THE ODIN SYSTEM

REFERENCE MANUAL

Geoffrey M. Clemm

*** Odin Version 1.17 ***

The Odin System is a simpler, more powerful, and more reliable replacement for Make. It computes complete dependency information automatically, making the build scripts shorter and easier to manage. Odin gains efficiency by parallel builds on remote machines, by eliminating most of the file system status queries required by Make, and by sharing from a cache of previously computed derived files.

Contents

1 Introduction **1**
- 1.1 Derived Object Managers . . . 1
- 1.2 Odin Objects . . . 1
  - 1.2.1 Source and Derived Objects . . . 1
  - 1.2.2 The Derived Object Cache . . . 2
- 1.3 Why Not Just Use Make? . . . 2
  - 1.3.1 Dependency Information . . . 2
  - 1.3.2 Specifying Build Variants . . . 3
  - 1.3.3 Storing Derived File Variants . . . 3
  - 1.3.4 Parallel Builds . . . 4
  - 1.3.5 Help . . . 4
  - 1.3.6 Errors . . . 4
  - 1.3.7 Operations on Lists . . . 4
  - 1.3.8 Editing while Building . . . 4
  - 1.3.9 Recursive Implicit Rules . . . 4
  - 1.3.10 Circular Dependencies . . . 5
  - 1.3.11 Efficient Dependency Computation . . . 5
  - 1.3.12 Tool Dependencies . . . 5
  - 1.3.13 Availability . . . 5

2 The Odinfile **6**
- 2.1 Targets . . . 6
- 2.2 Odin-Expressions . . . 6
  - 2.2.1 Lexical Conventions . . . 6
  - 2.2.2 Selection Expressions . . . 7
  - 2.2.3 Derivation Expressions . . . 7
  - 2.2.4 Parameterization Expressions . . . 8
  - 2.2.5 String Expressions . . . 9
- 2.3 Virtual Targets . . . 10
- 2.4 Text Targets . . . 10
  - 2.4.1 Executable Text Targets . . . 11
- 2.5 Command Targets . . . 11
- 2.6 Nested Target Definitions . . . 11

3 Getting Started **13**
- 3.1 Systems with Several Files . . . 13
- 3.2 Files in Other Directories . . . 14
- 3.3 Recursive Odin Calls . . . 15
- 3.4 Debugging Your Program . . . 15

4 Odin-Commands **17**
4.1 Query Odin-Commands . . . 17
4.1.1 Status of Odin Objects . . . 17
4.1.2 Error and Warning Messages . . . 18
4.2 Copy Odin-Commands . . . 18
4.2.1 Copying to Standard Output . . . 18
4.2.2 Editing with the Copy Odin-Command . . . 19
4.3 Execute Odin-Commands . . . 19
4.4 Variable Assignment Odin-Commands . . . 19
4.4.1 Dir . . . 20
4.4.2 BuildHosts, MaxBuilds . . . 20
4.4.3 KeepGoing . . . 20
4.4.4 History . . . 20
4.4.5 ErrLevel, WarnLevel, LogLevel . . . 20
4.4.6 HelpLevel . . . 21
4.4.7 VerifyLevel . . . 21
4.4.8 Size . . . 21
4.4.9 Environment Variables . . . 21
4.5 Help . . . 22
4.5.1 Source Type Help . . . 22
4.5.2 Derivation Help . . . 22
4.5.3 Parameterization Help . . . 23
4.5.4 Variable Help . . . 23

A Dependency Database Utilities **25**
A.1 Testing Host Files for Changes . . . 25
A.2 Redoing a Particular Derivation Step . . . 26
A.3 Browsing the Dependency Database . . . 26

B Tool Package Libraries **27**
B.1 Tool Packages . . . 27
B.1.1 Tool Package Selection . . . 27
B.1.2 Versioned Tool Packages . . . 28
B.2 Derivation Graphs . . . 28
B.3 Source Declarations . . . 28
B.4 Object Type Declarations . . . 29
B.4.1 Built-In Supertypes . . . 30
B.4.2 Built-In Derived Object Types . . . 30
B.5 Parameter Type Declarations . . . 33
B.6 Environment Variable Declarations . . . 33
B.7 Tool Declarations . . . 34
B.7.1 Input-Expressions . . . 34
B.7.2 Ignoring the Status of a Tool Input . . . 35
B.7.3 Ignoring Changes to the Value of a Tool Input . . . 35
B.7.4 EXEC Tool . . . 35
B.7.5 READ-LIST Tool . . . 36
B.7.6 COLLECT Tool . . . 36
B.7.7 READ-REFERENCE Tool . . . 37

C Derivation Graph Syntax **38**

D Derivation Graph Example **40**

Chapter 1

Introduction

1.1 Derived Object Managers

A software environment can be simplified if the user's attention is focused on the information provided by the environment rather than the tools that create this information. A *derived object manager* provides this focus by automating the process of tool invocation. A derived object manager responds to a request for a piece of computed information, or *derived object*, by invoking the minimal number of tools necessary to produce that object. If previously computed objects are automatically stored by the object manager for later re-use, significant improvements in response time can be achieved.

