects the files the source files include, this must be indicated with a "recursive" attribute. That avoids Aap will take care of this and do much more work than is required. Example:

```
:action depend {recursive}  c,cpp
:sys $CC $CFLAGS -MM $source > $target
```

III. Reference Manual

Chapter 24. Aap Command Line Arguments

Three Kinds Of Arguments

`aap [option].. [assignment]... [target]...`

The order of arguments is irrelevant. Options are explained in the next section.

Assignments take the form "VAR=value". This works just like putting this at the top of the main recipe used.

Targets specify what needs to be done. If "target" is omitted, one of the targets in the recipe is executed, see RECIPE EXECUTING.

Options

```
-a
--nocache For remote files, don’t use cached files, download it once for
this invocation.

-c command
--command command
After reading the recipe, execute "command".  May appear
several times, all the commands are executed.

Commands to be executed can be specified on the command line.
This is useful, for example, to fetch individual files:

aap -c ":fetch main.c"
aap -c ":commit common.h"

Since the recipe is first read (and all child recipes), the
attributes that are given to "main.c" will be used for
fetching.

When no target is specified nothing is build, only the
specified commands are executed.  But the toplevel commands in
the recipe are always executed.

Keep in mind that the shell must not change the argument, use
single quotes to avoid $VAR to be expanded by the shell:

aap -c ’:print $SOURCE’


-d flags switch on debugging for ’flags’

-f FILE
--recipe FILE specify recipe to execute

-F
```

```
--force force rebuilding even when a target appears up-to-date

-h
--help print help message and exit

-I DIR
--include DIR directory to search for included recipes

-j N
--jobs N maximum number of parallel jobs

-k
--continue continue after encountering an error

-l
--local Do not recurse into subdirectories.  Only applies to "add" and
"remove" targets on the command line.  Also for "revise" for
its remove action.

-n
--nobuild Only print messages about what will be done, don't execute
build rules.  Commands at the toplevel and commands to
discover dependencies are executed, but system commands,
commands that download, upload, write or delete files and
version control commands are skipped.
Note: since no files are downloaded, ":child" commands won't
work for non-existing recipes.

--profile FILE
Profile the execution of A-A-P and write the results in FILE.
This file can then be examined with the standard Python module
"pstats".

-R
--fetch-recipe
Fetch the recipe and child recipes.  This is impled by using
a "fetch" or "update" target.

-S
--stop stop building after encountering an error (default)

-s
--silent print less information, see MESSAGE.

-t
--touch Update signatures on targets without executing build commands.
After doing this the specified targets and intermediate
results are considered up-to-date.
Commands at the toplevel will be executed, except system
commands, commands that write a file and version control
commands.

-u
```

```
--up
--search-up search directory tree upwards for main.aap recipe

-V
--version print version information and exit

-v
--verbose print more information, see MESSAGE.

- end of options, only targets and assignments follow
```

Chapter 25. Recipe Syntax and Semantics

Recipe syntax definition

This defines the recipe syntax (more or less). NOTE: this may still change quite a bit.

Table 25-1. Notation

|	separates alternatives
()	grouping a sequence of items or alternatives
[]	optional items (also does grouping)
""	contains literal text; """ is one double quote
...	indicates the preceding item or group can be repeated
EOL	an end-of-line, optionally preceded by a comment
INDENT	an amount of white space, at least one space
INDENT2	an amount of white space, at least one space more than INDENT

A comment starts with "#" and continues until the end-of-line. It may appear in many places.

White space may be inserted in between items. It is often ignored, but does have a meaning in a few places.

Line continuation is done with a backslash immediately before an end-of-line. It’s not needed for an assignment and other commands that don’t have a build_block: the item continues if the following line has more indent than the first line.

```
aapfile          ::=   ( [ aap_item ] EOL ) ...
aap_item         ::=   dependency | rule | variant | otherfile | build_command

dependency       ::=   targets ":" [ dependencies ] [ EOL build_block ]
targets          ::=   item_list
dependencies     ::=   item_list
build_block      ::=   INDENT build_command [ EOL [ INDENT build_command ] ] ...

rule             ::=   ":rule" pattern ... ":" pattern ... [ EOL build_block ]

variant          ::=   ":variant" name ( EOL variant_item ) ...
variant_item     ::=   INDENT varvalue EOL INDENT2 build_block

otherfile        ::=   includefile | childfile
```

```
includefile          ::=   ":include" item_list
childfile            ::=   ":child" item_list

build_command        ::=   assignment | script_item | generic_command | message

assignment           ::=   assign_set | assign_append | assign_cond
assign_set           ::=   name "=" item_list
assign_append        ::=   name "+=" item_list
assign_cond          ::=   name "?=" item_list

script_item          ::=   script_line | script_block
script_line          ::=   "@" script_command
script_block         ::=   ":python" EOL ( script_command EOL ) ... ":end"

generic_command      ::=   ":" command_name [ command_argument ]

message              ::=   info_msg | error_msg
info_msg             ::=   ":print" expression ...
error_msg            ::=   ":error" expression ...

# A list can contain white-separated items and Python style expressions.

item_list            ::=   item [ white_space item ] ...
item                 ::=   ( simple_item | "(" item_list ")" ) [ attribute ... ]
simple_item          ::=   ( expression | non_white_item ) ...
attribute            ::=   "{" name [ "=" expression ] "}"
expression           ::=   string_expr | python_expr
string_expr          ::=   """ text """ | "’" text "’"
python_expr          ::=   "‘" python-expression "‘"
non_white_item       ::=   non-white-text | "$" [ "(" ] name [ ")" ]
```

```
A backslash at the end of the line is used for line continuation.
- Can’t have white space after it!
- Also for comments.
- Also for Python commands; A leading @ char and white space before it is
  removed.
- Lines are joined before inspecting indents.

# comment \
continued comment

@ python command \
@   continued python command \
       still continued
```

