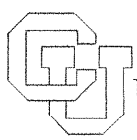
ODIN – An Extensible Software Environment Report And User's Reference Manual *

Geoffrey M. Clemm

CU-CS-262-84



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* I acknowledge Department of Energy support under contract no. DE-AC02-80ER10718 and National Science Foundation support under grant no. MCS80-00017.

ODIN - An Extensible Software Environment
Report and User's Reference Manual

by

Geoffrey M. Clemm
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-262-84 March 1984

(Revised December 1984)

I acknowledge Dept. of Energy support
under contract no. DE-AC02-80ER10718
and National Science Foundation support
under grant no. MCS80-00017

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION.

Abstract

The purpose of the Odin System is to provide an extensible program development environment. In Odin, a standard hierarchical file system is extended by the addition of user specified file types and user specified operations defined in terms of those file types. User operations are implemented through sequences of host system commands. Two languages are provided : an object oriented command language, and a specification language to allow the easy addition of new file types and operations. The user maintains in the host file system only those files that will be modified directly - the results of all operations are maintained within the Odin System.

CONTENTS

1. INTRODUCTION

The premise of the Odin System is that the basic software objects are files, and that the task of the programmer is to modify old files or to produce new files. The way an environment supports this task is by providing to the user a variety of "views" of files, where a view of a file is a mechanical, although possibly extremely complex, derivation from that file. For our purposes, a mechanical derivation is defined to be one that can be performed by a computer program - each of the output files of the program form a view of the input to the program.

The production of views is essential for three reasons. First, the form of the file most convenient for its specification may not be the form that it must take for eventual use (program source text vs. object code). Secondly, the task of producing a given new file can be an extremely difficult one, which often can be effectively solved only if a variety of different views (abstractions) of the existing files are available (call graphs, cross reference listings). Finally, views are the mechanism whereby properties of files can be specified, so that the environment can verify that these properties are maintained when a file is modified (error free compilation, correct spelling of documents).

In most programming environments, the user is largely restricted to the views produced by the designer of the programming environment. Even in the exceptional cases where the environment is designed to be extensible, performing this extension involves a degree of knowledge about the system possessed by only a few expert users. Alternative approaches have been to provide a massive set of tools in an attempt to cover as many views as possible, or to deliberately limit the environment to a specialized application area. While these approaches can be very

successful when the tools provided are well designed and integrated, many application areas involve constantly changing and evolving needs which cannot be met by a fixed environment. In addition, the advantages of new and improved tools must be foregone due to the difficulty or impossibility of integrating these tools into the existing environment.

The purpose of the Odin System is to provide an extensible program development environment. A specification language is provided to describe the behavior of new tools and the Odin Command Language offers a simple and consistent mechanism for requesting the results of these new tools.

2. THE COMMAND LANGUAGE

The Odin Command Language is an object-oriented command language. Tools are invoked only as needed to create a requested Odin object. For example, if an executable object were requested, various compilers and loaders might be invoked. The tools "might be" invoked because the Odin System automatically saves the objects from previous requests, so that a given object might already exist and therefore be immediately available.

2.1. Odin Objects

All Odin objects can be thought of as files (simple objects) or sets of files (compound objects). Examples of simple objects would be source code, executable binary, or output from a test run. Examples of compound objects would be the set of files containing the source code of a single program, the set of files containing different versions of the same source code, or an executable program with files containing input data for the program.

2.1.1. Primitive Odin Objects

The primitive Odin objects are host system files. In case the host system provides a hierarchical file system, a file name that does not begin with a slash ('/') is assumed to be relative to the "current" directory (e.g. the directory from which the Odin System was invoked). For example, if the current directory were "/usr/geoff/src", then both "/usr/geoff/src/test.c" and "test.c" would refer to the same file.

Every primitive object is given a type by the Odin system based on the file name "extension" of the host file. The extension of a file name is the string following the last period in the final segment of the file name, where segments are separated by a slashes. For example, the extension of each of the following file names is "c" :

```
/usr/geoff/src/test.c
src/test.c
test.c
test.1.c
```

The type of a primitive object determines which derived files can be produced from that file. In case the extension of a primitive object is null or is not recognized by the Odin system, no derived files can be produced from that file.

If a list of the possible extensions is desired, typing a question mark followed by a colon ("? :") to the Odin System would generate a message of the form :

```
Possible Base Types :
c ..........  C source code
vc .........  C code stored in rcs format
f ..........  Fortran77 source code
vf .........  Fortran77 code stored in rcs format
int ........  an integer
tgi ........  tree-building parser grammar
fsi ........  scanner grammar
tgiref .....  tree-building parser/scanner grammars
mf .........  Fortran77 source code with m4 constructs
h ..........  include data
i ..........  m4 Include Data
ref ........  reference file containing a list of file names
```

This states that all of the following would have extensions that the Odin system would recognize :

```
/tmp/test.c
project.vc
src/parser/new.tgiref
```

2.1.2. Derived Odin Objects

A derived Odin object is a file (or set of files) that can be produced from a primitive object (or another derived object) through the invocation of one or more tools. Examples of objects that can be derived from source code would be cross reference listings, executable binary code, or a formatted version.

There are three Odin operations for specifying derived objects : derivation, parameterization, and selection.

2.1.2.1. Derivation

A derivation is specified by appending to the name of an Odin object a colon (':') and the name of the desired derivation. For example,

test.c :fmt

would request a formatted version of test.c, and

test.c :fmt :run

would request the result of compiling and executing the formatted version of test.c.

If the name of the desired derivation has been forgotten or a list of the possible derivations is desired, a question mark ('?') can be put in place of the derivation name, and the Odin System will respond with a list of the possible derivation names that could appear at that position. For example,

test.c :fmt : ?

would generate the following message :

Possible Derivations from an Object of Type "fmt" :

obj ..... object code from c compiler
fmt ..... formatted version
xref ..... cross reference listing
run ..... results of executing a c program

This states that all of the following would be legal objects :

test.c :fmt :obj
test.c :fmt :fmt
test.c :fmt :xref
test.c :fmt :run

2.1.2.2. Parameterization

It frequently occurs that there is a variety of additional information that can be associated with a file and that will affect the derivatives produced from that file. In the Odin System, this additional information is associated with a file as the "parameters" of that file. For example : a

debug parameter could cause the compile derivative to contain run-time checks; a library parameter could cause the load derivative to have undefined externals satisfied from a non-default library; and a format parameter could cause all printable derivatives to be generated in line-printer format.

A parameterized object is specified by appending to the specification of an object a '+' and a parameter. For example, a debug parameter can be added to the object "test.c :fmt" as follows :

```
test.c :fmt +debug
```

If this new object is then run, e.g.

```
test.c :fmt +debug :run
```

the "run" object produced would contain debugging information.

It is often the case that a value should be associated with a given parameter. Such a value can be specified by appending to the parameter an equal-sign ('=') and the value. For example, if array bound violations are to be checked or if dereferencing of nil pointers are to be checked for the object "test.c", then respectively

```
test.c +debug=arrays
```

or

```
test.c +debug=nilref
```

would be specified.

If the value associated with a parameter is contained in another Odin object, the value is specified as the Odin object surrounded by parentheses. For example, suppose that there is a derivation named "lib" that will produce a library from a file of source code. Then the result of running "test.c" using the library produced from a file called "util.c"

would be specified as :

```
test.c +lib=(util.c :lib) :run
```

If the name of the desired parameter has been forgotten or a list of the possible parameters is desired, a question mark ('?') can be put in place of the parameter, and the Odin System will respond with a list of the possible parameters that could appear at that position. For example,

```
test.c :fmt + ?
```

would generate the following message :

```
Possible Paramters :      id lib debug
```

This states that all of the following would be legal objects :

```
test.c :fmt +id
test.c :fmt +lib
test.c :fmt +debug
```

In fact, both id and lib should be associated with parameter values, such as :

```
test.c :fmt +id=run5
test.c :fmt +lib=(/usr/lib/network.a)
```

but since this required value information is not stored in the derivation graph, an unexpected parameter value (or lack of a value) will only be detected by the appropriate tool after the erroneous object has been requested.

A more exact form of parameter help can be requested by specifying which derivation you are intending to apply to the parameterized object. For example,

```
test.c :fmt + ? :obj
```

would generate the following message :

```
Possible Paramters :      debug
```

This states that the following would be a legal object :

test.c :fmt +debug :obj

Since the id and lib parameters are not relevant to the derivation from fmt to obj, these are not listed.

2.1.2.3. Selection

As mentioned earlier, an Odin object can be either a simple object (i.e. a single file) or a compound object (i.e. a set of files). Frequently, it would be desirable to specify some subset of the objects in a compound object. To allow this, Odin associates a "key" with every Odin object.

A primitive object is given a key corresponding to its host file name, e.g. the key of "/usr/geoff/src/test.c" would be "test.c". A derived object is given a default key equal to the key of the primitive object from which it was derived. For example, the key of "src/test.c :run" would be "test.c".

In case a derived object is a compound object, the key for each element of the compound object is generated by the tool that produces the compound object. For example, suppose "src/test.c :output" specifies the output files generated when running "src/test.c". This derived file is a compound object because a program can generate more than one output file. Since the tool that executes a user's program is responsible for giving keys to the output files, they could be arbitrarily given the keys "out1", "out2", etc. A more useful and more likely convention would be for the tool to use the file names given by the user's program to the output files as the keys for the output files.

