

NASA CR-197763

NASA/WVU Software IV & V Facility Software Research Laboratory Technical Report Series

NASA-IVV-94-006
WVU-SRL-94-006
WVU-SCS-TR-95-6
CERC-TR-RN-94-012

*IN-61-016
48511
P- 15*

Software Packager User's Guide

by John R. Callahan

(NASA-CR-197763) SOFTWARE PACKAGER
USER'S GUIDE (West Virginia Univ.)
15 p

N95-26693

Unclas

G3/61 0048511

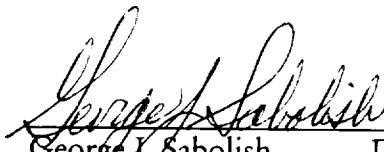


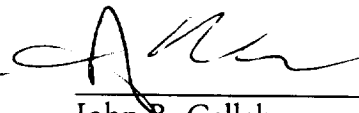
National Aeronautics and Space Administration



West Virginia University

According to the terms of Cooperative Agreement #NCCW-0040,
the following approval is granted for distribution of this technical
report outside the NASA/WVU Software Research Laboratory


George J. Sabolish Date
Manager, Software Engineering 4-10-95


John R. Callahan Date
WVU Principal Investigator 4-19-95

Software Packager User's Guide

John R. Callahan¹
Concurrent Engineering Research Center
West Virginia University

Abstract

Software integration is a growing area of concern for many programmers and software managers because the need to build new programs quickly from existing components is greater than ever. This includes building versions of software products for multiple hardware platforms and operating systems, building programs from components written in different languages, and building systems from components that must execute on different machines in a distributed network. The goal of software integration is to make building new programs from existing components more *seamless* --- programmers should pay minimal attention to the underlying configuration issues involved.

Libraries of reusable components and classes are important tools but only partial solutions to software development problems. Even though software components may have compatible interfaces, there may be other reasons, such as differences between execution environments, why they cannot be integrated. Often, components must be adapted or reimplemented to fit into another application because of *implementation* differences --- they are implemented in different programming languages, dependent on different operating system resources, or must execute on different physical machines.

The software packager is a tool that allows programmers to deal with interfaces between software components and ignore complex integration details. The packager takes modular descriptions of the structure of a software system written in the PACKAGE specification language and produces an integration program in the form of a MAKEFILE. If complex integration tools are needed to integrate a set of components, such as remote procedure call stubs, their use is implied by the packager automatically and stub generation tools are invoked in the corresponding MAKEFILE. The programmer deals only with the components themselves and not the details of how to build the system on any given platform.

¹Supported by the Advanced Research Projects Agency Grant MDA 972-91-J-1022, the National Aeronautics and Space Administration Grant NAG 5-2129, and the National Library of Medicine Grant N01-LM-3-3525.

1 Introduction

Software packaging [1] is an attempt to present a unified solution to the problems of building software applications from reusable software components. Software packaging hides the difficult problems of "putting the pieces together" that often prevent reuse, make configuration management difficult, and prevent distribution of run-time components. The software packager creates an integration package specialized for each constraint imposed by a specific run-time environment. For example, from a single description of the software structure, the packager produces different MAKEFILES [2] for each hardware platform and operating system. These MAKEFILES handle the special integration cases for each environment such as the location of include files, tools, libraries, etc.

The software packager tool (i.e., *package* in the UNIX shell) is a utility for generating integration programs (e.g., MAKEFILES) from descriptions of software system structures that can be ported between execution platforms. The packager relies on platform-specific rules to determine how to integrate software components. However, unlike the *imake* tool [3] that also uses platform-specific rules to integrate a portable application, the software packager allows more complex integrations of alternative implementations as well as heterogeneous and distributed components.

The software packager is designed to be adopted incrementally so that existing applications can be quickly and easily packaged. Programmers can later take advantage of more advanced features of software packaging. In the first examples below, we will deal with homogeneous integrations that should be familiar to programmers familiar with the *make* tool. In later sections, we will introduce more complex integrations.