In an extensible environment, the kinds of information potentially provided are extended by adding new tools that manipulate and generate new kinds of information. A few examples of the kind of information in a software environment are program source text, test data, modification histories, attributed syntax trees, compiled object code, data flow analysis, and results of symbolic execution.

A derived object manager must provide a language with which a user or a tool can name any desired object, and a specification language for describing the kinds of objects to be managed and the tools that produce them. In the Odin system, derived objects are named by *odin-expressions*, and the specification language is an extended production system called a *derivation graph* in which each tool is described by a single declaration.

1.2 Odin Objects

Each Odin object is either a *file*, a *string*, or a *list* (an ordered sequence of objects). An example of a file is a file containing source code, an executable binary file, or an output file from a test run. An example of a string is a command flag. An example of a list is the source files of a program or the arguments to a command.

1.2.1 Source and Derived Objects

Source objects are files that can be created or modified directly by the user. They can be regular files, directories, or symbolic links. Source objects cannot be automatically recreated by the Odin system, but are the basic building blocks from which the Odin system creates all other objects. Every source object is given a type by the Odin system based on its host filename, and this type determines what derived objects can be produced from the source object.

Derived objects are objects that can be produced from source objects and other derived objects through the invocation of one or more tools. Tools are invoked only as needed to create a specified derived object. The Odin system automatically saves objects from previous requests, so a given object might already exist and therefore be available immediately. Derived objects are created and modified only by the Odin system

itself, not by users. Examples of objects that can be derived from source code are a cross reference listing, executable binary code, or the list of files referenced by `#include` statements.

1.2.2 The Derived Object Cache

In the Odin system, all derived objects are stored in a directory called the *derived object cache*, or simply the *cache*. The cache also contains the database that stores the *depends* relationship between the output and input files of a tool run, and the *contains* relationship between a list and its elements.

All host files other than those in the cache are source objects. A derived object can be copied from the cache into a source directory, but this does *not* make the file in the source directory a derived object – it just creates a source object that happens to have the same contents as the derived object.

The default location for the cache is a directory named `.ODIN` in the user's home directory. A non-default cache location can be specified in the `$ODIN` environment variable, or with the `-c` option to the `odin` host-command. The main reasons for specifying a non-default location for the cache are to share a common cache with other users, or to locate the cache on a local disk for more efficient access to derived files.

The default name for a file in the cache is `label.id.type`, where `label` and `id` are the results of `:label` and `:id` derivations (see Section B.4.2), and where `type` is the type of the file. If this filename is too long for your operating system, set the environment variable $ODIN_SHORTNAMES and a file in the cache will be named simply `id.type`.

The default inter-process communication mechanism for the odin cache manager process is TCP/IP. If TCP/IP is not available, set the environment variable $ODIN_LOCALIPC and Unix domain sockets will be used instead.

1.3 Why Not Just Use Make?

The most ubiquitous build manager is Stu Feldman's Make program[**?**], with its many descendents[**?**][**?**][**?**][**?**]. Although ideal for small projects, Make has two major problems when used for large or complex projects: inaccurate dependency information and poor support for variant builds (such as builds for different architectures or builds with different levels of compiler optimization). Inaccurate dependency information leads to inefficient builds when unnecessary recomputations are performed and, more seriously, incorrect builds when necessary recomputations are not performed. Poor support for variant builds consists primarily of problems with specifying the variants and with storing the intermediate files for the variants. An indication of the difficulty of specifying a complex build variant is that often the only way to find out what a given build description means is to run Make and see what kind of commands it executes.

1.3.1 Dependency Information

Inaccurate dependency information results from a variety of causes. If the user maintains the dependency information by hand, it is virtually guaranteed to be incorrect. A "make-depend" tool can automate the production of dependency information, but because of the expense of running such a tool on an entire system, it is usually not automatically invoked before each build, again leading to builds based on incorrect dependency information. If a separate dependency file is created for each source file to allow incremental dependency computation, it becomes feasible to automatically compute dependency information before each build. Unfortunately, the source directories then become cluttered with dependency information files, and the overhead of opening and reading all the dependency information files becomes excessive.

In Odin, tools that automatically gather dependency information are as easy to describe as standard tools like compilers, and Odin takes care of running them incrementally when they are needed. The results of these tools are stored in a persistent database rather than in the user's Makefiles, so that this information can be retrieved efficiently and does not clutter up source directories.

Other dependency problems stem from Make's use of file modification time ordering to determine whether a file is up-to-date. Network clock inconsistencies and the use of tools that restore files with their original modification dates can result in changed sources files whose modification dates are less than those of derived

files that depend on them. In Odin, this problem is avoided by storing the date stamps of the inputs to a build step in the dependency database, so that any change to the modification time of a file (either earlier or later) triggers a re-build.