The subset, from a given compound object, of objects with a certain key can be specified by appending to the name of the compound object

an at-sign ('@') and a key. For example, suppose that running "src/test.c" produces three output files named "DATA", "source.list", and "source.errors". These three files could be specified as the three Odin objects,

```
src/test.c :output @DATA
src/test.c :output @source.list
src/test.c :output @source.errors
```

2.1.3. Status Level of Odin Objects

Associated with each Odin object is a status level, where a status level is one of OK, WARNING, ERROR, NOREAD, NOFILE, and ABORT. OK is considered the maximum status level and ABORT the minimum. The status of a primitive object is always OK. The status of a given derived object depends on the results of the tools needed to produce that object. If any tool generated warning messages, the status level of the given object is at most WARNING. If any tool generated error messages, the status level of the given object is at most ERROR. If any object that was needed to generate the given object was not readable, the status level of the given object is at most NOREAD. If any object that was needed to generate the given object did not exist, the status level of the given object is at most NOFILE. If any object that was needed to generate the given object had status level ERROR, then the status level of the given object is set to be ABORT.

If the status level of an object is less than OK, the status level is indicated whenever that object is displayed or copied. The actual warning or error messages that were produced can be displayed by requesting the results of running the internal WARNING tool or the internal ERROR tool (see "Tools"). Assume that the ":warn" and ":err" derivations invoke the WARNING tool and the ERROR tool respectively. If the request to display

the object,

test.c :run

resulted in the message,

Note : Abort Status set for "test.c :run"

the errors that caused the generation of the abort status would be listed in the object,

test.c :run :err

Error messages are included in the list of warning messages, so the list of errors is always a subset of the list of warnings. The difference between an error and a warning is that an error prevents the tool from generating its output, while a warning indicates that although output was generated, it might be faulty. An example of an error message from a loader would be

Unsatisfied external reference : "proc1".

An example of a warning message from a loader would be

Multiply defined external : "proc2", first copy loaded.

2.1.4. Sentinels

In any software system, it is very useful to be able to specify semantic constraints on the software objects in the system. In Odin, this capability is provided through distinguished Odin objects called "sentinels". Examples of sentinels would be

```
thesis.txt :spell
prog.c +input=(thesis.txt) :run
```

If sentinels are activated, whenever a modification to an atomic file causes the status level of any sentinel to become ERROR or less, Odin will

generate an error message indicating which sentinels have been violated. In the above examples, assume that the ":spell" object receives ERROR status if any spelling errors are detected, and that the ":run" object receives ERROR status if any error messages are generated in the attempt to compile and run "prog.c" with input file "thesis.txt". Then if "thesis.txt" is modified, Odin will check that the "thesis.txt" file is spelled correctly, and that it is acceptable to the "prog.c" program. In addition, assuming "prog.c" uses the system library "/usr/lib/jobs", if either "prog.c" or "/usr/lib/jobs" is modified, Odin will check that prog.c still runs successfully with "thesis.txt" as input.

The list of sentinel objects is stored in the special Odin object specified as a vertical bar ('|'). Any user is permitted to add arbitrary Odin objects (and therefore arbitrary semantic constraints) to this special object. A list of all sentinel violations can be obtained by requesting the object,

```
|:compound :err
```

where ":compound" invokes the internal Odin tool, COMPOUND, and ":err" invokes the internal Odin tool, ERROR (see "Tools").

2.2. Odin Commands

There are two basic Odin commands : the display command which is used to view an object and the transfer command which is used to copy or modify an object. A basic commands consist of the name of an Odin object, possibly followed by a description of what is to be done with that object. In addition to the basic commands, there are four utility commands concerning history substitution, command scripts, host system commands, and help. The utility commands are used to invoke previously specified Odin commands or to query various characteristics of the Odin system.

2.2.1. Display Command

The display command prints out an Odin object to the current standard output device, normally a terminal screen. An Odin object is displayed by specifying its name. For example,

```
test.c
```

would display the file named "test.c" in the current directory.

```
test.c +lib=(/usr/lib/simple.a) :run
```

would display the results of running the file "test.c" when loaded with the library "/usr/lib/simple.a".

2.2.2. Transfer Command

The basic form of the transfer command copies the contents of one Odin object into another Odin object. The second object must be a primitive object. An Odin object is copied by appending to the name of the first object a right-angle-bracket ('>') and the name of the second object. For example,

```
test.c > test2.c
```

would put a copy of the contents of "test.c" into "test2.c".

```
test.c :run :err > test.err
```

would put into "test.err" a copy of the list of errors generated in attempting to run "test.c".

In case it is desired that the copy should only take place if it would violate no sentinels, the "guarded copy" form of an Odin command script should be used (see "Sentinels" and "Command Scripts").

An extended form of the transfer command places an object as input to a host system command. This allows the use of host system "editors" or "viewers". In this form of the transfer command, appended to the object is a right-angle-bracket ('>'), a colon (':'), and the name of the host system command. For example,

```
test.c > :vi
```

would invoke the host system editor "vi" on the file "test.c", while

```
test.c :run :err > :more
```

would display the list of errors by running the host system command "more" with the list of errors as its input.

In case the colon and host system command name is omitted, a default host system command is invoked. The name of the default host system command is specified in the Odin "Editor" variable (see "Odin Variables"). For example, if the default host system command is "vi", then the following two commands are equivalent :

```
prog.ref >
prog.ref > :vi
```

2.2.3. History Substitution

A list of all basic commands invoked during a given Odin session is maintained by the Odin system. This list can be displayed and modified, and commands from this list can be selected and modified for re-execution.

The exclamation point character ('!') is to display the history list. Since the history list can grow quite large, only a given number of the most recent commands are actually displayed. This number is specified by the History variable (see "Odin Variables").

The history list can be modified by specifying the history list as the object in a transfer command. For example,

```
! > :vi
```

would invoke the host system editor "vi" on the history list.

A given command from the history list can be selected for execution by following the exclamation point with an integer or a word. An integer, i, selects the i'th most recent command in the history list. For example,

```
!1
```

would select the last basic command for re-execution. A word selects the most recent command that contains that word, where a word is an alphanumeric string. For example,

```
!run
```

could be used to select the command

```
test.c :run :err
```

The selected command can be modified before being re-executed by specifying the selection in the form of a transfer command. For example,

```
!run > :vi
```

specifies that the "run" command should be given to the host system editor, "vi", before being re-executed. In case no change has been made by the host system editor, the command will not be re-executed.

2.2.4. Command Scripts

An Odin command script consists of an Odin object that contains a list of Odin commands. This command script can be invoked by specifying a left-angle-bracket ('<') and the name of the Odin object. For example, if "script.odin" contained a list of Odin commands,

```
< script.odin
```

would invoke all the commands in script.odin.

2.2.4.1. Guarded Copy In case it is desired that a sequence of copy commands be performed only if no sentinels would be violated by the results of performing these commands, the "guarded copy" form of command script invocation should be used. In this form, the left-angle-bracket is immediately followed by a vertical bar ('|'). All commands in a script invoked from a "guarded copy" must be simple transfer commands (see "Transfer Commands"). For example, if the file "test.copy" contained the following text :

```
/usr/tmp/test.c > /sys/rn.c
/usr/tmp/fix_lib.a > /usr/lib/jobs.a
```

then the command :

```
<| test.copy
```

would cause Odin to check the effect of these two transfer commands on all the sentinel affected by them, with the changes only being made if no sentinels are violated. If simply

```
< test.copy
```

were used, then the changes would be made whether or not any sentinels were violated in the process.

2.2.5. Host System Commands

Host system commands can be invoked by specifying a percent sign ('%') and a host system command. If the host system command contains Odin metacharacters (such as '<', '>', and ':'), then the command must be enclosed in quotes ('"'). For example,

```
% ls
% 'ls *.c > dir.list'
```

would invoke the "ls" and "ls *.c > dir.list" host system commands.

2.2.6. Help

A simple help facility is provided to describe the syntax of Odin commands and Odin objects. A list of topics is generated in response to a single question-mark ('?'). Following is a complete list of the current topics, and the results of requesting each topic.

```
?
Topics :
   Syntax
   Help
   Quit
   Display
   Copy
   Edit
   Script
   HostCommand
   Variables
   OdinFileName
   HostFileName
   Operation
   Parameter
   ParameterKey
   ParameterValue
   FileType
   BaseType
```

```
? syntax
   [a] is optionally a
   [a]... is zero or more a's

? help
   ?
   ? Topic
   OdinFileName : ?
   OdinFileName + ?
   OdinFileName + ? : FileType

? quit
   Control-D

? display
   OdinFileName

? copy
   OdinFileName > OdinFileName

? edit
   OdinFileName Editor

? editor
   >
   > : HOST_TOOL

? history
   !
   ! Editor
   ! HISTORY_ENTRY
   ! HISTORY_ENTRY Editor

? script
   < OdinFileName
   <| OdinFileName

? hostcommand
   % [NAME]...
   % 'STRING'

? variables
   ? =
   VARIABLE = ?
   VARIABLE =
   VARIABLE = VALUE

? odinfilename
   HostFileName [Operation]...
   | [Operation]...

? hostfilename
   FILENAME.BaseType
```

```
? operation
   + Parameter
   : FileType
   @ KEY

? parameter
   ParameterKey [= ParameterValue]

? parameterkey
   Use a help command of the form
     OdinFileName + ?
   or
     OdinFileName + ? : FileType
   to determine appropriate ParameterKeys

? parametervalue
   NAME
   'STRING'
   ( OdinFileName )

? filetype
   Use a help command of the form
     OdinFileName : ?
   to determine appropriate FileTypes

? basetype
   Use a help command of the form
     ? :
   to determine appropriate BaseTypes
```

2.3. Odin Variables

Odin provides to the user a set of variables. These consist of read-only variables and user-modifiable variables. A read-only variable provides to the user status information about the Odin system; a user modifiable variable allows the user to affect the operation of Odin in various ways. Currently the functions affected by changing the values of user modifiable variables are the working directory, the default editor, the help facility, the history facility, the log facility, and the maximum total file space used by derived objects.