When lines are joined because a command continues with a line with more indent, the line break and leading white space of the next line are replaced with a sinble space. An exception is when the line ends in "$br": The $br is removed, the line break is inserted in the text and leading white space of the next line is removed.

VARIABLES:

When concatenating variables and using rc-style expansion, the attributes of the last variable overrule the identical attributes of a previous one.

```
v1 = foo {check = 1}
v2 = bar {check = 2}
vv = $*v1$v2

-> vv = foobar{check = 1}{check = 2}
```

When using rc-style expansion, quotes will not be kept as they are, but removed and re-inserted where used or necessary. Example:

```
foo: "file 1.c" foo.c
:print "dir/$*source"
```

Results in: "dir/file 1.c" "dir/foo.c"

Be careful with using "$\" and quotes, you may not always get what you wanted.

To get one item out of a variable that is a list of items, use an index number in square brackets. Parenthesis or curly braces must be used around the variable name and the index. The first item is indexed with zero. Example:

```
BAR = beer coffee cola
:print $(BAR[0])
beer
BAR_ONE = $(BAR[2])
:print $BAR_ONE
cola
```

Using an index for which no item exists gives an empty result. When $MESSAGE includes "warning" a message is printed about this.

Normally using $VAR gets what you want. A-A-P will use the kind of quoting expected and add attributes when needed. However, when you want something else, this can be specified:
$var depends on where it’s used

$?var when the variable is not set or defined use an empty string instead of generating an error

$-var without attributes (may collapse white space)

```
$+var with attributes

$*var use rc-style expansion (may collapse white space)

$=var no quotes or backslashes
$’var aap quoted (using ’ and/or " where required, no backslashes)
$"var quoted with " (doubled for a literal ")
$\var special characters escaped with a backslash
   $!var depends on the shell, either like $’var or $"var

In most places $var is expanded as $+’var (with attributes, using ’ and " for
quoting).  The exceptions are:
:sys $-!var no attributes, shell quoting
$n in $(v[$n]) $-=var no attributes, no quoting
:del $-’var no attributes, normal quoting

The quoted variables don’t handle the backslash as a special character.  This
is useful for MS-Windows file names.  Example:

prog : "dir\file 1.c"
:print $’source

Results in:  "dir\file 1.c"



ASSIGNMENT:

overview:
   var = value     assign
   var += value     append (assign if not set yet)
   var ?= value     only assign when not set yet
   var $= value     evaluate when used
   var $+= value    append, evaluate when used
   var $?= value    only when not set, evaluate when used

Assignment with "+=" or "$+=" appends the argument as a separate item.  This
is actually done by inserting a space.  But when the variable wasn’t set yet
it works like a normal assignment:

VAR += something

is equal to:

@if globals().has_key("VAR"):
@   VAR = VAR + " " + "something"
@else:
@   VAR = "something"

Assignment with "?=" only does the assignment when the variable wasn’t set
yet.  A variable that was set to an empty string also counts as being set.
Thus when using "aap VAR=" the empty value overrules the value set with "?=".
```

```
VAR ?= something
```

is equal to:

```
@if not globals().has_key("VAR"):
    VAR = something
```

When using "$=", "$+=" or "$?=" variables in the argument are not evaluated at the time of assignment, but this is done when the variable is used.

```
VAR = 1
TT $= $VAR
VAR = 2
:print $TT
```

prints "2".

When first setting a variable with "$=" and later appending with "+=" the evaluation is done before the new value is appended:

```
VAR = 1
TT $= $VAR
TT += 2
VAR = 3
:print $TT
```

prints "1 2"

Note that evaluating a python expressions in “ is not postponed.