The use of file modification time ordering can also result in inefficient builds when a change to a source file does not result in a corresponding change to an intermediate derived file. For example, adding a comment to a source file will usually not change the result of compiling that file, which means that there is no reason to relink the unchanged object code to form a new executable. Odin avoids this inefficiency by re-running a tool only if the *value* of one of its inputs has changed. In particular, this allows Odin to build directly from an RCS `xxx,v` or an SCCS `s.xxx` file, without first checking out a version into a source directory (the check-out operation is just a tool like any other tool). This is not feasible in Make because every time a new version is added to the version control file, it would assume everything derived from *any* version of that file must be recomputed.

1.3.2 Specifying Build Variants

In Make, a derived file variant is specified by changing the values of the appropriate Make variables. For example, the `CC` variable can be changed to build with a different C compiler, and the `CCFLAGS` variable can be changed to build with different C compiler flags. Unfortunately, the scope of a variable is the entire Makefile, while a build variant often involves different variable values for different intermediate files (for example, most of the files are to be compiled with debugging on, except for certain files containing time critical routines which are to be compiled with optimization on). To work around this problem, in Make you have to introduce a special build rule for each intermediate file build variant.

In Odin, a filename can be extended by an arbitrary number of build *parameters*, that are then used by Odin to customize the way that file is processed in any build step that refers to it. For example,

```
prog.c +debug
```

indicates that the debug flag should be set whenever `prog.c` is used in a build step, while

```
io.c +optimize=2
```

indicates that optimization level 2 should be set whenever `io.c` is used in a build step. If the parameterized filename refers to a list of other files, the parameters are inherited by each of the referenced files. Because of the controlled scoping provided by Odin parameters and the expressive power of Odin implicit build rules, explicit build rules are hardly ever needed in an Odinfile.

1.3.3 Storing Derived File Variants

Make usually stores a derived file in the directory containing the source file from which it is derived, since this allows other builds that reference the source file to reuse that derived file. The main disadvantage of this approach is that only one variant of a given derived file can exist at a given time. The build manager can easily mistake which version is currently in the directory, leading to an incorrect build. Other disadvantages are that a user must have write permission to the source directories in order to re-build derived files, a user can mistakenly delete a source file thinking it is a derived file, and a user can mistakenly edit a derived file thinking it is a source file.

The usual approach to this problem is to have the user explicitly allocate different directories to contain the results of different variant builds. Unfortunately, this makes it difficult for build sharing to take place, and unless the version of Make has been extended to write out the variable values used for the builds in a given directory, it is easy to get incorrect builds if different variable values are used for builds in the same directory.

Another approach to this problem used in ClearMake[**?**] consists of introducing a a proprietary *multi-version file system*. Every file opened for reading during the build and every variable value is tracked by the system, and if the same set of files with the same variables are present in a later build request, the previously computed results of the build are re-used. Even this approach, though, requires that the user explicitly manage variant derived file locations if concurrent variant builds of files local to a view are desired.

The alternative approach used by Odin that does not require replacing the native Unix file system is to use a derived file cache. In effect, the file namespace is extended to provide names for every derived file variant. For example,

```
/usr/src/prog.c :exe
```

is the executable computed by compiling and linking `/usr/src/prog.c`,

```
/usr/src/prog.c +debug :exe
```

is the executable with debugging information, and

```
/usr/src/prog.c +optimize=2 :exe
```

is the executable compiled with level-2 optimization. These extended names are parsed by the Odin interpreter, which either maps the name to a source file or to the correct derived file in the derived file cache.

1.3.4 Parallel Builds

Odin provides concurrent parallel builds on local and remote hosts. You just specify a list of build hosts and the maximum number of parallel builds.

1.3.5 Help

Odin provides a powerful help system that uses its knowledge of the currently installed set of implicit rules (including those added by the local site administrator and an individual user). In particular, it can tell you what kinds of file extensions are currently recognized, what kinds of files can be derived from a specific type of file, and what parameters can be specified to produce a variant of a given derived file.

1.3.6 Errors

Odin maintains a persistent database of all error and warning messages associated with each build step. You can request a summary report of all errors and/or warnings associated with all the build steps of a particular derived object. For derived object variants, any valid error information from other variants is reused from the database rather than recomputed.

1.3.7 Operations on Lists

Enhanced versions of Make each provide their own idiosyncratic set of operations for manipulating filenames and lists of filenames. In Odin, these operations are performed by arbitrary Unix tools – this is feasible because Odin caches the results of these tools in the dependency database.

1.3.8 Editing while Building

You can edit files that are currently being used for one or more builds, and Odin will automatically recognize the changes and redo the parts of the builds that are affected by the changes, before letting the builds complete.

1.3.9 Recursive Implicit Rules

Odin allows you to recursively chain implicit rules, so for example, you can specify that the files included by a file are the files directly included by the file plus the files included by the files included by the file. Even enhanced versions of Make that do support chaining of implicit rules do not allow recursive chaining.