Dir

In commands, file names that do not begin with a slash ('/') refer to files with respect to the current working directory. Initially this directory is the one from which Odin was invoked by the user. This directory can be changed by modifying the value of the Dir variable.

ErrFile

All error messages are sent to a file which is initially set to be the standard output device. These messages can be redirected by modifying the ErrFile variable.

Editor

The Editor variable specifies the name of the default host system tool to be used for the abbreviated form of the transfer command.

HelpLevel

The HelpLevel variable specifies what degree of detail should be provided when the user asks for a list of possible derived file types (with the "file :?" help command). Normally, only commonly used file types are

described, but the HelpLevel can specify that all possible file types should be described.

History

The History variable specifies how many of the recent commands should be listed when the history list is displayed (see "History Substitution").

LogFile, LogLevel

The "log" contains a brief description of each of the tools that were invoked to satisfy the request for an object. In addition, whenever a derived file is deleted by Odin to conserve disk space, a message describing the file deleted is sent to the log file. Since objects are saved by the Odin System between requests, the tool executions needed to satisfy a given request will vary. In particular, if an object is requested immediately after a request for that same object, no tools will be invoked. The LogFile variable specifies where the log information should be placed and the LogLevel variable specifies how detailed the generated log information should be.

MaxSize, MinSize, Size

Since the Odin system has the capability of deleting and recreating derived objects at will, parameters of interest are how much total space the derived files should be allowed to occupy and how much space is currently being occupied. After an Odin command is completed, derived objects will be deleted if necessary until Size is less than the MaxSize. The MinSize variable is provided to allow Odin scripts to specify that MaxSize should be at least a specified amount, without affecting MaxSize if it is already larger than that amount.

Sentinel

The Sentinel variable is a boolean that can be on or off. Normally any modification to a file during an Odin session will cause a broadcast of the change to all affected derived files. Any objects specified as sentinels will automatically be updated to reflect the modification. If the Sentinel variable is turned off, no broadcasts will take place, and no sentinels will be updated until such an update is explicitly requested.

Verify

The Verify variable is a boolean that can be on or off. Normally Odin assumes that any modification to a host system file will take place through an Odin transfer command, and that the modification will be broadcast to all derived files that this would affect. In case host system files were modified other than through an Odin transfer command, or were modified while the Sentinel variable was turned off, the Verify variable should be turned on to indicate that actual host system date stamps should be inspected to determine if host system files have been modified.

2.3.1. Variable Manipulation Commands

Four commands are provided to manipulate these variables :

2.3.1.1. Show Variables

A list of the available variable names is generated in response to the command,

? =

Currently, this command would generate the list,

Dir Editor ErrFile HelpLevel History LogFile LogLevel

MaxSize MinSize Sentinel Size Verify

Of these variables, only Size is read-only.

2.3.1.2. Describe Variable

A description of the possible values that can be assigned to a given variable is generated in response to the command,

Variable = ?

The descriptions of the current set of variables are as follows :

Dir = ?
The current working directory.

Editor = ?
The default editor.

ErrFile = ?
1 : Error information sent to standard output.
2 : Error information sent to standard error.
filename : Error information sent to file named "filename".

HelpLevel = ?
1 : Help returns information for common file types.
2 : Help returns information for all file types.

History = ?
The number of lines displayed from the history file.

LogFile = ?
1 : Log information sent to standard output.
2 : Log information sent to standard error.
filename : Log information sent to file named "filename".

LogLevel = ?
1 : No log information is generated.
2 : Insert commands executed by scripts into the log.
3 : And names of objects generated by external tools.
4 : And names of objects generated by internal tools.
5 : And names of objects deleted.
6 : And names of objects touched by broadcast.

MaxSize = ?
The maximum disk space (kilobytes) to be used by derived objects.

MinSize = ?
A minimum value for MaxSize (will not decrease MaxSize).

Size = ?
  The current amount of disk space (kilobytes) used by derived objects.

Sentinel = ?
  off : Sentinel validation off.
  on : File modification validated by Sentinels.

Verify = ?
  off : Assume all host system files modified through Odin.
  on : Check all host system files for external modification.

2.3.1.3. Show Variable Value

The value of a variable is generated in response to the command,

Variable =

where "Variable" is a legal variable name. The default values of the current variables are :

Dir =
  the directory from which Odin was invoked

Editor =
  vi

ErrFile =
  1

HelpLevel =
  1

History =
  5

LogFile =
  1

LogLevel =
  3

MaxSize =
  5000

MinSize =
  5000

Size =
  0

```
Sentinel =
  on

Verify =
  off
```

2.3.1.4. Set Variable

A variable is given a new value with a command of the form,

```
Variable = Value
```

For example,

```
Dir = ../src

Editor = emacs

ErrFile = err.out

HelpLevel = 2

History = 10

LogLevel = 5

LogFile = test.log

MaxSize = 1000

MinSize = 7000

Sentinel = off

Verify = on
```

An attempt to set the value of a read-only variable will generate an error message.

3. SPECIFICATION LANGUAGE

The specification language is designed to allow the integration of any existing tool or set of tools into the Odin System, with no modification to the tools themselves. This is critical when a tool only exists in the form of executable binary, as is often the case for host system provided tools. The only tools provided by the Odin System itself are ones whose purpose is to support this task of integration.

For example, a compiler would be provided in an Odin environment by describing the host system compiler in the Odin specification language. On the other hand, Odin itself provides a tool that will interpret a file containing a list of file names as a "collection of files", so that this collection of files can be treated as a single object by a user of Odin. Odin would ensure that a request to run a tool on this collection would in fact invoke the tool on each of the elements in the collection.

The specification of each tool is entered into a text file called a "derivation graph". Basically, a specification consists of a description of the input and output behavior of the tool, and the host system command file or Odin system tool that performs the desired operations.

For example, a simple formatter could be described as follows :

```
fmt "formatted version of C code" :
        USER pol_c.cmd
                : c
```

where fmt is the name of the results of applying a C code formatter, the string in quotes on the first line describes this object, the name following the keyword USER on the second line specifies the host system command file that will invoke the formatter, and c names the kind of file which is suitable as input to the formatter.

In general, the i/o behavior of a tool can be far more complex than this simple example, but this basic model of the naming the output of a tool, naming the procedure that invokes the tool, and then describing the input to the tool, will always be followed.

3.1. Atomic File Types

Every type of file that is to be edited directly by the user is given a unique "atomic file type". Each atomic file type is declared in the derivation graph by specifying the name of the atomic file type followed by the keyword ATOMIC and a string that provides a short English description of that type of file. For example, atomic file types for C and Fortran source code could be declared as follows :

```
c ATOMIC "C source code"
f ATOMIC "Fortran77 source code"
```

The English description is given with the name of the atomic file type when a user requests at run time a description of what atomic file types are currently known by the system (i.e. "? basetypes").

3.2. Derived File Types

Every type of file that is produced by some computer program or tool is given a unique "derived file type". Each derived file type must be described in the derivation graph. A description of a derived file type consists of a description of the structure of the derived file followed by a description of the tool that produces the derived file and a description of the inputs needed by the tool. For example, in the following rather complex derived file type description :

```
dbx <
  exe-dbx ^null "executables for a dbx run"*
  srcs-dbx (null) "sources for a dbx run"*
  keys-dbx (null) "names of source files for a dbx run"*
  core-dbx "core dump for a dbx run"*
  > "Berkeley symbolic debugger run" :
       USER dbx.cmd
              : exe
              : (objsrcU)
              : (objkeyU)
              : PARAMETERS(id)
```

the description of the the structure of the derived file is :

```
dbx <
  exe-dbx ^null "executable for a dbx run"*
  srcs-dbx (null) "sources for a dbx run"
  keys-dbx (null) "names of source files for a dbx run"*
  core-dbx "core dump for a dbx run"*
  > "Berkeley symbolic debugger run" :
```

the description of the tool is :

```
       USER dbx.cmd
```

and the description of the input is :

```
              : exe
              : (objsrcU)
              : (objkeyU)
              : PARAMETERS(id)
```

As with atomic file types, derived file types are associated with a string that provides a short English description of that type of file. This

English description is given with the name of the derived file type when a user requests at run time a list of what file types can be derived from a given object, based on the file type of that object.

This description can be marked with an asterisk indicating that it describes an "intermediate derived file type". The Odin variable, HelpLevel, specifies whether intermediate derived file types will be included in help messages. The default is to not report intermediate derived file types. In the example above, exe-dbx, keys-dbx, and core-dbx were declared as intermediate derived file types.

3.2.1. Derived File Structure

Due to the great variety in output behavior of tools, it is necessary to provide a flexible language for describing the various possible kinds of derived file types. Examples of different kinds of outputs that a tool might generate would be a single data file, a single file that refers to another file, a fixed number of different kinds of output files, or an arbitrary number of similar output files. The description of the structure of a derived file is always terminated by a colon.

3.2.1.1. Simple Derived File

A file with a "simple" derived file type is just an ordinary text or data file. Some common simple file types would be assembler code generated from a higher level language, executable binary, cross reference listings, and error reports. A simple file type is analogous to a basic variable type in a programming language, such as boolean, character, or integer. Odin allows a user to introduce an arbitrary number of such basic types.

A simple derived file type specification consists of the name of the derived file type followed by a text string describing the type and a colon.