BLOCK ASSIGNMENT

The normal assignment command uses a single line of text. When broken into several lines they are joined together, just like with other commands. $br can be used to insert a line break. Example:

```
foo = first line$br
second line$br
third line $br
```

The block assignment keeps the line breaks as they are. The same example but using a block assignment:

```
foo << EOF
  first line
  second line
  third line
    EOF
```

The generic format is:

```
{var} << {term}
```

```
line1
...
{term}
```

{term} can be any string without white space. The block ends when {term} is found in a line by itself, optionally preceded by white space and followed by white space and a comment.

The amount of indent to be removed from all the lines is set by the first line. When the first line should start with white space use $empty.

All the variations of the assignment command can be used:

```
   var << term     assign
   var +<< term     append (assign if not set yet)
   var ?<< term     only assign when not set yet
   var $<< term     evaluate when used
   var $+<< term    append, evaluate when used
   var $?<< term    only when not set, evaluate when used
```

ATTRIBUTES

Items can be given attributes. The form is:

```
{name = value}
```

"value" is expanded like other items, with the addition that "}" cannot appear outside of quotes.

This form is also possible and uses the default value of 1:

```
{name}
```

Examples:

```
bar : thatfile {check = $MYCHECK}
foo {virtual} : somefile
```

The "virtual" attribute is used for targets that don’t exist (as file or directory) but are used for selecting the dependency to be build. These targets have the "virtual" attribute set by default:

TARGET	COMMONLY USED FOR
all	build the default targets
clean	remove intermediate build files
distclean	remove all generated files
test	run tests
check	same as "test"
install	build and install for use
tryout	build and install for trying out

reference generate or update the cross-reference database

fetch obtain the latest version of each file
update fetch and build the default targets

checkout checkout (and lock) from version control system
commit commit changes to VCS without unlocking
checkin checkin and unlock to VCS
unlock unlock files from a VCS
add add new files to VCS
remove remove deleted files from VCS
revise like checkin + remove
tag add a tag to the current version

prepare prepare for publishing (generated docs but no exe)
publish distribute all files for the current version

finally always executed last (using "aap finally" is uncommon)

These specific targets may have multiple build commands. They are all executed to update the virtual target. Normally there is up to one target in each (child) recipe.

Note that virtual targets are not related to a specific directory. Make sure no other item in this recipe or any child recipe has the same name as the virtual target to avoid confusion. Specifically using a directory "test" while there also is a virtual target "test". Name the directory "testdir" to avoid confusion.

The "comment" attribute can be used for targets that are to be specified at the command line. "aap comment" will show them.

```
% aap comment
target "all": build everything
target "foo": link the program
```

EXECUTING COMMANDS

A dependency and a rule can have a list of commands. For these commands the following variables are available:

$source The list of input files as a string.
$source_list The list of input files as a Python list.
$source_dl Only for use in Python commands: A list of
dictionaries, each input item is one entry.
$target The list of output files as a string.
$target_list The list of output files as a Python list.
$target_dl Only for use in Python commands: A list of
dictionaries, each output item is one entry.
$buildtarget The name of the target for which the commands are
executed. It is one of the items in $target.
$match For a rule: the string that matched with %

Example:

```
prog : "main file.c"
:print building $target from $source
```

Results in: building prog from "main file.c"
Note that quoting of expanded $var depends on the command used.

The Python lists $source_list and $target_list can be used to loop over each item. Example:

```
$OUT : foo.txt
@for item in target_list:
:print $source > $item
```

The list of dictionaires can be used to access the attributes of each item. Each dictionary has an entry "name", which is the (file) name of the item. Other entries are attributes. Example:

```
prog : file.c {check = md5}
@print sourcelist[0]["name"], sourcelist[0]["check"]
```

Results in: file.c md5

RATIONALE

In an assignment and other places a Python expression can be used in backticks. Expanding this is done before expanding $VAR items, because this allows the possibility to use the Python expression to result in the name of a variable. Example:

```
foovaridx = 5
FOO = $SRC‘foovaridx‘
Equal to:
FOO = $SRC5
```

In the result of the Python expression $ characters are doubled, to avoid it being interpreted as the start of a variable reference. Otherwise Python expressions with arbitrary results would always have to be filtered explicitly. Variables can still be obtained without using the $, although none of the modifiers like $+ and $* are available. Example:

```
FOO = foo/‘glob("*.tmp")‘
```

May result in:

```
FOO = foo/one.tmp two.tmp
```

Note that "foo/" is only prepended to the whole result, not each white-separated item. If you do want rc-style expansion, use two commands:

```
TT = ‘glob("*.tmp")‘
```

```
FOO = foo/$*TT

Result:
FOO = foo/one.tmp foo/two.tmp
```

Watch out for unexpected results when rc-style expansion is done for $*VAR.
Example:

```
VAR = one two
FOO = $*VAR/‘glob("*.tmp")‘
```

Would result in:

```
FOO = one/one.tmp two/one.tmp two.tmp
```

because the “ part is expanded first, thus the assignment is executed like:

```
FOO = $*VAR/one.tmp two.tmp
```

The backticks for a Python expression are also recognized inside quotes,
because this makes the rule for doubling backticks consistent. Example:

```
FOO = "this“file" that“file
```

Doubling the backtick to avoid it being recognized as a Python expression
makes it impossible to have an empty Python expression. There appears to be
no reason to use an empty Python expression.