2 Example

We introduce software packaging with a simple example of integrating two homogeneous source components written in the C programming language. Although this is a simple example where two homogeneous programs are integrated, it helps illustrate software packaging in terms of existing methodologies. Given the source program (see *User's Guide Example A*):

```
main()
{
    int i = 5;
    printf("The factorial of %d is %d\n", i, ifact(i));
}
```

in a file `main.c` and the source program

```
ifact(x)
    int x;
{
    if(x < 1) return 1;
    else return (x*ifact(x-1));
}
```

in the file `fact.c`, then the software packager can be used to build an executable file (e.g., `a.out`) from these objects. Instead of a MAKEFILE, the programmer describes the structure of the application in a PACKAGE file. The software packager looks for this software structure description in the special file

Package in the current working directory. Our program has a simple modular structure that is described in the Package file in the PACKAGE specification language as follows:

```
import stdpkg;

module Main;
module Fact;

implement Root as {
    Main: main;
    Fact: f;
}

implement Main with cmain {
    FILE=main.c
}

implement Fact with cfunc {
    FILE=fact.c
}
```

This specification describes an application comprised of instances of two modules (the syntax of package specifications are described in the next section.). Each module has a single implementation corresponding to a source file written in the C programming language. The identifiers `cmain` and `cfunc` correspond to types of components that implement modules. Given this description, the packager determines if and how to build the executable. It uses a set of rules that is specific to each execution environment. For example, in our environment (i.e., a UNIX based platform), a single `cmain` object and 0 or more `cfunc` objects can be integrated together into an executable using the C compiler and linker. The use of these tools, however, is implied because the packager constructs the necessary MAKEFILE automatically.

To produce the MAKEFILE automatically, invoke the *package* command from the shell

```
% package
```

in the directory with the Package specification. The software packager will generate the MAKEFILE shown in Appendix A. The application can then be integrated by invoking

```
% make
```

to build the executable file *a.out*.

3 Basic Package Specifications

A package specification describes the structure of an application in terms of its components without describing which implementations are used for different components and without specifying how the application is integrated on various execution platforms. The goal is to express a truly portable description of a software application.

In general, a package specification consists of module declarations and their associated implementations. In the previous example, two modules --- `Main` and `Fact` --- are declared. Another module, called `Root`, is automatically declared in the imported package `stdpkg`. The remainder of the package consists of implementations for the modules `Root`, `Main`, and `Fact`. The implementation of `Root` is called a composite implementation. The implementations of `Main` and `Fact` are called primitive implementations.

We must alter the PACKAGE specifications presented above because they are incomplete. One problem is that the modules **Main** and **Fact** are unelaborated, i.e., they have no formal interfaces. For some integrations, particularly homogeneous integrations such as above, modules can be left unelaborated. The packager does not require complete interface descriptions of modules in order to be more compatible with existing practices. For more complex integrations involving heterogeneous or distributed components, however, it is necessary to fully describe an interface in terms of the functions used and defined by a module. We must change the PACKAGE specification of the application as follows (see *User's Guide Example B*):

```

import stdpkg;

module Main {
    use ifact(int) (int);
}
module Fact {
    def ifact(int) (int);
}

implement Root as {
    Main: main;
    Fact: f;
    bind main'ifact to f'ifact;
}

implement Main with cmain {
    FILE=main.c
}

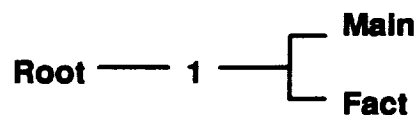
implement Fact with cfunc {
    FILE=fact.c
}

```

The added lines in the new PACKAGE specification are shown in bold. They show that the **Main** module "uses" a function called **ifact** that has an integer as formal parameter and produces an integer result. Likewise, the **Fact** module supplies an **ifact** function with an identical syntax. In the composite implementation of the **Root** module, the **bind** clause specifies that all uses and definitions of functions in all instances of modules be linked together. Further details of module declarations and the **bind** statement will be explained in later sections.

3.1 Software Structure Graphs

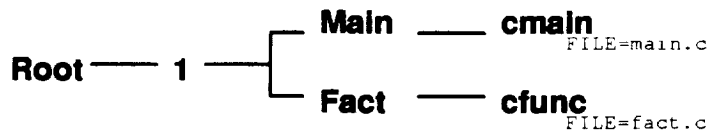
In general, a PACKAGE specification describes the modules and their logical connections in an application. A PACKAGE specification for an application corresponds to a structure called a *software structure graph* rooted at the module **Root**. The **implement Root as** clause is a composition of two module instances that comprise the first level of the application. The form **implement X as** is used to specify a composite implementation for any module **X**. For example, in the specification above, the application's **Root** module as implemented by a composite system comprised of two module instances: an instance of a **Main** module (named **main**) and an instance of **Fact** module (named **f**). Thus, at the highest level, the structure graph for this application looks like:



The additional node "1" in the structure graph is added to distinguish this composite implementation of **Root** from other possible implementations. All modules may have multiple, alternate implementations.