For example, in :

exe "executable binary" :

"exe" is declared to be a simple derived file type.

3.2.1.2. Reference Derived File

A file with a "reference" derived file type is a file that refers to another file. This is analogous to a pointer type in a programming language. Whenever such a file is used, such as when it is displayed or when it is given as input to a tool, it is automatically dereferenced by Odin so that what is displayed or received as input is actually the file referred to. There are two kinds of reference derived file types - pointer reference and name reference.

Pointer Reference Derived File

A file with a "pointer reference" derived file type contains the actual name of the file being referred to. For atomic files, this is just the host system file name; for derived files this is the name of the file in which Odin chose to place the information for that derived file, such as "/usr/odin/ODIN/FILES/c/157823".

A pointer reference derived file type specification is like a simple derived file type specification except that immediately following the name of the file type is added a carat ('^') and the file type of the file being referred to. For example, in :

tgi_ptr ^ tgi "parser grammar" :

"tgi_ptr" is declared as being a pointer to a file of type "tgi".

Name Reference Derived File

A file with a "name reference" derived file type contains an Odin command specification of a file. For atomic files, this will be just the host system file name, the same as in a pointer reference file types; for derived files this will be a specification of the derived file, such as "a.f :fmt" or "test.c +lib=(/usr/lib/network.a) :run".

A name reference derived file type specification is like a pointer reference derived file type specification except that immediately following the name of the referred to file type is added an at-sign ('@'). For example, in :

f_main ~ fcast@ "scanner default main program" :

"f_main" is declared as containing the name of a file of type "fcast".

3.2.1.3. Compound Derived File

A file with a given "compound" derived file type consists of a set of files, each of which has the same file type called the "element file type" or is another compound derived file of the given type. A compound file that contains only files of the element file type is called a "flat compound file" - one that also contains other compound files is called a "nested compound file". A flat compound file is analogous to an array in a programming language - a nested compound file is analogous to a tree. There are two kinds of compound derived file types - compound reference type and compound source type.

Compound Reference Derived File

A file with a "compound reference" derived file type consists of a list of references to other files. These references can be either by pointer or by name, as with reference derived file types.

A compound reference derived file type specification is like a simple derived file type specification except that immediately following the name of the file type is added the name of the element file type in parentheses. For example, in :

objC (obj) "list of object modules" :

"objC" is declared as containing pointers to elements of type "obj".

If the reference is by name, an at-sign ('@') is appended to the element file type name. For example, in :

so_ref (null@) "list of nroff included files" :

"so_ref" is declared as containing the names of elements of type "null".

Compound Source Derived File

A file with a "compound source" derived file type consists of a set of files, all of which were generated by the tool. This is distinguished from compound reference files where only references to existing files are generated by the tool.

A compound source derived file type specification is like a compound reference derived file type specification except that square brackets ('[' ']') are used instead of parentheses. For example, in :

output [data] "output files from a test run" :

"output" is declared as being a set of files of type "data".

3.2.1.4. Composite Derived File

A file with a "composite" derived file type consists of a set of a fixed number of files, each of which has a specific, although possibly different, file type. This is analogous to a record or structure type in a programming language. In Odin, most tools that are normally considered

to produce multiple outputs are instead considered to be tools that produces a single composite file as output. The members of a composite file type can be compound, reference, or simple file types.

A composite derived file type specification is like a simple derived file type specification except that immediately following the name of the file type is added a pair of angle brackets ('<' '>') containing a list of member file type specifications. Each member file type specification is either a compound, a reference, or a simple file type specification, except that the terminating colon is omitted. For example, in :

```
fscan <
  fst "scanner tables"*
  fst_lst "fscan compiler listing"*
  f_drive ^fcast@ "scanner driver routines"*
  f_main ^fcast@ "scanner default main program"*
  > "scanner tables"* :
```

"fscan" is declared as being a structure containing four elements - a simple type "fst", a simple type "fst_lst", a name reference type "f_drive", and a name reference type "f_main". The tool that produces "fscan" would be responsible for generating an "fst", an "fst_lst", an "f_drive", and an "f_main" output file - the Odin system would then be responsible for producing the fscan composite file from these four members.

3.2.2. Inputs

In order to produce a file of a given type, one or more input files are needed by the tool that creates this file. These input files are specified as a list of file types, each preceded by a colon. These file types can be atomic file types, derived file types, or parameter file types. For example,

```
f-scan (f) "source files for a scanner module"* :
        COLLECT
                : fst
```

```
        : f_drive
```

specifies that the file types "fst" and "f_drive" are needed as input.

In addition, it is sometimes convenient to have a constant file as an input file, where this constant file contains data needed by the tool. In this case the name of the constant file is placed in quotes, again preceded by a colon. In the above example, if "f_drive" is the same for all tool invocations, the specification could be modified to read :

```
f-scan (f) "source files for a scanner module"* :
        COLLECT
                : fst
                : "/usr/lib/std.f_drive"
```

3.2.2.1. Parameter File Types

Normally, when a derived file is being produced, the actual inputs to a tool are determined automatically by Odin based on the object from which the file is derived. It sometimes is the case that a user would like to pass additional information to certain of the tools. This can be done when a derived file is requested at run time by appending to the description of the object from which the file is derived, a list of parameters. A parameter consists of a parameter file type followed by the information that is to be placed in the input file corresponding to that parameter file type. If the parameter value is a compound file, then instead of creating an input file for that parameter, an input directory is created, and each element of the compound file is linked into that directory with a name matching its Odin key. Normally a tool will allow a parameter file to be omitted, in which case a default value will be assumed.

The parameter file types used as input to the tool producing a given file type are described in the derivation graph by specifying the keyword

PARAMETERS followed by a list of parameter names separated by commas. For example,

: PARAMETERS (debug, lib)

would indicate that the "debug" and "lib" parameter files will be used.

3.2.2.2. Transitive Needed File Types

In case one of the needed file types is a compound file, the question arises whether just the list of names of elements of the compound file is needed, or whether the data in those files is needed as well. The default is that only the list of names is needed. If the data in these files is needed, this is specified by placing parentheses around the appropriate needed file type. For example,

: (cmpd)

would indicate that the elements of the "cmpd" input are required, while

: cmpd

would indicate that only the names of the elements of the "cmpd" input are required.

3.2.3. Tools

The tool specifies what process must be executed to produce the specified derived file from the specified inputs. There are two kinds of tools - "internal tools" that are provided by Odin and "external tools" that are provided by the user.

3.2.3.1. Internal Tools

An internal tool is selected in a derived file specification with the

keyword for that internal tool. For example, in the specification

```
ckey "name of c file"* :
        KEY
                : c
```

the internal tool KEY is selected.

Currently there are sixteen internal tools :

STRUCT

The STRUCT internal tool produces a composite file from a text file containing a sequence of odin file specifications, one per line. Each specified file in order is placed as the corresponding member of the composite file. If the number of lines in the text file is not equal to the number of members of the composite file, the STRUCT tool generates an error message.

COMPOUND

The COMPOUND internal tool produces a compound pointer reference file from a compound name reference file.

COLLECT

The COLLECT internal tool produces a single compound reference file from a set of compound reference files by constructing a new compound reference file whose elements are the set of input files.

FLATTEN

The FLATTEN internal tool produces a flat compound file from a nested compound file. This is done by performing a depth first search of the input compound file, and adding a reference to each simple file found, in the order in which it is visited, to the output file.

UNION

The UNION internal tool produces a flat compound file from a nested compound file. This is similar to the FLATTEN internal tool, except that only one copy of each element file is placed in the result - if a file has already been placed into the result file, any later occurrences of that file in the input compound file will be ignored.

HOMOMORPHISM derivation-spec

The HOMOMORPHISM internal tool produces a compound file from another compound file by applying the derivation following the HOMOMORPHISM keyword to each element of the input compound file. A derivation is specified for homomorphisms in the same way that a derived file is specified in the Odin command language, except that the keyword HOMOMORPHISM is treated as the atomic file, and vertical bars ('|') are used in place of colons (':'). For example, if it is desired that the "obj_src" derivation be applied to each element of the input compound file, then the tool would be specified as

HOMOMORPHISM | obj_src

P-HOMOMORPHISM derivation-spec

The P-HOMOMORPHISM (parameterized homomorphism) internal tool is identical to the HOMOMORPHISM internal tool, except that the parameters used to produce the input to the tool are added to the parameters specified for the P-HOMOMORPHISM tool. This is used primarily when a tool is to be applied recursively to its results, in which case it is desirable that the parameters be passed along to the recursive invocations.

APPLY

The APPLY internal tool is similar to the HOMOMORPHISM tool, except that the derivation to be performed is stored in a file rather than specified in the derivation graph. Unlike the HOMOMORPHISM tool which applies one derivation to each of the elements of its input file, the APPLY tool applies each of the derivations in its first input file to its second input file. The APPLY tool provides the ability to generate at runtime the derivations to be performed.

KEY

The KEY internal tool generates a file containing the key of the input file. This is the key that would be used by the Odin selection operator.

CAT

The CAT internal tool produces a simple file from a compound file by concatenating together the contents of all simple files that are elements of the compound file. The order of concatenation is the same depth first order of the FLATTEN and UNION internal tools.

ERROR

The ERROR internal tool produces a simple file from an arbitrary input file. This simple file contains all error messages generated by any tool in the process of creating the input file.

WARNING

The WARNING internal tool produces a simple file from an arbitrary input file. This simple file contains all warning and error messages generated by any tool in the process of creating the input file.