Chapter 26. A-A-P Commands

Alphabetical list of Commands

This is the alphabetical list of all A-A-P commands. Common arguments are explained at the end.

Some commands can be used in a pipe. A pipe is a sequence of commands separated by ’|’, where the output of one command is the input for the next command. Example:

```
:cat foo | :eval re.sub(’this’, ’that’, stdin) | :assign bar
```

Unix tradition calls the output that can be redirected or piped "stdout". Reading input from a pipe is called "stdin".

In the commands below [redir] indicates the possibility to redirect stdout.

:action *action filetype-out* [*filetype-in*]

Define the commands for an action. See Chapter 22.

```
:do build {target = prog} foo.c
```

See :do for executing actions.

```
:add Version control command, see Chapter 16.
:addall Version control command, see Chapter 16.

:assign varname Assign stdin to a variable.  Can only be used after a
"|".

:attr {attrname} ... itemname ...
:attribute {attrname} ... itemname ...
Add the list of attributes "{attrname} ..." to each
item in the list of items "itemname ...".
Creates a node for each "itemname".

:attr itemname {attrname} ...
:attribute itemname {attrname} ...
Add attributes "{attrname} ..." to item "itemname".

      The above two forms can be mixed.  Example:

      :attr {fetch = cvs://} foo.c patch12 {constant}

      This adds the "fetch" attribute to both foo.c and patch12, and
      the "constant" attribute only to patch12.  This does the same in
      two commands:
```

```
:attr {fetch = cvs://} foo.c patch12
:attr {constant} patch12

Note: the attributes are added internally.  When using ":print
$var" this only shows the attributes given by an assignment, not
the ones added with ":attr".
```

:cat [*redir*] *fname*...

Concatenate the arguments and write the result to stdout. Files are read like text files. The "-" argument can be used to get the output of a previous pipe command. When redirecting to a file this output file is created before the arguments are read, thus you cannot use the same file for input.

See here for [redir].

```
:checkin Version control command, see Chapter 16.
:checkinall Version control command, see Chapter 16.
:checkout Version control command, see Chapter 16.
:checkoutall Version control command, see Chapter 16.

:child name Read recipe "name" as a child.  Mostly works like the
commands were in the parent recipe, with a few
exceptions:
- When "name" is in another directory, change to that
  directory and accept all items in it relative to
  that directory.
- Build commands defined in the child are executed in
  the directory of the child.  Thus it works as if
  executing the child recipe in the directory where it
  is located.
- Variables from the parent recipe can be used but
  changes to them will be used in the child only.
  New variables are local to the child recipe.  Except
  for the variables exported with ":export".
- Build commands are executed with the variables of
  the child recipe available, plus the variables of
  the toplevel parent.  For exported variables the
  value of the toplevel parent is used, for others the
  local values are used.
The "fetch" attribute is supported like with
":include".

:chmod [options] mode name ...
Change the protection flags of a file or directory.
Currently "mode" must be an octal number, like used by
the Unix "chmod" command.  Useful values:
755 executable for everyone, writable by
user
```

```
444 read-only
600 read-write for the user only
660 read-write for user and group

OPTIONS
{f} {force} don’t give an error when the file
doesn’t exist

:commit Version control command, see Chapter 16.
:commitall Version control command, see Chapter 16.

:copy [options] from ... to
Copy files or directory trees.  "from" and "to" may be
URLs.  This means :copy can be used to upload and
download a file, or even copy a file from one remote
location to another.  Examples:
:copy file_org.c  file_dup.c
:copy {r}  onedir  twodir
:copy *.c backups
:copy http://vim.sf.net/download.php download.php
:copy $ZIP ftp://upload.sf.net/incoming/$ZIP
:copy ftp://foo.org/README ftp://bar.org/mirrors/foo/README

When "from" and "to" are directories, "from" is
created in "to".  Unlike the Unix "cp" command, where
this depends on whether "to" exists or not.  Thus:

:copy {recursive} foo bar

will create the directory "bar/foo" if it doesn’t
exist yet.  If the contents of "foo" is to be copied
without creating "bar/foo", use this:

:copy {recursive} foo/* bar

OPTIONS
{f} {force} forcefully overwrite an existing file
or dir (default)
{e} {exist} {exists} don’t overwrite an existing
file or directory
{i} {interactive} before overwriting a local
file, prompt for confirmation
(currently doesn’t work for remote
files)
{u} {unlink} when used with {recursive}, don’t copy
a symlink, make a copy of the file or
dir it links to
{p} {preserve} preserve file permissions and
timestamps as much as possible
{r} {recursive} recursive, copy a directory tree.
"to" is created and should not exist
yet.
{q} {quiet} don’t report copied files
```

Wildcards in local files are expanded. This uses Unix style wildcards. When there is no matching file the command fails (also when there are enough other arguments).