1.3.10 Circular Dependencies

Odin allows a file to depend on itself. This may seem unreasonable until you start writing tools that generate code, and you find that it would be very useful to have the tool generate part of itself. In this situation, Odin will terminate the computation when the tool produces a file that is identical to the one used as input to the computation (i.e., when the computation reaches a *fixed-point*).

1.3.11 Efficient Dependency Computation

In very large systems, even when a desired derived file is up-to-date, simply `fstat()`'ing all the source files upon which it depends can take Make several minutes. Since Odin can broadcast file change information in its dependency database, dependency computation can be made proportional to the number of file dependencies that have *changed* rather than the total number of file dependencies.

1.3.12 Tool Dependencies

Build steps automatically depend on the tools and scripts that produce them, so if you change a tool, Odin will recompute anything produced by that tool the next time you request a build. If the tool change did not affect a particular run of that tool, the recomputation is short-circuited, and the rest of the derived files that depend on that output of the tool are marked as still being valid.

1.3.13 Availability

Finally, Odin is portable, free, and available in source form. It can be retrieved via anonymous ftp from the compressed tar formatted file `distribs/odin/odin.tar.Z` at `ftp.cs.colorado.edu (128.138.243.151)`.

Chapter 2

The Odinfile

2.1 Targets

A build *target* in a directory is a source file whose value is computed by the derived object manager. Just as Make uses a file named `Makefile` to specify targets for a directory, Odin uses a file named `Odinfile`. When the host-command:

```
odin prog
```

is executed, Odin checks to see if `prog` is a target in `./Odinfile`. If it is a target, but is not up-to-date, Odin invokes whatever tools are necessary to bring it up-to-date. Odin automatically looks in all directories that contain the input files for these tools, and if there is an `Odinfile` that contains a target for an input file, Odin ensures that the input file is up-to-date before using it.

A target definition in an `Odinfile` consists of a filename followed by two equal-signs and an *odin-expression*. For example, the `Odinfile` entry:

```
prog == prog.c +debug :exe
```

declares that the value of the source file `prog` should always be equal to the value of the file specified by the odin-expression `prog.c+debug:exe`.

2.2 Odin-Expressions

Just as any source file can be named with a source filename, any derived file can be named with an *odin-expression*. An odin-expression is just like a source filename, except that in addition to the slash operator used in source filenames for selecting elements of directories, there is a plus-sign operator for adding parameters to an object and a colon operator for selecting a derivation from an object. For example, in the odin-expression:

```
src/prog.c+debug:exe
```

`prog.c` is selected from the `src` directory, the parameter `+debug` is associated with `src/prog.c`, and the `:exe` (i.e. executable binary) derivation is selected from `src/prog.c+debug`.

2.2.1 Lexical Conventions

Lexically, an odin-expression is composed of a sequence of *identifier* and *operator* tokens, and is terminated by a new-line character. An odin-expression can be continued on multiple lines by escaping each new-line character with a backslash. This backslash (but not the new-line) is deleted before the expression is parsed. Multiple odin-expressions can be specified on the same line by separating them with semi-colon operators.

An identifier token is just a sequence of characters. The following characters:

```
: + = ( ) / % ; ? $ < > ! <space> <tab> <new-line> # \ '
```

must be escaped to be included in an identifier. A single character can be escaped by preceding it with a backslash (e.g. `lost\+found`). A sequence of characters can be escaped by enclosing them in single quote marks (e.g. `'lost+found'`).

Unescaped *whitespace* characters (spaces, tabs, and new-lines) are ignored during parsing except when they separate adjacent identifiers. A comment is indicated by a sharp and is terminated by a new-line character. For example, the odin-expression:

```
# this is a comment
```

is a comment and is equivalent to whitespace.

An odin-expression can be surrounded by parentheses. Parentheses are required for nested odin-expressions (such as values of parameters) or for the empty expression `()` which represents an immutable empty file.

2.2.2 Selection Expressions

A selection expression, indicated by the slash operator, selects a file from a directory. The argument to the slash operator is the *key* of the desired file. For example, the odin-expression:

```
src/prog.c
```

selects key `prog.c` from the directory `src`. (The careful reader will note that an odin-expression composed entirely of selection expressions bears an uncanny resemblance to a standard Unix filename.) Any special character in the key must be escaped. For example, `prog.c++` must be escaped, as in `prog.c\+\+` or `'prog.c++'`.

Odin splits each key into two parts: a *root* and a *type-name*. The type-name is the longest prefix of the key that matches one of the declared source types prefixes (see Section B.3). If no prefix match is found, the type-name is the longest suffix of the key that matches one of the declared source type suffixes. If no suffix match is found, the type-name is the empty string.