SENTINEL

The SENTINEL internal tool produces a compound file from an arbitrary input file. This compound file will consist of all sentinels that depend on the input file.

NAME

The NAME internal tool produces a simple file from a compound or composite file consisting of the names of all the elements or members of the input file.

COPYCHK

The COPYCHK internal tool produces a composite file from a simple file. The input file must be a sequence of simple transfer commands. The composite file generated consists of two compound reference files - the first a list of the origin files for the transfer commands and the second a list of the destination files for the transfer commands.

COPYTST

The COPYTST internal tool produces a compound reference file from two compound files. The first input file is an arbitrary list of Odin objects. The second input file is a list of two compound files, the first of which is a list of origin files for transfer commands and the second of which is a list of destination files for transfer commands (as produced by the COPYCHK tool). The output of the COPYTST tool is a list of objects corresponding to those in the first input file, where each object is modified to show what the effect would be of performing the specified set of transfers. This tool in conjunction with the COPYCHK tool is used to implement guarded commands (see "Command Scripts").

3.2.3.2. External Tools

An external tool is selected in a derived file specification with the keyword USER followed by the name of a host system command file that implements that tool. This host system command file is written with macro names in place of the various input files it will use and output files it will produce. When it is necessary to generate a given derived file whose tool is an external tool, Odin creates a copy of the command file with macro names replaced with actual file names. This modified command file is then given to the host system to execute.

For example, if the obj_f type was specified in the derivation graph as follows :

```
obj_f <
  obj-f "Fortran77 object module"*
  obj_key-f "Fortran77 source code file name"*
  obj_src-f ^null "Fortran77 source code"*
  > "Fortran77 object module information"* :
       USER obj_f.cmd
              : f
              : fkey
              : PARAMETERS(debug)
```

then the "obj_f.cmd" file for a Berkeley 4.2 Unix machine could be :

```
cd $(RUNDIR)

set source = 'cat $(fkey)'
if ($source:e != 'f') set source = $source.f
ln -s $(f) $source

set flags = ''
if (-e $(PRM)/debug) set flags = '-g'

(f77 $flags -c $source) >&! ERRORS

sed -n '/rror/p' < ERRORS >! $(ERROR)
sed -n '/arning/p' < ERRORS >! $(WARNING)
if (-e $source:r.o) mv $source:r.o $(obj-f)
echo "$source" >! $(obj_key-f)
echo "$(f)" >! $(obj_src-f)

echo 0 >! $(OK)
```

The command file for an external tool is considered to be one of the inputs that affects any object that is produced by that external tool. This implies that any time a command file is modified, all objects produced from that command file will be regenerated.

Macros

Command file macros consist of a dollar sign ('$'), a left parenthesis ('('), a macro name, and a right parenthesis (')'), with no embedded spaces. Examples of macros would be :

```
$(f)
$(ERROR)
$(<2)
```

Input File Name Macros

An input file is specified in a command file with a macro name that is the derived file type name for that input. For parameter inputs, there is a standard directory whose macro name is PRM into which all files for parameter inputs are placed by name. Therefore a parameter input is referenced by $(PRM)/parameter-name. For example, with the preceding specification for obj_f, the macro $(f) would stand for the input file of type f, the macro $(fkey) would stand for the input file of type fkey, and the macro $(PRM)/debug would stand for the input file associated with the debug parameter.

In case the parameter value is a compound file, the elements of the compound file will be linked into a directory that can be referred to as $(PRM.DIR)/parameter-name. The name of an element in this directory will be the same as its Odin key.

An alternative specification of an input file is with a macro name that consists of a left angle bracket ('<') followed by an integer. Assuming k is

an integer, $(<k) would refer to the k'th input in the derived file type specification. In the example above, $(<1) would be equivalent to $(f), and $(<2) would be equivalent to $(fkey). The purpose of this alternate method is to allow one command file to be used for several different but related external tools even when they have different input file types. For example, in the following specifications :

```
inc_ref-i (null@) "list of m4-style included files"* :
        USER inc_m4.cmd
                : i

inc_ref-mf (null@) "list of m4-style included files"* :
        USER inc_m4.cmd
                : mf
```

the process that is run on a file of type i is the same as the one that is run on a file of type mf. In this case, $(<1) would have to be used in the inc_m4.cmd command file to refer to the input file. For example, on a Berkeley 4.2 Unix machine, the following inc_m4.cmd file could be used :

```
cd $(RUNDIR)
($(TOOL)/inc_m4.exe < $(<1) >! $(>1)) >&! $(ERROR)
echo 0 >! $(OK)
```

In addition to the macros for input files, there are three standard macro names, CURDIR, RUNDIR, and TOOL. $(CURDIR) stands for the directory containing the host system file from which the output file is derived. $(RUNDIR) stands for a temporary working directory in which the command file will be executed. $(TOOL) stands for the standard directory in which is placed the executables for external tools that are not provided by the host operating system.

Output File Name Macros

An output file is specified in a command file with a macro name that is the derived file type name for that output. For simple, reference, and compound reference derived file types, there would be just one output file. For compound source derived file types there would be one output directory in which each element of the compound source file will be created. For composite derived file types, there would be one output file for each member of the composite type. For example, with the specification :

```
run <
  stdout "standard output from a test run (when +out is set)"
  output [data] "output files from a test run"
  core-run "core dump of a test run"*
  > "test run" :
       USER run.cmd
            : exe
```

the output from the test run would be placed in the file $(stdout), the files generated by the test run will be placed in the directory $(output), and the core dump if any will be placed in the file $(core-run). An example of a run.cmd file for Berkeley 4.2 Unix would be :

```
cd $(output)

($(exe) >! $stdout) >&! $(WARNING)
if ($status != 0) echo run failed >>! $(ERROR)

if (-e core) mv core $(core-run)

echo 0 >! $(OK)
```

Analogously with input files, an output file can be specified with a macro name that consists of a right angle bracket ('>') followed by an integer. Assuming k is an integer, $(>k) would refer to the k'th output in the derived file type specification. In the example above, $(>1) would be equivalent to $(stdout), $(>2) would be equivalent to $(output), and $(>3)

would be equivalent to $(core-run).

In addition to the macros for output files, there are three standard macro names for error reporting : ERROR, WARNING, and OK. If any fatal errors are encountered, these should be written to the file specified as $(ERROR). If any recoverable errors are encountered, these should be written to the file specified as $(WARNING). Finally, when the script terminates, a line consisting of the character '0' should be written to the file specified as $(OK). The file $(OK) will be used by Odin to determine if the script was able to terminate - if the script itself dies $(OK) will be left empty and Odin will assign abort status to the output of the tool. If the script did not abort, the files $(ERROR) and $(WARNING) will be used by Odin to determine if error or warning status should be set for the output of the tool.

3.3. Linking File Types

A linking file type is declared in the derivation graph by specifying the name of the linking file type followed by the keyword DERIVED and a string that provides a short English description of that type of file. Linking file types are used to specify relationships between other file types in the derivation graph.

It frequently occurs that the input necessary to produce a given derived file type, TypeX, can be provided by two or more different file types, Src1 and Src2. Rather than specify two derived file types, TypeX1 and TypeX2, where TypeX1 can be derived from Src1 and TypeX2 can be derived from Src2, it is more convenient to link the two possible input file types to a new file type, SrcX, and specify that this new file type is the input file type to produce TypeX.

For example, suppose that input to produce an executable binary file type "exe" can be provided by both the file type "obj-c" produced by a C compiler and the file type "obj-f" produced by a Fortran compiler. Rather than specifying two different file types, e.g. "exe-c" and "exe-f", that produce executable binaries from "obj-c" and "obj-f" files respectively, a linking file type "obj" can be specified :

obj DERIVED "relocatable binary"

This "obj" file type is then specified as the input to the tool that produces an "exe" file type. Equivalence links are then specified to indicate that either "obj-c" or "obj-f" can be used as an "obj" file type.

3.3.1. Equivalence Links

A equivalence link is created by specifying the "from" file type followed by an arrow ('=>') followed by the "to" file type. In the preceding

example these links would be added to the derivation graph :

```
obj-c => obj
obj-f => obj
```

3.3.2. Cast Links

It sometimes occurs that a file type that is derived from a given file type can be used in the same way that the given file type could be used. The commonest example of this would be a program formatter. The output from the formatter can be used in all the ways that the original file could be used - it can even be formatted again. This situation is indicated in the derivation graph by specifying a cast link from the derived file type to the given file type. A cast link is specified like an equivalence link except that the head of the arrow is a vertical bar ('=|'). For example, to indicate that formatted c code can be used whenever c code can be used, the following would be specified :

```
fmt-c =| c
```

3.4. Pre-Defined File Types

Four pre-defined file types are provided by the specification language to facilitate the construction of generic tools that accept virtually any text or data as input. An example of such a tool would be a "diff" tool that detects differences between two files.

These pre-defined file types could be thought of as being specified in a standard derivation graph prelude of the form :

```
.composite ATOMIC "Any Composite File"
.compound ATOMIC "Any Compound File"
.derived    ATOMIC "Any Derived File"
.simple     ATOMIC "Any Atomic or Simple Derived File"
```

The diff tool could then be specified as :

```
diff "list of differences between a set of files" :
        USER diff.cmd
                : .compound
```

In addition, to allow control over how a given object is displayed, the linking file type

```
.view           DERIVED "The Form in which an Object is Displayed"
```

is provided. Before any object is displayed or transferred, the Odin system will attempt to perform the ".view" derivation from that object. If no such derivation can be performed, the original object is used. For example, if the derivation graph contains the specification :

```
c-name => .view
c-name "names of the elements of a composite file"* :
        NAME
                : .composite
```

then whenever a composite object is requested, the names of members of the composite object will be displayed.