The "1" subtree of **Root** is only one possible implementation of the module **Root** (hence one possible implementation of the entire application as well). It is possible to specify alternative implementations for **Root** by using additional **implement Root** clauses.

Primitive implementations for modules can also be specified with the form **implement X with Y**. A primitive implementation corresponds to an object of type **Y** in the execution environment that implements a module **X**. The object type **Y** may correspond to *any artifact of any type* in the environment ---- a source code file, service, port number, thread, executable program, or a data file. Users may define their own objects types (see the **object** statement for details). Object types may have various associated properties like **FILE**, **LANGUAGE**, **LOCATION**, and **VERSION**. The object types **cmain** and **cfunc** are defined in the imported **stdpkg** file. Primitive implementations of modules are found at the leaves of a software structure graph because they cannot be further elaborated. The last two clauses in the **PACKAGE** specifications above are primitive implementations of the **Main** and **Fact** modules respectively. With these implementations, the final software structure graph for our example looks like:



In this structure graph, all modules have only single implementations. It is possible to assign multiple implementations to any module including the **Root** module. The software packager will choose appropriate implementations for each module based on integration rules specific to a particular execution platform (e.g., machine and operating system). For example, suppose we implement the **Main** module in FORTRAN (see *User's Guide Example C*):

```

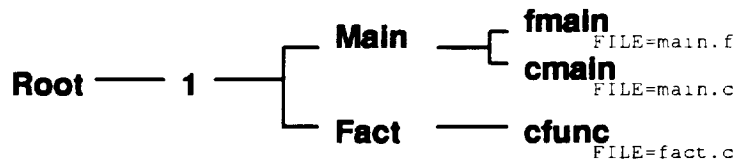
      n = 5
      nresult = ifact(n)
      write(6,10) n,nresult
10    format('The factorial of ',I5,' is ',I5)
      end
  
```

and add the following primitive implementation to the Package specification:

```

implement Main with fmain {
    FILE=main.f
}
  
```

for the **Main** module where the object type **fmain** has associated properties appropriate for a FORTRAN implementation with a main entry point. The FORTRAN implementation in **main.f** must support the **Main** module specification. This alternate implementation for the **Main** module would make the structure graph look like



The packager will choose the appropriate collection of implementations that are viable (i.e., they can be integrated) in the target environment. By analogy, software packaging is like compiling a program. A compiler translates source code into a machine program that can be assembled into an executable program. A packager translates a software system description into an integration program that can be invoked to build the executable program.

In the case where two or more implementations are viable for a module, the packager chooses nondeterministically between them. To force the packager to distinguish between implementations, a programmer can use constraints to eliminate candidate implementations. Constraints are described in a later section.

Our examples have illustrated basic packager techniques in which two software components are integrated. While homogeneous integration can be done with existing tools (e.g., MAKE and IMAKE), the packager can be used to specify and build more complex, heterogeneous, and distributed configurations. Homogeneous integration is simply the degenerate case of the more general concept of software packaging. Packaging subsumes these existing techniques and expands on them into more general forms of software integration.

3.2 Attributes

All components in a package specification can be assigned attributes (i.e., key-value pairs). The scope of an attribute consists of the entire software structure subgraph below the node in which the attribute is declared. For example, if we declare the attribute **ARCH=sparc** in the **Root** composite implementation, then the **ARCH** attribute is visible in both the primitive implementation nodes of the **Main** and **Fact** modules (see *User's Guide Example D*):

```
...
implement Root as {
    ARCH=sparc
    Main: main;
    Fact: f;
}

implement Main with cmain {
    FILE=main-$(ARCH).c
}
...
```

Again, the value of the **FILE** attribute of the **cmain** implementation of the **Main** module would be **main-sparc.c**. The reference to the attribute **\$(ARCH)** in the primitive implementation of **Main** searches up the software structure graph for a declaration of **ARCH**.

The results of shell commands can also be assigned to attributes. The form of a standard attributes assignment is (see *User's Guide Example E*):

```
ARCH=sparc
```

To assign the result of a shell command, use the form

```
ARCH := arch
```

where the right-hand side is a valid UNIX shell command. In this example, the **ARCH** attribute is assigned the hardware architecture type using the UNIX *arch* command.

3.3 Parameters

Modules and their implementations in **PACKAGE** specifications can have formal parameters that are bound to actual parameters when a module instance is created in a composite implementation. For example, consider the following **PACKAGE** specification for a simple example in which the source file name is dependent on a parameter (see *User's Guide Example F*):


```

...
module Main(x);
...
implement Root as (
    Main: main(sparc);
    ...
)

implement Main(x) with cmain (
    FILE=main-$(x).c
)
...