Assuming that `s.` is the only declared source type prefix and `.c` is the only declared source type suffix, then the following relationships hold:

FILENAME	KEY	ROOT	TYPE-NAME
src/prog.c	prog.c	prog	.c
src/s.prog.c	s.prog.c	prog.c	s.
src/prog	prog	prog	

The type-name is used to determine the object type of the file (see Section B.3), and the root is used to determine the object type of generic objects derived from the file (see Section B.4.1).

2.2.3 Derivation Expressions

A derivation expression, indicated by the colon operator, is used to specify a derived object. The argument to the colon operator is an *object type* that has been declared in one of the tool packages (see Section B.4). For example, the odin-expression:

```
prog.c :fmt
```

names a formatted version of `prog.c`, and the odin-expression:

```
prog.c :fmt :exe
```

names the result of compiling and linking the formatted version of `prog.c`.

A derived object can be a directory, in which case it is called a *derived directory*. Elements of a derived directory are selected with the same slash operator used to select elements of source directories. For example, if `src/prog.c:output` is a derived directory containing the output files from a test run of `prog.c`, and this directory contains three files named `DATA`, `source.listing`, and `source.errors`, then these three files are named by the odin-expressions:

```
src/prog.c:output/DATA
src/prog.c:output/source.listing
src/prog.c:output/source.errors
```

An element of a derived directory cannot be another directory, although it can be a symbolic link to another directory.

2.2.4 Parameterization Expressions

A parameterization expression, indicated by the plus-sign operator, extends an object with additional information that affects the derived objects produced from that object. The argument to the plus-sign operator is a *parameter type* (see Section B.5), optionally followed by an equal-sign operator and a sequence of one or more identifiers and parenthesized object-expressions. The identifiers specify string objects. For example, in the odin-expression:

```
prog.c +lib=(support.c.sm +debug :a) +lib=termcap
```

the value of the first `+lib` parameter is the object `support.c.sm+debug:a` and the value of the second `+lib` parameter is the string `termcap`. An example of an odin-expression with a parameter value that is a sequence is:

```
prog.c +lib=dbm (support.c.sm +debug :a) socket
```

which specifies that the `+lib` parameter has a value that is a sequence consisting of the string `dbm`, the object `support.c.sm+debug:a` and the string `socket`.

If the parameter value is omitted, it is equivalent to specifying the identifier consisting of a single space `' '` as the value. For example, the following two odin-expressions:

```
prog.c +debug
prog.c +debug=' '
```

are equivalent.

The parameter values of a given parameter type in an odin-expression form an ordered set, where the order of the values is the order specified in the odin-expression. Therefore, the odin-expressions:

```
prog.c +lib=dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are not equivalent. If multiple copies of the same parameter value appear, only the first of the multiple copies is kept. For example, the odin-expressions:

```
prog.c +lib=socket +lib=dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are equivalent. If a parameter has a value that is a sequence, that value is only considered the same as another identical sequence. For example, the odin-expressions:

```
prog.c +lib=socket dbm +lib=socket :exe
prog.c +lib=socket +lib=dbm :exe
```

are not equivalent, while the odin-expressions:

```
prog.c +lib=socket dbm +lib=socket +lib=socket dbm :exe
prog.c +lib=socket dbm +lib=socket :exe
```

are equivalent.

The parameter value lists of each parameter type are disjoint, therefore, the order of parameters of different types is not significant. For example, the odin-expressions:

```
prog.c +debug +profile :exe
prog.c +profile +debug :exe
```

are equivalent.

Parameters of a list object are inherited by all the elements of the list. For example, if the odin-expression:

```
prog.c.sm :list
```

named a list whose elements were `main.c` and `subrs.c`, then the odin-expression:

```
prog.c.sm :list +debug
```

associates the `+debug` parameter with `main.c` and `subrs.c`.

Parameters are not inherited forward or backward through explicit derivations. Therefore, in the odin-expression:

```
prog.c +debug :o +optimize=2 :exe
```

the `:o` derivation has debugging, but no optimization, while the `:exe` derivation has optimization, but no debugging.

Compare that to the similar expression in which the `:o` is implicit:

```
prog.c +debug +optimize=2 :exe
```

both parameters apply to :exe and all implicit steps (e.g. `:o`) in between, so prog.c is both compiled and linked with debugging and optimization.

Frequently, several odin-expressions share a common set of parameters. To support this kind of sharing, a parameterization expression can take the form of a plus-sign followed by a parenthesized object-expression. The common set of parameters are then stored in the file named by the object-expression. For example, if the file `my.prms` contained the text:

```
+debug +lib_sp=(/local/lib) +lib=socket +lib=dbm +gnu
```

then the odin-expressions:

```
prog.c +(my.prms) :exe
prog.c +debug +lib_sp=(/local/lib) +lib=socket +lib=dbm +gnu :exe
```

would be equivalent.

2.2.5 String Expressions

A string expression, indicated by the equal-sign operator followed by an identifier, is used to specify arguments to tools that are not filenames. For example, the odin-expression:

```
=cxv
```

names the string `cxv`. For example in the odin-expression:

```
prog.c +lib=(mylib.a) +define=FRED:exe
```

`mylib.a` refers to the file of that name in the current directory while `FRED` is a string.