When (after expanding wildcards) there is more than one "from" item, the "to" item must be a directory.

For "to" only local files, ftp:// and scp:// can be used. See "URLs" for info on forming URLs.

Attributes for "from" and "to" are currently ignored.

```
:del [options] file ...
:delete [options] file ...
```
Delete files and/or directories.

OPTIONS
{f} {force} don’t fail when a file doesn’t exist
{r} {recursive} delete directories and their contents recursively
{q} {quiet} don’t report deleted files

Wildcards in local files are expanded. This uses Unix style wildcards. When there is no matching file the command fails (also when there are enough other arguments).

CAREFUL: if you make a mistake in the argument, anything might be deleted. For example, accidentally inserting a space before a wildcard:

```
:del {r} dir/temp *
```

To give you some protection, the command aborts on the first error. Thus if "dir/temp" didn’t exist in the example, "*" would not be deleted.

```
:deldir [options] dir ...
```
Delete a directory. Fails when the directory is not empty.

OPTIONS
{f} {force} don’t fail when a directory doesn’t
exist; still fails when it exists but
is not a directory or could not be
deleted
{q} {quiet} don’t report deleted directories

:do *`action`* [*`fname`*...]

Execute an action. The commands executed may depend on the types of the input and/or output files. See Chapter 22.

```
:do build {target = prog} foo.c
```

See :action for defining actions.

```
:eval [redir] python-expression
Filter stdin using a Python expression.
When not used after "|" evaluate the Python
expression.
The Python expression is evaluated as specified in the
argument.  The "stdin" variable holds the value of the
input as a string, it must be present when ":eval"
is used after "|".
$var items are not expanded before evaluating the
python expression.  To get the value of $var use
"var" as a Python variable.
The expression must result in the filtered string or
something that can be converted to a string with
str().  This becomes stdout.  The result may be empty.
Examples:

:print $foo | :eval re.sub('<.*?>', '', stdin) > tt
:eval os.name | :assign OSNAME

Note that the expression must not contain a "|"
preceded by white space, it will be recognized as a
pipe.  Also there must be no ">" preceded by white
space, it will be recognized as redirection.
$VAR items will be expanded in the filter command.


:execute name [ argument ] ...
Execute recipe "name" right away.  This works like
restarting aap, except that variables are passed to
the recipe.  ":export" can be used to get variables
from the executed recipe into the current recipe.
The "fetch" attribute is supported like with
":include".
Optional arguments may be given, like on the command
line.  This is useful for specifying targets and
variable values. "-f recipe" is ignored.
Example:
TESTPROG = ./myprog
:execute test.aap test1 test2

This command is useful when a recipe does not contain
dependencies that interfere with sources and targets
```

```
in the current recipe.  For example, to build a
command the current recipe depends on.  For example,
when the program "mytool" is required and it doesn’t
exist yet, execute a recipe to build and install it:

@if not program_path("mytool"):
:execute mytool.aap install
:sys mytool

Another example: build two variants:

:execute build.aap GUI=motif
:execute build.aap GUI=gtk

:exit Quit executing recipes.  When used in build commands,
the "finally" targets will still be executed.  But a
":quit" or ":exit" in the commands of a "finally"
target will quit further execution.

:export varname ... When used at the toplevel, export variable "varname"
with its current value to the recipe that uses the
current recipe as a child or executes it.
When used in build commands, export variable "varname"
to the recipe or build commands that invoked and
export to the toplevel of the current recipe.  Useful
to pass a value to the "finally" target or remember a
value for the next invocation.
When the variable was assigned a value with "$=" the
argument will be evaluated now.
Note: When exporting a variable that contains the name
of a file, a relative file name will not be valid in
another directory.  When using these commands:
:child dir/main.aap
:cat $RESULT
using the following in dir/main.aap causes trouble:
RESULT = foo.txt bar.txt
export RESULT
Instead use:
RESULT = foo.txt bar.txt
RESULT = ‘aap_abspath(RESULT)‘
export RESULT


:global name ... Define variable "name" to be global to other recipes.
Even when "name" has been assigned a value, build
commands (actions, dependencies or rules) will still
use the value passed on by who invoked the build
commands.  Example:
CFLAGS = -O4
:global CFLAGS
This means that when the value of CFLAGS is changed
this value will be used for build commands defined in
this recipe.
```

```
:include name Read recipe "name" as if it was included in the
current recipe.  Does not change directory and file
names are considered to be relative to the current
recipe, not the included recipe.
The "fetch" attribute can be used to specify a list
of locations where the recipe can be fetched from.

:local name ... Define variable "name" to be local to this recipe.  It
will not be passed on to child recipes.  Build commands
defined in this recipe (actions, dependencies or rules)
will use the value from the recipe instead of a value
passed from who invoked the commands.

:mkdir [options] dir ...
Create directory.  This fails when "dir" already
exists and is not a directory.
A "mode" attribute on a directory can be used to
specify the protection flags for the new directory.
Example:
:mkdir {r} ~/secret/dir {mode = 0700}
The default mode is 0644.  The effective umask may
reset some of the bits though.

OPTIONS
{f} {force} don’t fail when a directory already
exist; still fails when it is not a
directory or could not be created
{q} {quiet} don’t report created directories
{r} {recursive} also create intermediate directories,
not just the deepest one

Note: automatic creation of directories can be done by
adding the {directory} attribute to a source item.

:move [options] from ... to
Move files or directories.  Mostly like ":copy",
except that the "from" files/directories are renamed
or, when renaming isn’t possible, copied and deleted.

OPTIONS
{f} {force} forcefully overwrite an existing file
or directory (default)
{e} {exist} {exists} don’t overwrite an existing
file or directory
{i} {interactive} before overwriting a local
file, prompt for confirmation
(currently doesn’t work for remote
files)
{q} {quiet} don’t report moved files
```