```

where the **FILE** attribute of the **cmain** implementation of the **Main** module would be `main-sparc.c`. Formal parameters can also be used in composite implementations of modules. To access the value of a formal parameter the form `$(x)` is used where `x` is the name of the formal parameter. This form is identical to accessing the values of graph-scoped attributes.

3.4 Constraints

Attributes are useful, but their full utility is realized when combined with constraints. Constraints may be declared at any node in a software structure graph. The scope of a constraint is the same as the scope of an attribute: all subgraphs below the current node in the software structure graph. For example, if we want to restrict the selection of a particular implementation for the **Main** module to a fast implementation (e.g., an implementation using static arrays instead of dynamic allocation), the constraint `ARCH==sparc` can be set in the composite implementation for the **Root** module as follows (see *User's Guide Example G*):

```

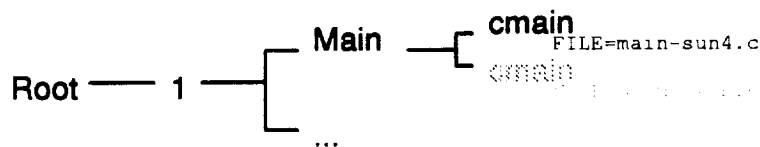
...
implement Root as (
    ARCH==sparc
    Main: main;
    Fact: f;
)

implement Main with cmain (
    ARCH=sg1
    FILE=main-iris.c
)

implement Main with cmain (
    ARCH=sparc
    FILE=main-sun4.c
)
...

```

In this case, the second implementation of **Main** is chosen because it satisfies the higher level constraint on the **ARCH** attribute. The resulting manufacture graph would be



The alternative implementation of the `Main` module was eliminated from consideration (shown in gray). This technique has also been used to maintain and package multiple versions of an application from a single Package specification.

3.5 Object Types

The primitive implementation types `cmain` and `cfunc` are called object types and they are declared in the `stdpkg` imported file as:

```
object cmain : cobj {
    ENTRY=main
}
object cfunc : cobj;
```

where both object types inherit attributes from a parent type `cobj` that is defined as

```
object cobj : object {
    LANGUAGE=C
}
```

These object types are declared in the `stdpkg` imported file. Users may declare their own object types and corresponding attribute assignments.

3.6 Module Ports

One of the more advanced features of software packaging deals with its capabilities as a module interconnection language (MIL). In the first example, the `Main` and `Fact` modules were unelaborated, i.e., they were declared without any formal module interface:

```
module Main;
module Fact;
```

But we then elaborated the interfaces for these modules as a series of ports that describe the use and definitions of functions as follows:

```
module Main {
    use ifact(int)(int);
}
module Fact {
    def ifact(int)(int);
}
```

The elaborated module interfaces describe the resources used and provided by a component so that it is completely encapsulated. This approach is different than traditional object-oriented programming where only provided resources (e.g., methods) are part of an object's interface description. Module interconnection languages like the PACKAGE specification language subsume the object-oriented approach because module instances can be nested in composite implementations of more abstract modules.

3.7 Bindings

Module interface ports are connected together with the use of binding statements in composite implementations (see *User's Guide Example H*):

```
implement Root as {
    Main: main;
    Fact: f;
    bind main'ifact to f'ifact;
}
```

The packager produces a file named `a.pkg` (the default name) that contains all the bindings in a specification. The file is accessed by stub generators during the integration phase (e.g., during the execution of the `MAKEFILE`). At that phase, other tools can determine if the integration is viable. For example, the package specifications:

```
import stdpkg;

module Main {
    use ifact(int)(int);
}
module Fact {
    def ifact(int)(int);
}

implement Root as {
    Main: main;
    Fact: f;
    bind main'ifact to f'ifact;
}

implement Main with cmain {
    FILE=main.c
}

implement Fact with rpcsvc {
    PROGRAM=4516
    VERSION=1
    LOCATION=hopper.cs.wvu.edu
}
```

During the integration process, external tools like stub generators can use the `a.pkg` file to generate the needed "glue" code. Such extra code is often needed to implement connections between components in run-time execution environments.