2.3 Virtual Targets

In addition to targets, an `Odinfile` can contain *virtual targets* that define a set of *virtual files*. A virtual target is like a target, except that the value is not copied into a host file, but is simply associated with the specified virtual filename. Virtual files provide a mechanism for specifying aliases for complicated odin-expressions, and a mechanism for specifying build-support files without cluttering host directories.

A virtual target definition is the same as a target definition (see Section 2.1), except that the filename begins with an unescaped percent-sign. For example, the `Odinfile` entry:

```
%prog == prog.c +debug :exe
```

binds the odin-expression `prog.c+debug:exe` to the virtual filename `prog`.

The syntax for selecting a virtual file from an `Odinfile` is identical to that for selecting a file from a directory, except that a percent-sign is used instead of a slash. For example, the odin-expression:

```
src/Odinfile%prog
```

selects the virtual file `prog` from `src/Odinfile`.

The explicit `/Odinfile` above may be omitted. For example, the odin-expressions:

```
src/Odinfile%prog
src/%prog
src%prog
```

are equivalent, as are:

```
./Odinfile%prog
./%prog
.%prog
%prog
```

which all reference the current working directory.

Although only a file named `Odinfile` can specify targets, any file can contain virtual targets. For example, if `src/models` were a host file containing the text:

```
%fastprog == prog.c +optimize=2 :exe
```

then the odin-expression:

```
src/models%fastprog
```

obtains the virtual target for `%fastprog` from `src/models`.

2.4 Text Targets

The value of a target can be specified directly as lines of text (a *here document*), instead of as an odin-expression. In this case, the value declaration consists of two left-angle-brackets optionally followed by an arbitrary tag identifier. For example, the `Odinfile` entry:

```
prog.c.sm == << END
    main.c
    routines.c
END
```

declares `prog.c.sm` to be a text target. The value of `prog.c.sm` is then a file containing the text:

```
    main.c
    routines.c
```

If the tag identifier is omitted, the text value ends at the first line containing only whitespace characters. For example, the `Odinfile` entry:

```
prog.c.sm == <<
    main.c
    routines.c
```

is equivalent to the previous definition.

If the tag identifier begins with a new-line character, the terminating tag identifier does not contain the leading new-line character, and the last new-line character in the text value is ignored. For example, the `Odinfile` entry:

```
vowels == << \\
TAG
aeiou
TAG
```

defines the contents of the file `vowels` to be the five characters: "`aeiou`".

2.4.1 Executable Text Targets

Execute permission can be set for a text target by specifying it to be an *executable text target*. A text target is an executable text target if an exclamation-point is specified immediately before the `<<`. For example, the `Odinfile` entry:

```
%backup == !<<
   backup_name=$1.back
   mv $1 $backup_name
   cp $backup_name $1
```

specifies `%backup` to be an executable text target.

2.5 Command Targets

A target is a *command target* if an exclamation-point is specified following the filename. For example, the `Odinfile` entry:

```
prog ! == prog.c.sm +debug :exe
```

specifies `prog` to be a command target.

Normally, when a filename is specified as an argument to the `odin` host-command, Odin simply ensures that the value of the filename is up-to-date. If the filename is a command target, Odin also executes the file after it is made up-to-date. For example, with the preceding target definition for `prog`, the host-command:

```
odin prog
```

makes sure `prog` is up-to-date with respect to `prog.c.sm+debug:exe`, and then executes it.

2.6 Nested Target Definitions

A file containing additional target definitions can be specified in a *nested target definition*. A nested target definition consists of two equal-signs followed by an odin-expression. A common use of nested target definitions is to pre-process the text of the target definitions. For example, the `Odinfile` entries:

```
== %nested :cpp

%nested == << EOF
#ifdef linux
   prog == prog.c +gnu +debug :exe
#else
   prog == prog.c +debug :exe
#endif
EOF
```

specify that the `+gnu` parameter should be used on the `linux` platform.

Chapter 3

Getting Started

To get started using Odin, go to a directory containing your favorite "hello world" program.

```
1% cd /u/fred
2% cat hello.c

main()
{ printf("Hello World.\n"); return 0; }
```

Then create an `Odinfile` containing the line `hello==hello.c:exe`.

```
3% ls
Odinfile   hello.c
4% cat Odinfile

hello == hello.c :exe
```

You are now ready to build `hello`. Just say:

```
5% odin hello
```

By default, Odin issues a message every time it invokes a tool, for example:

```
scan_for_includes hello.c
cc -c hello.c
cc hello.o
** Copying up-to-date value into /u/fred/hello
```

The first tool scanned the source file for includes; the second generated object code; the third generated an executable; and the fourth copied the executable into the file named `/u/fred/hello`.