:print [*redir*] [*text*...]

Print the arguments on stdout. Without arguments a line feed is produced. $var items are expanded, otherwise the arguments are produced literally, including quotes:

```
:print "hello"
```

results in:

"hello"

Leading white space is skipped, but white space in between arguments is kept. To produce leading white space start with the "empty" variable:

```
:print $empty   indented text
```

results in:

indented text

When used in a pipe the `stdin` variable holds the input.

See here for [redir].

```
:publish file ...
:publish {attribute} ... file ...
Publish the files mentioned according to their
"publish" or "commit" attribute.
Creates directories when needed (for CVS only one
level).

:publishall file ...
Like ":publish" but also remove files that are not
an argument.  Careful!

:python
    python-commands A block of Python code.  The block ends when the indent
drops to the level of ":python" or below.

:python  {term}
python-commands
   {term} A block of Python code.  The block ends when {term} is
found on a line by itself.  The Python commands may
have any indent.
White space before and after {term} is allowod and a
comment after {term} is also allowed.  {term} can
contain any characters but no white space.

:quit See ":exit".

:recipe {fetch = URL ... }
Location of this recipe.  The "fetch" attribute is
used like with ":child": a list of locations.  The
```

first one that works is used.
When aap was started with the "fetch" argument,
fetch the recipe and restart reading it. Using the
"fetch" or "update" target causes this as well.
The commands before ":recipe" have already been
executed, thus this may cause a difference from
executing the new recipe directly. The values of
variables are restored to the values before executing
the recipe.
Fetching the recipe is done only once per session.

:fetch file ...
Fetch the files mentioned according to their
"fetch" or "commit" attribute. When a file does not
have these attributes or fetching fails you will get
an error message.
Files that exist and have a "fetch" attribute with
value "no" are skipped.
The name "." can be used to update the current
directory:
:fetch . {fetch = cvs://$CVSROOT}
:fetch {attribute} ... file ...
Like above, apply {attribute} to all following items.

:remove Version control command, see Chapter 16.
:removeall Version control command, see Chapter 16.
:reviseall Version control command, see Chapter 16.

:rule tpat ... : spat ...
commands
Define a rule to build files matching the pattern
"tpat" from a file matching "spat".
There can be several "tpat" patterns, the rule is used
if one of them matches.
There can be several "spat" patterns, the rule is used
if they all exist (or no better rule is found).
When "commands" is missing this only defines that
"tpat" depends on "spat".
Can only be used at the toplevel.
The "skip" attribute on ’tpat’ can be used to skip
certain matches.
$target and $source can be used in "commands" for the
actual file names. $match is what the "%" in the
pattern matched.
Alternative: instead of matching the file name with a
pattern, ":action" uses filetypes to specify commands.

:start cmds Like ":sys" and ":system", but don’t wait for the
commands to finish. Errors are ignored.
Runs in the same terminal, which will cause problems
when the command waits for input. Open a new terminal
to run that command in. Example:

```
:start xterm -e more README

WARNING: Using ":start" probably makes your recipe
non-portable.

:sys [options] cmds
:system [options] cmds
Execute "cmds" as system (shell) commands.  Example:

:system filter <foo >bar
:sys echo one
two

The following lines with more indent are appended,
replacing the indent with a single space.  Example:

:sys echo one
two

This echos "one two".

Options:
    {i} or {interactive}: don’t log output (see below)
    {q} or {quiet}: Don’t echo the command
    {l} or {log}: Redirect all output to the log file,
                  do not echo it
{interactive} and {log} cannot be used at the same
time.

Output is logged by default.  If this is undesirable
(e.g., when starting an interactive command) prepend
"{i}" or "{interactive}" to the command.  It will be
removed before executing it.  Example:

:system {i} vi bugreport

When the "async" variable is set and it is not empty,
":sys" works like "start", except that consecutive
commands are executed all at once in one shell.

WARNING: Using ":sys" or ":system" probably makes your
recipe non-portable.

:syseval [redir] command
Execute shell command "command" and write its output
to stdout.  Only stdout of the command is captured.
When {stderr} is just after the command name, stderr
is also captured.  Example:

:syseval hostname | :assign HOSTNAME

When used in a pipe, the stdin is passed to the
```