4 Package Imports

Files imported into `PACKAGE` specifications are found in a directories named by the `PKGIMPORT` environment variable or by using the `-I` command line option. Unlike the include mechanism in the C preprocessor, the packager does not simply expand the imported file inline. Instead, it searches the import directories for files whose extension is `".pko"` (package object file). This file is then imported into the packaging process during construction of the software structure graph.

5 Package Library

If the environment variable `PKGLIBRARY` is set or the `-L` option is used, the packager will search the specified directories for all files `X.pko` where `X` is the name of any module that needs expanding in the software structure graph. Thus, if we specify the command:

```
% package -L /usr/local/lib
```

in our previous example of the factorial application, the packager will search the directory `/usr/local/lib` for the files `Main.pko` and `Fact.pko`. In this fashion, developers can transparently share reusable components as alternative implementations in their applications.

6 Basic Rule Specifications

Normally, developers should not be concerned with packager rule specifications. Skip this section if you are not a system administrator. Software packager rules are relatively fixed for any execution environment and shared by all developers in the environment. Rules resemble attribute grammars because

they describe the abstract form of integration processes in an environment, e.g., the abstract form of legal MAKEFILES. For example, the rules used to produce the Makefile shown in Appendix A is shown in Appendix B.

In general, a rules file is a "grammar" consisting of individual rules such as

```

exec      :      cmain cfuncs
          :      {
          :      ...
          :      }
          ;

```

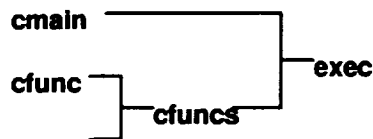
where `exec` is on the left-hand side of the rule, `cmain` and `cfuncs` are on the right-hand side of the rule. The labels `exec` and `cfuncs` represent non-terminal object types. Non-terminal object types are found on the left-hand side of rules. The `cmain`, however, label represents a terminal object type. It is not on the left-hand side of any rule.

The goal of the packager is to find a tree whose internal nodes correspond to rules in the abstract grammar and whose leaves correspond to a complete set of primitive implementations in a given software structure graph. The resulting tree, if found, is called a *software manufacture graph*. The packager operates in two phases:

1. Perform a depth first, backtracking search of the rules to find nodes corresponding to *all* leaf nodes of the structure graph.
2. If a complete graph is found, then execute actions associated with the graph starting at the `exec`² node in the manufacture graph.

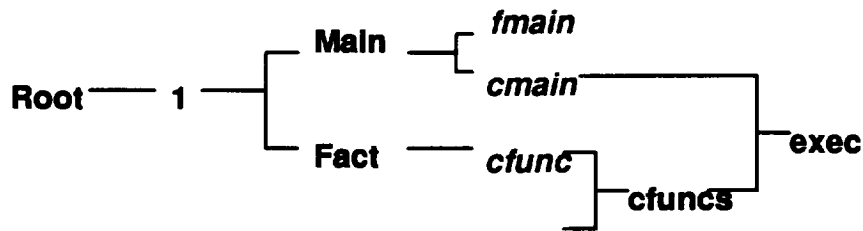
If the packager does not find a manufacture graph, then it is *not possible* to integrate the application in the target environment. There is no free lunch or magic! The tools to integrate a system must exist. The packager only hides their use from developers.

The rules express the integration processes available in an environment in terms of *abstract software manufacture graphs*. A set of rules for an environment describe the abstract software manufacture graphs and look something like an attribute grammar. A specific software manufacture graph, called a concrete software manufacture graph, corresponds to the integration tree if it exists. For example, the manufacture graph for our second example (with an alternative implementation of the Main module in FORTRAN):



is found by the packager given the rules in Appendix B. Again, these rules are shared by all users in an environment and maintained by the system administrator. Normal users should never deal with or have to write such rules. The leaf nodes of the concrete manufacture graph above correspond to the leaf nodes of the software structure graph. Another view of the packager process would be to take the software structure graph and the manufacture graph and connect the two structures together like this:

²The type of target object that the packager builds is not limited to executables. It can be used to build documents (e.g., dvi files), libraries, and other composite software objects.



This view, called a *software production graph*, presents a clear view that the `fmain` implementation of the `Main` module is not chosen but all primitive implementations are accounted for in the manufacture graph. Once the manufacture graph is determined, then the packager can execute actions associated with each node starting from the root (of the manufacture graph, not the software structure graph). The actions are contained in the `:{ ... }` clause after each rule. Actions are comprised of commands (one per line). In the rules in Appendix B, the first command,

```

exec      :      cmain cfuncs
          :;
          %all: $1.APPNAME
          ...