You have now built your first executable with Odin. Now to test it:

```
%6 hello
Hello World.
```

3.1 Systems with Several Files

Interesting systems will usually consist of more than one file. In Odin, you list the source files that make up the system in a file called a *system model*. The files named in `#include` statements should not be listed in the system model – Odin will discover them by running a tool that scans the source files looking for `#include` statements.

Modify `hello.c` so that it calls a `print()` routine in the file `print.c`.

```
7% ls
Odinfile   hello    hello.c   print.c
8% cat hello.c

main()
  { print("Hello "); print("World.\n"); return 0; }

9% cat print.c

print(s) char *s;
  { printf(s); }
```

Now you need to create a system model, `hello.c.sm`, for the new `hello` system. The system model can be a regular file, but it can equally well be specifed as a *virtual file* in the `Odinfile`.

```
10% cat Odinfile

hello == %hello.c.sm :exe

%hello.c.sm == << EOF
   hello.c; print.c
EOF
```

The new version of `hello` can now be built:

```
11% odin hello
```

You will notice although the file `hello` has changed, no new files have appeared in the source directory.

```
12% ls
Odinfile   hello   hello.c   print.c
```

This is because all derived files are stored in the derived file cache directory (`$HOME/.ODIN` by default – a non-default location can be specified in the `$ODIN` environment variable).

But the derived files are available for re-use. For example, if you to add a comment to the `print()` routine:

```
13% cat print.c

/* print the character sequence s to standard output */
print(s) char *s;
  { printf(s); }
```

and then ask Odin for `hello` again:

```
14% odin hello
```

you will notice a couple of things. First, the previous results of scanning `hello.c` for includes and compiling it are both re-used, as you would expect. Somewhat more surprising may be that Odin notices that the compilation of the modified `print.c` produced the same object code. Odin then skips the link phase since the old executable is still valid.

3.2 Files in Other Directories

The files in a system model can be in any directory, using either absolute or relative pathnames. Odin knows that `/u/fred/print.c` is potentially different from `/u/jane/print.c` so will not be confused if you switch from one to the other in your system model.

```
15% cat ../jane/print.c

print(s) char *s;
  { printf("*%s*", s); }

%16 cat Odinfile

hello == %hello.c.sm :exe

%hello.c.sm == << EOF
   hello.c; ../jane/print.c
EOF

%17 odin hello
```

3.3 Recursive Odin Calls

Suppose that `/u/jane/print.c` is modified to include a `y.tab.h` file generated by the `yacc` tool from a file named `print.y`. The fact that `y.tab.h` is generated from `print.y` is indicated in `/u/jane/Odinfile`.

```
18% ls ../jane
Odinfile   print.c     print.y
19% cat ../jane/print.c

#include "y.tab.h"
print(s) char *s;
  { printf("*%s*", s); }

20% cat ../jane/Odinfile

y.tab.h == print.y :h
```

If you were using `make`, you would have to somehow get `make ../jane` invoked before you tried to `make hello`. In Odin, all these dependencies are computed for you, so a simple `odin hello` is sufficient.

```
21% odin hello
```

3.4 Debugging Your Program

Odin uses parameters instead of tool flags to modify the behavior of tools. In particular, `+debug` adds debugging information. Either the entry for `hello` in the `Odinfile` can be modified, or as is done below, a second entry such as `test-hello` can be added.

```
22% cat Odinfile

hello == %hello.c.sm :exe

test-hello == %hello.c.sm +debug :exe

%hello.c.sm == << EOF
   hello.c; print.c
EOF
```

At this point, you can build `test-hello` and run the debugger (the first step is redundant but illustrative).

```
23% odin test-hello

24% odin test-hello!gdb
```

Chapter 4

Odin-Commands

In addition to being used as a means for making source files up-to-date, the Odin system can also be used as a command interpreter for *odin-commands*. Odin-commands are either given as arguments to the `odin` host-command, or, if the `odin` host-command is invoked with no arguments, odin-commands are read interactively from standard input.

When Odin is used as an interactive command interpreter, it first prints out a banner indicating the version of Odin you are using, and then an arrow indicating it is ready to accept input. You exit the Odin interpreter by typing an end-of-file, commonly the control-D character.

```
1% odin
Odin Version 1.17
-> ^D
2%
```

There are four kinds of odin-commands: *query odin-commands*, *copy odin-commands*, *execute odin-commands*, and *variable assignment odin-commands*. If the Odin interpreter encounters a question-mark in an odin-command, it generates a help message describing what could have been typed at the location of the question-mark.

4.1 Query Odin-Commands

A *query odin-command* consists simply of an odin-expression. A query odin-command causes the Odin object named by the odin-expression to be brought up to date, and reports information concerning the status of the object (such as error and warning messages produced by tool steps that were run to produce the object).

```
-> print.c:exe
--- </u/fred/print.c :exe> generated errors ---
ld: Undefined symbol
   _main
cc failed
->
```

The level of detail of this information is controlled by the value of the `ErrLevel` and `WarnLevel` variables (see Section 4.4.5).