3.5. Comments

Comments can be placed anywhere within the derivation graph. A comment is initiated with the sharp character ('#') and is terminated by the end-of-line character.

4. IMPLEMENTATION

The Odin command interpreter is currently implemented in the language C on a VAX 11/780, consisting of 13,000 lines of source code. It is being used to develop the TOOLPACK [Ostr 82] software environment, and a subset of Odin has been translated into FORTRAN-77 to be used as the command interpreter and tool integration mechanism for the portable TOOLPACK/IST software environment.

The Odin derivation graph compiler is also implemented in the language C on a VAX 11/780, and consists of 3,000 lines of source code. There exist Odin derivation graph specifications for the objects produced by most popular Unix tools, for all TOOLPACK objects, and for the objects created by the TREGRM/FSCAN [Clemm 83, Clemm 81] parser generating system.

5. ACKNOWLEDGEMENTS

Stu Feldman's Make process [Feld 79] and Lee Osterweil's Virtual File System [Ostr 81] are critical elements in the design and implementation of Odin. The need and motivation for a system such as Odin was provided by Lee Osterweil and the Toolpack project [Ostr 82].

REFERENCES

[Clemm 81] G. M. Clemm, "FSCAN Report", University of Colorado Technical Report #CU-CS-202-81, 1981.

[Clemm 83] G. M. Clemm, "TREGRM Report and User's Manual", University of Colorado Technical Report #CU-CS-249-83, 1983.

[Feld 79] Stuart I. Feldman, "Make--A Program for Maintaining Computer Programs," *Software–Practice and Experience* 9 (April 1979) pp. 255-265.

[Ostr 81] L. J. Osterweil, "Preliminary Toolpack Architectural Design", Dept. of Comp. Sci., Univ. of Colo., Boulder, Colo., 1981.

[Ostr 82] L. J. Osterweil, "Toolpack - An Experimental Software Development Environment Research Project", Proc. 6th Int. Conf. on Software Eng., Tokyo, pp.166-175.

Appendix A :

Syntax of Odin Commands

```
Command
  -> Display
  -> Transfer
  -> History
  -> Script
  -> HostCommand
  -> Help
  -> Variable ;

Display
  -> DerivSpec ;

Transfer
  -> DerivSpec '>' DerivSpec
  -> DerivSpec Editor ;

Editor
  -> '>' ':' ToolName
  -> '>' ;

History
  -> '!'
  -> '!' Editor
  -> '!' HistoryEntry
  -> '!' HistoryEntry Editor ;

Script
  -> '<' DerivSpec
  -> '<' '|' DerivSpec ;

HostCommand
  -> '%' "Word"+ ;

Help
  -> '?'
  -> '?' Topic
  -> ':' '?'
  -> DerivSpec ':' '?'
  -> DerivSpec '+' '?'
  -> DerivSpec '+' '?' ':' DerivType ;

Variable
  -> '?' '='
  -> VarName '=' '?'
  -> VarName '='
  -> VarName '=' VarValue ;

DerivSpec
  -> AtomicFN
  -> AtomicFN Operations ;
```

```
Operations
  -> Operation+ ;

Operation
  -> '+' Prm
  -> ':' DerivType
  -> '@' Element ;

Prm
  -> PrmKey
  -> PrmKey '=' PrmVal ;

PrmVal
  -> "Word"
  -> '(' PrmValFile ')' ;

PrmValFile
  -> DerivSpec ;

AtomicFN
  -> '|'
  -> "Word" ;

DerivType -> "Word"

HistoryEntry -> "Word" ;

Topic -> "Word" ;

PrmKey -> "Word" ;

ToolName -> "Word" ;

Element -> "Word" ;
```

Appendix B :

Syntax of Odin Specifications

```
DerivationGraph
  -> DGEntry + ;

DGEntry
  -> Derivation
  -> EquivalenceArc
  -> CastArc
  -> LinkingType
  -> AtomicType ;

Derivation
  -> OutputSpec ToolSpec SourceSpec ;

EquivalenceArc
  -> FileType '=>' FileType ;

CastArc
  -> FileType '=|' FileType ;

LinkingType
  -> FileType 'DERIVED' FileTypeDesc ;

AtomicType
  -> FileType 'ATOMIC' FileTypeDesc ;

OutputSpec
  -> OutputType FileTypeDesc ':' ;

OutputType
  -> FileType '<' Members '>'
  -> KeyedType ;

Members
  -> MemberSpec+ ;

MemberSpec
  -> KeyedType FileTypeDesc ;

KeyedType
  -> FileType '[' CmpdType ']'
  -> CmpdType ;

CmpdType
  -> FileType '(' RefType ')'
  -> FileType '^' RefType
  -> FileType ;

RefType
  -> FileType '@'
  -> FileType ;
```

```
ToolSpec
  -> 'USER' "Name"
  -> 'STRUCT'
  -> 'COMPOUND'
  -> 'COLLECT'
  -> 'FLATTEN'
  -> 'UNION'
  -> 'HOMOMORPHISM' HomomorphismSpec
  -> 'P-HOMOMORPHISM' HomomorphismSpec
  -> 'APPLY'
  -> 'SELECT'
  -> 'KEY'
  -> 'CAT'
  -> 'ERROR'
  -> 'WARNING'
  -> 'SENTINEL'
  -> 'NAME'
  -> 'COPYCHK'
  -> 'COPYTST' ;

SourceSpec
  -> (':' SourceType) + ;

SourceType
  -> FileType
  -> '(' FileType ')'
  -> 'PARAMETERS' '(' (PrmKey // ',') ')'
  -> "String" ;

HomomorphismSpec
  -> Operation+ ;

Operation
  -> '+' PrmKey '=' FileType
  -> '|' FileType ;

FileTypeDesc
  -> "String" '*'
  -> "String" ;

FileType -> "Name" ;

PrmKey -> "Name" ;
```

Appendix C :