command. Example:

```
:print $var | :syseval sort | :assign var
```

Note the difference with the ":sys" command:
redirection in ":sys" is handled by the shell, for
":syseval" it is handled by aap.

When executing the command fails, the result is empty.
The exit value of the command is available in $exit.

WARNING: Using ":syseval" probably makes your recipe
non-portable.

:syspath path arg ...
Use "path" as a colon separated list of commands, use
the first one that works.
When %s appears in "path", it is replaced with the
arguments. If it does not appear, the arguments are
appended.
Other appearences of % in "path" are removed, thereby
reducing %% to % and %: to : while avoiding their
special meaning.
Don’t forget that "path" must be one argument, use
quotes around it to include white space.
Example:

```
:syspath ’vim:vi:emacs’ foobar.txt
```

Output is not logged.
Note: on MS-Windows it’s not possible to detect if a
command worked, the first item in the path will always
be used.

WARNING: Using ":syspath" probably makes your recipe
non-portable.

:tag Version control command, see Chapter 16.
:tagall Version control command, see Chapter 16.

:tee [redir] fname ...
Write stdin to each file in the argument list and also
write it to stdout. This works like a T shaped
connection in a water pipe. Example:

```
:cat file1 file2 | :tee totfile | :assign foo
```

:touch [options] name ...
Update timestamp of file or directory "name".

OPTIONS
{f} {force} create the file when it doesn’t exist
{e} {exist} create the file when it doesn’t exist,

```
don’t update timestamp when the file
already exists
If "name" doesn’t exist and {force} and {exist} are
not present the command fails.
If "name" doesn’t exist and {force} or {exist} is
present an empty file will be created.
If "name" does exist and {exist} is present nothing
happens.
A "directory" attribute can be used to specify a
non-existing "name" is to be created as a directory.
There is no check if an existing "name" actually is a
directory.
A "mode" attribute can be used to specify the mode
with which a new file or directory is to be created.
The value is in the usual octal form, e.g., "0644".

:unlock Version control command, see Chapter 16.
:unlockall Version control command, see Chapter 16.

:update [{force}] target ...
Update "target" now, if it is outdated or when
"{force}" is used.
One or more targets can be specified, each will be
updated.
When this appears at the top level, a dependency or
rule for the target must already have been specified,
there is no look-ahead.

:verscont Version control command, see Chapter 16.
```

Common arguments for Commands

[*redir*]

Redirect the output of a command. Can be one of these items:

> *fname*	write output to file "fname"; fails when "fname" already exists
>! *fname*	write output to file "fname"; overwrite an existing file
>> *fname*	append output to file "fname"; create the file if it does not exist yet
| *command*	pipe output to the following "command"

The redirection can appear anywhere in the argument, except inside quotes. The normal place is either as the first or the last argument. The pipe to the next command must appear at the end.

The file name can be a URL. The text will first be written to a local file and then the file is moved to the final destination.

The white space before the file name may be omitted. White space before the ">" and "|" is required. To avoid recognizing the ">" and "|" for redirection and pipes, use $gt and $pipe.

When a command produces text on stdout and no redirection or pipe is used, the stdout is printed to the terminal.

```
URLs

In various places URLs can be used to specify remote locations and the method
how to access it.

http://machine/path HTTP protocol, commonly used for web sites.
read-only
"machine" can also be "machine:port".

ftp://machine/path FTP protocol.
"machine" can also be "machine:port".

For authentication the ~/.netrc file is used if
possible (unfortunately, the Python netrc module has a
bug that prevents it from understanding many netrc
files).

Alternatively, login name and password can be
specified just before the machine name:
ftp://user@machine/path
ftp://user:password@machine/path
When ":password" is omitted, you will be prompted for
entering the password.
Either way: ftp sends passwords literally over the
net, thus this is not secure!  Should use "scp://"
instead.

scp://machine/path SCP protocol (using SSH, secure shell).
Requires the "scp" program installed.
Additionally a user name can be specified:
scp://user@machine/path
"path" is a relative path to the directory where "ssh"
logs in to.  To use an absolute path prepend a slash:
scp://machine//path
The resulting path for the "scp" command uses a ":"
instead of the first slash.
```

Chapter 27. Common Attributes

This is the first paragraph of Common Attributes.