```

will produce the output

```
all: a.out
```

where the ```%``` symbol directs the subsequent line to the current output file. By default, the current output file is the `MAKEFILE`. The characters `"all: "` are echoed directly into the output. The clause `$1.APPNAME` directs the packager to insert the value of the `APPNAME` attribute of the first object on the right-hand side of the associated rule. In this case, `$1` refers to the `cmain` object and the value of its `APPNAME` attribute is the string `a.out`. This attribute is declared in the definition of the `Root` module in the `stdpkg` file.

Commands on subsequent lines in the action associated with the rule work in the same fashion. The command

```
%$1.APPNAME: $1.FILE:r.o $2(OBJS)
```

illustrates two interesting features. First, in the clause `$1.FILE:r.o`, the `:r` represents an operation that specifies that a single file extension be stripped from the value of the `$1.FILE` attribute value. In this case, the value of `$1.FILE` (e.g., the `FILE` attribute of the `cmain` object) is `main.c`. Thus, the value of `$1.FILE:r` is the string `"main"` since the extension `".c"` is stripped off. Then, the extension `".o"` is just appended to this to form the output string `"main.o"` for insertion in the output line.

The subsequent `$2(OBJS)` clause does not access an attribute of a primitive implementation object. By analogy with attribute grammars, the `OBJS` value is a synthesized attribute of the subtree rooted at the `cfuncs` branch of the node associated with the `exec : cmain cfuncs` rule. In the rule `cfuncs : cfunc cfuncs`, the clause

```
{ (OBJS) $1.FILE:r ".o" $2(OBJS) }
```

declares the synthesized attribute `OBJS` associated with each node. The rule for `cfuncs` also has an alternative rule whose right-hand side is empty:

```

cfuncs    :      cfunc cfuncs
          :;
          |      ← empty rule
          ;

```

If the first part of the rule fails in a search, the second part of the rule will always succeed. The second part of the rule does not declare any synthesized attributes or actions.

Finally, in the action associated with the rule `cexec : cmain cfuncs`, the command `$2` on the last line of the action:

```
exec      :      cmain cfuncs
          :{
          ...
          $2
          }
          ;
```

directs the packager to execute the action of the node associated with the second object on the right-hand side of the rule. Thus, the `$2` clause executes the action associated with the `cfuncs : cfunc cfuncs` rule. Table 1 shows all commands possible in rule actions and their semantics.

The resulting MAKEFILE is shown in Appendix A. The rules capture the abstract form of all MAKEFILES that integrate C programs. Through the use of both inherited and synthesized attributes, the output MAKEFILE is customized.

7 References

- [1] J. Callahan and J. Purtilo, A packaging system for heterogeneous execution environments, *IEEE Transactions on Software Engineering*, June 1991, Volume 17, Number 6, pp. 626-635.
- [2] Feldman, S., Make: A Program for Maintaining Computer Programs, *Software Practice and Experience*, April 1979, Volume 9, Number 4, pp. 255-265.
- [3] McNutt, D., Imake: Friend or Foe?, *SunExpert*, November 1991, Volume 2, Number 11, pp. 46-50.

Appendix A: A simple MAKEFILE

```
all: a.out

a.out: main.o fact.o
    cc -o a.out main.o fact.o

main.o: main.c
    cc -c main.c

clean::
    rm -f main.o *~ core a.out *.bak

clobber::
    rm -f main.o a.out *~ core a.out *.bak

fact.o: fact.c
    cc -c fact.c

clean::
    rm -f fact.o

clobber::
    rm -f fact.o
```

Appendix B: A simple rules file

```

exec      :      cmain cfuncs
          :{
          %all: $1.APPNAME
          *
          %$1.APPNAME: $1.FILE:r.o $2(OBJS)
          *      cc -o $1.APPNAME $1.FILE:r.o $2(OBJS)
          *
          %$1.FILE:r.o: $1.FILE
          *      cc -c $1.FILE
          *
          %clean::
          *      rm -f $1.FILE:r.o *~ core a.out *.bak
          *
          %clobber::
          *      rm -f $1.FILE:r.o $1.APPNAME *~ core a.out *.bak
          *
          $2
          }
          ;

cfuncs   :      cfunc cfuncs
          [ (OBJS) $1.FILE:r ".o" $2(OBJS) ]
          :{
          %$1.FILE:r.o: $1.FILE
          *      cc -c $1.FILE
          *
          %clean::
          *      rm -f $1.FILE:r.o
          *
          %clobber::
          *      rm -f $1.FILE:r.o
          $2
          :
          ;

```