Example of an Odin Specification

```
# Standard Specification Header

err "errors generated while producing derivation for display" :
        ERROR
                : (.stat)

warn "warnings generated while producing derivation for display" :
        WARNING
                : (.stat)

.derived => .stat

c-name => .stat
c-name => .view

c-name "names of the elements of a composite file"* :
        NAME
                : .composite


cat => .stat
cat => .view

cat "contents of a compound file" :
        CAT
                : (.compound)

name "names of the elements of a compound file" :
        NAME
                : .compound


error "errors generated while producing derivation" :
        ERROR
                : (.derived)


check (.null) "check the effect of a copy command file"* :
        COPYTST
                : (copy_sntU)
                : (copy_dsc)

copy_sntU (.null) "sentinels of the destinations of a copy command file"* :
        UNION
                : (copy_snt)

copy_snt (.null) "sentinels of the destinations of a copy command file"* :
        HOMOMORPHISM ( :sentinel )
                : (copy_dst)
```

```
copy_dsc (.null) "description of copy command"* :
        COLLECT
                : (copy_org)
                : (copy_dst)

copy_chk <
  copy_org (.null) "origin files in a copy command file"*
  copy_dst (.null) "destination files in a copy command file"*
  > "files in a copy command file"* :
        COPYCHK
                : .simple

compound (.null) "files named in a reference file"* :
        COMPOUND
                : .simple

key "key values (for selection)" :
        KEY
                : .simple

s-name => .view

s-name "names of the sentinels watching a file"* :
        NAME
                : sentinel

sentinel (.null) "sentinels watching a file" :
        SENTINEL
                : .simple
                : "| : compound"

.simple => .view


# Text

fmt-txt "formatted text" :
        USER nroff.cmd
                : txt
                : (all_so_ref)
                : PARAMETERS(m)

all_so_ref (null) "list of nroff-style transitively included files"* :
        COLLECT
                : ind_so_ref
                : so_ref

ind_so_ref (null) "list of nroff-style indirectly included files"* :
        HOMOMORPHISM |all_so_ref
                : so_ref
```

```
so_ref (null@) "list of nroff-style included files"* :
        USER inc_so.cmd
                : txt
txt ATOMIC "text with nroff constructs"

tbl => txt
tbl "output from tbl processor" :
        USER tbl.cmd
                : txtt
txtt ATOMIC "text with tbl and nroff constructs"

eqn => txtt
eqn "output from eqn processor" :
        USER eqn.cmd
                : txte
txte ATOMIC "text with eqn, tbl, and nroff constructs"


# RCS

c-rcs "C RCS version"* :
        USER co.cmd
                : vc
                : PARAMETERS(date, rev, state, who)
c-rcs => c

vc ATOMIC "C code stored in rcs format"
vc => rcs

f-rcs "Fortran RCS version"* :
        USER co.cmd
                : vf
                : PARAMETERS(date, rev, state, who)
f-rcs => f

vf ATOMIC "Fortran77 code stored in rcs format"
vf => rcs

log "log of changes to a source file" :
        USER rlog.cmd
                : rcs
                : PARAMETERS(date, rev, state, who)


# Fibonacci (every language must implement Fibonacci)

fib "Fibonacci function value" :
        USER fib.cmd
                : int
                : (fib_dep)
                : PARAMETERS(val)
```

```
fib_dep (fib) "input files needed to compute fib(n)"* :
        APPLY
                : int
                : fib_dep_desc

fib_dep_desc "two integers needed to compute fib(n)"* :
        USER fib_dep.cmd
                : int
                : PARAMETERS(val)

int ATOMIC "an integer"


# Parsing

parse_src (fcast) "source files for a tree-building parser/scanner program"* :
        COLLECT
                : f-parse
                : t_main
parse_src => cmpd

f-parse (f) "source files for a tree-building parser/scanner module"* :
        COLLECT
                : tgt
                : nodes
                : t_drive
                : pgt
                : p_drive
                : fst
                : f_drive

tregrm <
  tgt "tree-building tables"*
  tgt_lst "tregrm compiler listing"*
  pgi "parser grammar"*
  nodes "parse tree node types"*
  t_drive ~fcast@ "tree-building driver routines"*
  t_main ~fcast@ "tree-building parser default main program"*
  > "tree-building tables and parser grammar"* :
        USER tgt.cmd
                : tgi
                : fst
tgt =| f
nodes =| f

tgi ATOMIC "tree-building parser grammar"
```

```
lr <
  pgt "parser tables"*
  pgt_lst "parser listing"*
  p_drive ^fcast@ "parser driver routines"*
  > "parser tables"* :
        USER pgt.cmd
              : pgi
pgt =| f

scan_src (f) "source files for a scanner program"* :
        COLLECT
              : f-scan
              : f_main

f-scan (f) "source files for a scanner module"* :
        COLLECT
              : fst
              : f_drive

fscan <
  fst "scanner tables"*
  fst_lst "fscan compiler listing"*
  f_drive ^fcast@ "scanner driver routines"*
  f_main ^fcast@ "scanner default main program"*
  > "scanner tables"* :
        USER fst.cmd
              : fsi
fst =| f

fsi ATOMIC "scanner grammar"

tgi_fsi <
  tgi_ptr ^tgi "parser grammar"*
  fsi_ptr ^fsi "scanner grammar"*
  > "parser grammar - scanner grammar pair"* :
        STRUCT
              : tgiref

tgiref ATOMIC "tree-building parser/scanner grammars"


# Fortran-77 Formatted Source

fmt-f "formatted version of Fortran77 code" :
        USER pol.cmd
              : f
              : PARAMETERS(pol)
fmt-f =| f
```

```
# C Formatted Source

fmt-c "formatted version of C code" :
        USER pol_c.cmd
                : c
fmt-c =| c


# C and Fortran-77 Symbolic Debugger

dbx-flw <
  exe-flw ^null "executable for a dbx run"*
  srcs-flw (null) "sources for a dbx run"*
  keys-flw (null) "names of source files for a dbx run"*
  core-flw "core dump for a dbx run"*
  > "analysis of a core dump from a dbx run" :
        USER dbx_flw.cmd
                : dbx_flwin
                : PARAMETERS(id)
dbx =| dbx_flwin

dbx <
  exe-dbx ^null "executable for a dbx run"*
  srcs-dbx (null) "sources for a dbx run"*
  keys-dbx (null) "names of source files for a dbx run"*
  core-dbx "core dump for a dbx run"*
  > "Berkeley symbolic debugger run" :
        USER dbx.cmd
                : exe
                : (objsrcU)
                : (objkeyU)
                : PARAMETERS(id)


# Program Test Run

run <
  stdout "standard output from a test run (when +stdout is set)"
  output [data] "output files from a test run"
  core-run "core dump of a test run"*
  > "test run" :
        USER run.cmd
                : PARAMETERS(id,input,stdin,stdout)
                : exe
```

```
irun <
  script "transcript of an interactive test run"*
  stdin "standard input for an interactive test run"*
  istdout "standard output from an interactive test run"
  ioutput [idata] "output files from an interactive test run"
  core-irun "core dump of an interactive test run"*
  > "interactive test run"* :
        USER irun.cmd
              : exe
              : PARAMETERS(id,in)

# Library Archive

libmake <
  lib "object library archive"
  lib_lst "listing from an archive creation"*
  > "object library archive"* :
        USER lib.cmd
              : (objU)


# Executable Binary

exe "executable binary"* :
        USER exe.cmd
              : obj
              : PARAMETERS(debug,lib)

exe-C "executable binary from set of source files"* :
        USER exel.cmd
              : (objU)
              : PARAMETERS(debug,lib)
exe-C => exe


# Compiled Load Module

obj DERIVED "object module"


objU(obj) "set of object modules"* :
        UNION
              : (objC)

objC(obj) "list of object modules"* :
        HOMOMORPHISM |obj
              : (cmpd)

objkeyU(obj_key) "set of source names for object modules"* :
        UNION
              : (objkeyC)
```

```
objkeyC(obj_key) "list of source names for object modules"* :
        HOMOMORPHISM |obj_key
                : (cmpd)

objsrcU(obj_src) "set of source files for object modules"* :
        UNION
                : (objsrcC)

objsrcC(obj_src) "list of source files for object modules"* :
        HOMOMORPHISM |obj_src
                : (cmpd)


# C Source Code

obj_c <
  obj-c "C object module"*
  obj_key-c "C source code file name"*
  obj_src-c ^null "C source code"*
  > "C object module information"* :
        USER obj_c.cmd
                : PARAMETERS(debug)
                : (all_c_ref)
                : ckey
                : c
obj-c => obj
obj_key-c => obj_key
obj_src-c => obj_src

all_c_ref (null) "list of C-style transitively included files"* :
        COLLECT
                : ind_c_ref
                : c_ref

ind_c_ref (null) "list of C-style indirectly included files"* :
        P-HOMOMORPHISM |all_c_ref
                : c_ref

c_ref-c (null@) "list of C-style included files"* :
        USER inc_c.cmd
                : PARAMETERS(ignore)
                : c
c_ref-c => c_ref

ckey "name of C file"* :
        KEY
                : c

c ATOMIC "C source code"
```

```
# C Include Files

c_ref-h (null@) "list of C-style included files"* :
        USER inc_c.cmd
                : PARAMETERS(ignore)
                : h
c_ref-h => c_ref

h ATOMIC "C Include Data"


# M4 Pre-Processor for Fortran77

f-m4 "Fortran77 output from M4 pre-processor"* :
        USER m4.cmd
                : PARAMETERS(pdef)
                : (all_m4_ref)
                : mf
f-m4 => f

all_m4_ref (null) "list of M4-style transitively included files"* :
        COLLECT
                : ind_m4_ref
                : m4_ref

ind_m4_ref (null) "list of M4-style indirectly included files"* :
        P-HOMOMORPHISM |all_m4_ref
                : m4_ref

m4_ref-mf (null@) "list of M4-style included files"* :
        USER inc_m4.cmd
                : PARAMETERS(pdef)
                : mf
m4_ref-mf => m4_ref

mf ATOMIC "Fortran77 source code with M4 constructs"


# M4 Include Files

m4_ref-i (null@) "list of M4-style included files"* :
        USER inc_m4.cmd
                : PARAMETERS(pdef)
                : i
m4_ref-i => m4_ref

i ATOMIC "M4 Include Data"
```

```
# Fortran-77 Source Code

obj_f <
  obj-f "Fortran77 object module"*
  obj_key-f "Fortran77 source code file name"*
  obj_src-f ^null "Fortran77 source code"*
  > "Fortran77 object module information"* :
        USER obj_f.cmd
                : PARAMETERS(debug)
                : fkey
                : f
obj-f => obj
obj_key-f => obj_key
obj_src-f => obj_src

fkey "Fortran77 source code file name"* :
        KEY
                : f

f ATOMIC "Fortran77 source code"

fcast =| f


# Reference Files

cmpd-cf(f) "files specified in a "ref" file" :
        COMPOUND
                : ref
cmpd-cf => cmpd
cmpd DERIVED "files specified in a

ref ATOMIC "reference file containing a list of file names"
```

Appendix D :

Examples of Odin Command Files

Following are the Berkeley Unix 4.2 command files for the derivation graph example in Appendix C.

co.cmd :

```
cd $(RUNDIR)

set dflag = ''
if (-e $(PRM)/date) set dflag = -d'cat $(PRM)/date'
set rflag = ''
if (-e $(PRM)/rev) set rflag = -r'cat $(PRM)/rev'
set sflag = ''
if (-e $(PRM)/state) set sflag = -s'cat $(PRM)/state'
set wflag = ''
if (-e $(PRM)/who) set wflag = -w'cat $(PRM)/who'

ln -s $(<1) $(<1),v
(co -p -q $dflag $rflag $sflag $wflag $(<1),v >! $(>1)) >&! $(ERROR)

echo 0 >! $(OK)
```

dbx.cmd :

```
cd $(RUNDIR)

mkdir src
set sources = 'cat $(objsrcU)'
set keys = 'cat $(objkeyU)'
@ i = 1
while ($i <= $#sources)
  ln -s $sources[$i] src/'cat $keys[$i]'
  @ i = $i + 1
  end

dbx -I src $(exe)

echo $(exe) >! $(exe-dbx)
cat $(objsrcU) >! $(srcs-dbx)
cat $(objkeyU) >! $(keys-dbx)
if (-e core) mv core $(core-dbx)

rm -f -r src
echo 0 >! $(OK)
```

dbx_flw.cmd :

```
cd $(RUNDIR)

set dbxENV = 'cat $(dbx_flwin)'

set exeptr = $dbxENV[1]
set srcptr = $dbxENV[2]
set keyptr = $dbxENV[3]
set core = $dbxENV[4]

mkdir src
set sources = 'cat $srcptr'
set keys = 'cat $keyptr'
@ i = 1
while ($i <= $#sources)
  ln -s $sources[$i] src/'cat $keys[$i]'
  @ i = $i + 1
  end
if (! -z $core) cp $core core

dbx -I src $exeptr

echo $exeptr >! $(exe-flw)
cat $srcptr >! $(srcs-flw)
cat $keyptr >! $(keys-flw)
if (-e core) mv core $(core-flw)

rm -f -r src
echo 0 >! $(OK)
```