For the "virtual" and "comment" see Chapter 25.

```
Attributes can be added to a node with the ":attr" command and by using them
in a dependency or rule.  Note that an assignment does not directly associate
the attribute with a node.  This only happens when the variable is used in an
":attr" command or dependency.


STICKY ATTRIBUTES

When attributes are used in a rule or dependency, most of them are only used
for that dependency.  But some attributes are "sticky": Once used for an item
they are used everywhere for that item.  Sticky attributes are:
virtual virtual target, not a file
remember virtual target that is remembered
directory item is a directory
filetype type of file
constant file contents never changes
fetch list of locations where to fetch from (first
one that works is used)
commit list of locations for VCS
publish list of locations to publish to (they are all
used)
force rebuild a target always
depdir directory to put an automatically generated
dependency file in; when omitted $BDIR is used
signfile file used to store signatures for this target


SIGNATURES: the check attribute

The default check for a file that was changed is an md5 checksum.  Each time a
recipe is executed the checksums for the relevant items are computed and
stored in the file "aap/sign".  The next time the recipe is executed the
current and the old checksums are compared.  When they are different, the
build commands are executed.  This means that when you put back an old version
of a file, rebuilding will take place even though the timestamp of the source
might be older than the target.

Another check can be specified with {check = name}.  Example:

foo.txt : foo.db {check = time}
:sys db_extract $source >$target

Other types of signatures supported:

time Build the target when the timestamp of the source
differs from the last time the target was build.
```

```
newer Build the target if its timestamp is older than the
timestamp of the source.  This is what the good old
"make" program uses.

md5 Build the target if the md5 checksum of the source
differs from the last time the target was build.
This is the default.

c_md5 Like "md5", but ignore changes in comments and amount
of white space.  Appropriate for C programs. Slows
down computations considerably.

none Don’t check time or contents, only existence.  Used
for directories.

When mixing "newer" with other methods, the build rules are executed if the
target is older than the source with the "newer" check, or when one of the
signatures for the other items differs.

The "aap/sign" file is normally stored in the directory of the target.  This
means it will be found even when using several recipes that produce the same
target.  But for targets that get installed in system directories (use an
absolute path), virtual targets and remote targets this is avoided.  For these
targets the "aap/sign" file is stored in the directory of the recipe that
specifies how to build the target.

To overrule the directory where "AAP/sign" is written, use the attribute
{signdirectory = name} for the target.
To overrule the file where the signatures are written, use the attribute
{signfile = name} for the target.  "name" cannot end in "sign".
```

Chapter 28. Common Variables

This is the first paragraph of Common Variables.

```
STANDARD VARIABLES:

name default info
$$ $ a single $
$# # a single #
$lt < a single <
$gt > a single >
$br \n a line break
$bar | a single |
$pipe | a single |
$empty Empty.  Can be used to get leading white space with
":print".

$AAPVERSION version of A-A-P the recipe was written for
$EXESUF suffix for an executable file (".exe" for MS-Windows)
$OBJSUF suffix for an object file (".o" for Unix, ".obj" for
MS-Windows)
$LNKSUF suffix for a (symbolic) link (empty)

$CC cc command to execute the compiler
$CFLAGS arguments always used for $CC
$LDFLAGS arguments for $CC when linking, before the object files
$LIBS arguments for $CC when linking, after the object files
$CPPFLAGS arguments for $CC when compiling sources (not when
linking objects)

$GMTIME GMT time in seconds since 1970 Jan 1
(the time is set once, it remains equal while
executing)
$DATESTR Date as a string in the form "2002 Month 11"
$TIMESTR Time as a string in the form "23:11:09" (GMT)
$OSTYPE Type of operating system used:
posix Unix-like (Linux, BSD)
mswin MS-Windows (98, XP)
msdos MS-DOS (not MS-Windows)
os2 OS/2
mac Macintosh
java Java OS
riscos Risc OS
ce MS-Windows CE

$TARGET list of target files, usually the name of the
resulting program
$SOURCE list of source files

$CACHE List of directories to search for cached downloaded
files.  Default for Unix:
/var/aap/cache  ~/.aap/cache  aap/cache
```

```
For MS-Windows, OS/2:
$HOME/aap/cache  aap/cache
Directories that don’t exist are skipped.
When using a relative name, it is relative to the
current recipe.  Thus the recipe specified with
":child dir/main.aap" uses a different cache
directory.
See CACHE below.

When this variable is set, currently cached files are
flushed.  Otherwise this only happens when exiting.
Thus this command can be used to flush the cache:
CACHE = $CACHE

$cache_update Timeout after which cached files may be downloaded
again.  See CACHING below.

Variables set by A-A-P:

$targetarg Target(s) specified on the command line.

$aapversion Version number of A-A-P.  E.g., 31 (version 0.031) or
1003 (version 1.003)
```

IV. Appendixes

Appendix A. License