eqn.cmd :

```
(eqn < $(txtt) >! $(tbl)) >& $(ERROR)

echo 0 >! $(OK)
```

exe.cmd :

```
cd $(RUNDIR)

cp $(obj) SOURCE.o

set compiler = 'cc'
if ("'nm -gp SOURCE.o | fgrep 'T _MAIN_'" !~ '') set compiler = 'f77'

set flags = ''
if (-e $(PRM)/debug) set flags = '-g'
set libs = ''
if (-e $(PRM)/lib) set libs = `cat $(PRM)/lib`

($compiler $flags SOURCE.o $libs -o $(exe)) >&! $(ERROR)

echo 0 >! $(OK)
```

exel.cmd :

```
cd $(RUNDIR)

set files = `cat $(objU)`

set search = `nm -gp $files | fgrep 'T _MAIN_'`
set compiler = 'cc'
if ("$search" !~ '') set compiler = 'f77'

set flags = ''
if (-e $(PRM)/debug) set flags = '-g'
set libs = ''
if (-e $(PRM)/lib) set libs = `cat $(PRM)/lib`

($compiler $flags $files $libs -o $(exe-C)) >&! $(ERROR)

echo 0 >! $(OK)
```

fib.cmd :

```
@ val = `cat $(int)`
if (-e $(PRM)/val) @ val = `cat $(PRM)/val`

if ($val < 2) then
  @ result = $val
else
  set vals = `cat $(fib_dep)`
  @ valone = `cat $vals[1]`
  @ valtwo = `cat $vals[2]`
  @ result = $valone + $valtwo
  endif

echo $result > $(fib)

echo 0 >! $(OK)
```

fib_dep.cmd :

```
@ val = `cat $(int)`
if (-e $(PRM)/val) @ val = `cat $(PRM)/val`

if ($val > 1) then
  @ valone = $val - 1
  @ valtwo = $val - 2
  echo '+val=' $valone ':fib' > $(fib_dep_desc)
  echo '+val=' $valtwo ':fib' >> $(fib_dep_desc)
  endif

echo 0 >! $(OK)
```

fst.cmd :

```
cd $(RUNDIR)

($(TOOL)/fscan < $(fsi) >! $(fst_lst)) >&! $(ERROR)

if (-e TABLES) mv TABLES $(fst)
echo $(TOOL)/fst_drive.f >! $(f_drive)
echo $(TOOL)/fst_main.f >! $(f_main)

echo 0 >! $(OK)
```

inc_c.cmd :

```
cd $(RUNDIR)

set ignore = ""
if (-e $(PRM)/ignore) set ignore = $(PRM)/ignore

($(TOOL)/inc_c.exe $(CURDIR) $ignore < $(<1) >! $(>1)) >&! $(ERROR)

echo 0 >! $(OK)
```

inc_m4.cmd :

```
cd $(RUNDIR)

set pdef = ""
if (-e $(PRM)/pdef) set pdef = `cat $(PRM)/pdef`

($(TOOL)/inc_m4.exe $pdef < $(<1) >! $(>1)) >&! $(ERROR)

echo 0 >! $(OK)
```

inc_so.cmd :

```
cd $(RUNDIR)

($(TOOL)/inc_so.exe < $(<1) >! $(>1)) >&! $(ERROR)

echo 0 >! $(OK)
```

irun.cmd :

```
cd $(ioutput)
if (-e $(PRM)/in) then
  $(TOOL)/catchio.exe -i $(stdin) -o $(istdout) -e $(WARNING) \
   -s $(script) $(exe) < `cat $(PRM)/in`
  if ($status != 0) echo run failed >>! $(ERROR)
else
  $(TOOL)/catchio.exe -i $(stdin) -o $(istdout) -e $(WARNING) \
   -s $(script) $(exe)
  if ($status != 0) echo run failed >>! $(ERROR)
endif

if (-e core) mv core $(core-irun)

echo 0 >! $(OK)
```

lib.cmd :

```
  cd $(RUNDIR)

  set files = `cat $(objU)`

  rm -f $(lib)
  (ar rv $(lib) $files >! $(lib_lst)) | sed '/creat/d' >&! $(ERROR)
  (ranlib $(lib) >>! $(lib_lst)) >>&! $(ERROR)

  echo 0 >! $(OK)
```

m4.cmd :

```
  set mac = m4
  if (-e $(PRM)/tie) set mac = $(TOOL)/TIEMAC

  cd $(CURDIR)
  (m4 < $(mf) >! $(f-m4)) >& $(ERROR)

  echo 0 >! $(OK)
```

nroff.cmd :

```
  set mflag = ' '
  if (-e $(PRM)/m) set mflag = -m`cat $(PRM)/m`

  cd $(CURDIR)
  (nroff $mflag < $(txt) >! $(fmt-txt)) >& $(ERROR)

  echo 0 >! $(OK)
```

obj_c.cmd :

```
  cd $(RUNDIR)

  set source = `cat $(ckey)`
  if ($source:e != 'c') set source = $source.c
  ln -s $(c) $source
  set flags = ''
  if (-e $(PRM)/debug) set flags = '-g'

  (cc -c $flags -I$(CURDIR) $source) >&! ERRORS

  cat ERRORS
  sed -n '/rror/p' < ERRORS >! $(ERROR)
  sed -n '/arning/p' < ERRORS >! $(WARNING)

  if (-e $source:r.o) mv $source:r.o $(obj-c)
  echo "$source" >! $(obj_key-c)
  echo "$(c)" >! $(obj_src-c)

  echo 0 >! $(OK)
```

obj_f.cmd :

```
  cd $(RUNDIR)

  set source = `cat $(fkey)`
  if ($source:e != 'f') set source = $source.f
  ln -s $(f) $source

  set flags = ''
  if (-e $(PRM)/debug) set flags = '-g'

  (f77 $flags -c $source) >&! ERRORS

  sed -n '/rror/p' < ERRORS >! $(ERROR)
  sed -n '/arning/p' < ERRORS >! $(WARNING)
  if (-e $source:r.o) mv $source:r.o $(obj-f)
  echo "$source" >! $(obj_key-f)
  echo "$(f)" >! $(obj_src-f)

  echo 0 >! $(OK)
```

pgt.cmd :

```
  cd $(RUNDIR)

  ($(TOOL)/lr < $(pgi) >! $(pgt_lst)) >&! $(ERROR)
  mv TABLES $(pgt)
  echo $(TOOL)/pgt_drive.f >! $(p_drive)

  echo 0 >! $(OK)
```

pol.cmd :

```
cd $(RUNDIR)

cp $(f) SOURCE
if (-e $(PRM)/polish && ! -z $(PRM)/polish) cp `cat $(PRM)/polish` P77PAR
if ((! -e P77PAR) || -z P77PAR) cp $(TOOL)/P77PAR P77PAR

($(TOOL)/polish) >&! $(ERROR)

sed '/^ *$/d' < ERRPOL >> $(ERROR)
sed '/^ *$/d' < WRNPOL >! $(WARNING)
sed 's/ *$//' < PRETTY >! $(fmt-f)

echo 0 >! $(OK)
```

pol_c.cmd :

```
cd $(RUNDIR)

(indent $(c) $(fmt-c)) >&! $ERROR

echo 0 >! $(OK)
```

rlog.cmd :

```
cd $(RUNDIR)

set dflag = ''
if (-e $(PRM)/date) set dflag = -d`cat $(PRM)/date`
set rflag = ''
if (-e $(PRM)/rev) set rflag = -r`cat $(PRM)/rev`
set sflag = ''
if (-e $(PRM)/state) set sflag = -s`cat $(PRM)/state`
set wflag = ''
if (-e $(PRM)/who) set wflag = -w`cat $(PRM)/who`

ln -s $(<1) $(<1),v
(rlog $dflag $rflag $sflag $wflag $(<1),v >! $(>1)) >&! $(ERROR)

echo 0 >! $(OK)
```

run.cmd :

```
cd $(output)

if (-e $(PRM.DIR)/input) then
  set input = '(cd $(PRM.DIR)/input; ls)'
  endif
if (-e $(PRM)/stdin) then
  set stdin = 'cat $(PRM)/stdin'
  endif
if (-e $(PRM)/stdout) then
  set stdout = '$(stdout)'
  endif

if ($?input) ln -s $(PRM.DIR)/input/* .

if ($?stdin && $?stdout) then
  ($(exe) < $stdin >! $stdout) >&! $(WARNING)
else if ($?stdin) then
  $(exe) < $stdin
else if ($?stdout) then
  ($(exe) >! $stdout) >&! $(WARNING)
else
  $(exe)
endif

if ($status != 0) echo run failed >>! $(ERROR)

if ($?input) then
  rm -f $input
  endif

if (-e core) mv core $(core-run)

echo 0 >! $(OK)
```

tbl.cmd :

```
(tbl < $(txtt) >! $(tbl)) >& $(ERROR)

echo 0 >! $(OK)
```

tgt.cmd :

```
cd $(RUNDIR)

ln -s $(fst) INPUT2
($(TOOL)/tregrm < $(tgi) | sed 's/  *$$//' >! $(tgt_lst)) >&! $(ERROR)

if (-e TABLE1) mv TABLE1 $(tgt)
if (-e TABLE2) mv TABLE2 $(pgi)
if (-e TABLE3) mv TABLE3 $(nodes)
echo '$(TOOL)/tgt_drive.f' >! $(t_drive)
echo '$(TOOL)/tgt_main.f' >! $(t_main)

echo 0 >! $(OK)
```