4.1.1 Status of Odin Objects

Associated with each Odin object is a status level, where the status level is one of `OK`, `WARNING`, `ERROR`, `CIRCULAR`, `NOFILE`, and `ABORT`. `OK` is the maximum status level and `ABORT` the minimum.

The status of a given derived object depends on the results of the tools that produced 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 an object that was needed to create the given object is the object itself, the status level of the given object is at most `CIRCULAR`. 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` or less, then the status level of the given object is set to be `ABORT`.

The status of a source object is `NOFILE` if the host file does not exist, the status of the value of its target value (see Section 2.1) if it is a target, and otherwise `OK`.

4.1.2 Error and Warning Messages

The warning or error messages produced by all tool invocations are saved by the Odin system. 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 is:

```
Unsatisfied external reference: "proc1".
```

An example of a warning message from a loader is:

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

A text file containing a summary of all error messages for an object can be obtained by applying the `:err` derivation to the object. For example, the file named by the odin-expression:

```
prog.c :exe :err
```

contains a summary of all error messages produced by any tool used in the generation of the `prog.c:exe` object. The `:warn` derivation produces a text file containing both warning and error messages for an object.

4.2 Copy Odin-Commands

A *copy odin-command* copies the contents of a specified Odin object into another object (the destination object). The copy is performed only if the status level of the specified object is no lower than `WARNING`. The destination object must be a source object since only source objects can be directly modified by a user.

There are two forms of the copy odin-command: *copy-to*, indicated by a right-angle-bracket, and *copy-from*, indicated by a left-angle-bracket. Examples of these two odin-commands are:

```
-> prog.c +debug :exe > prog
-> prog < prog.c +debug :exe
```

If the destination object is a directory, the *label* of the specified object is used to name the new copy. The label of a source file is the last component of the pathname of the source file. The label of a derived object is *source-label.type-name* where *type-name* is the name of the output type of the tool that produced it (see Section B.7). and where *source-label* is the label of the source file from which it is derived. For example, the label of `/usr/src/prog.c` is `prog.c` and the label of `/usr/src/prog.c:exe` is `prog.c.exe`.

If a list is copied into a directory, each element of the list is copied individually into the directory.

4.2.1 Copying to Standard Output

If the destination object is omitted from a copy-to odin-command, the specified object is displayed on the current standard output device. For example, the odin-command:

```
-> prog.c >
```

displays the file named `prog.c`. Note that if d1 is a directory:

```
-> d1 >
```

is short-hand for:

```
-> d1/Odinfile >
```

4.2.2 Editing with the Copy Odin-Command

If only the destination object is specified in a copy-from odin-command, the specified object is given to the host-system editor indicated by the `$EDITOR` environment variable, with the `vi` editor the default. For example, if the value of the `$EDITOR` variable is `emacs`, then the odin-command:

```
-> prog.c <
```

invokes the `emacs` editor on the file `prog.c`.

4.3 Execute Odin-Commands

An *execute odin-command* consists of an odin-expression followed by an exclamation-point and a host-command line, where either the odin-expression or the host-command can be omitted. Examples of execute odin-commands (with output omitted) are:

```
-> prog.c +debug :stdout !more -s
-> ! ls *.c
-> prog.c :exe !
```

The result of the execute odin-command is to make the Odin object up-to-date, append its filename to the host-command line, and give the resulting extended host-command line to the host system for execution.

The exclamation-point has the special lexical property that if the first non-whitespace character following it is not a colon, a semi-colon, or an equal-sign, then the rest of the line is treated as a single escaped sequence of characters. This avoids the confusion resulting from interactions between host-command and Odin character escape conventions. A leading colon, equal-sign, or whitespace character can be included in the escaped sequence of characters by preceding it with a backslash.

If the host-command is omitted, the Odin object itself is executed. If execute permission is set for the Odin object, it is given to the host operating system for execution; otherwise, the Odin object is assumed to contain odin-commands that are to be executed by the Odin interpreter.

4.4 Variable Assignment Odin-Commands

The behavior of the Odin interpreter can be modified by changing the value of an *Odin variable*. The functions affected by Odin variables are the current working directory, the distributed parallel build facility, the help facility, the error and log facility, the file change notification facility, and the maximum total file system space used by derived objects.

A variable assignment odin-command consists of the name of an Odin variable followed by an equal-sign operator and an odin-expression. For example, the odin-commands:

```
-> dir = ../src
-> warnlevel = 4
```

assign the value `../src` to the `Dir` variable and the value `4` to the `WarnLevel` variable (Odin variable names are case-insensitive).

If the value is omitted from a variable assignment odin-command, Odin displays the current value of the specified variable. For example, after the preceding odin-commands, the odin-command:

```
-> warnlevel =
4
```

can be used to find out the current value of `WarnLevel`.

The Odin variables and their default values are:

```
Dir ........... odin_invocation_directory
MaxBuilds ..... 2
BuildHosts .... LOCAL : LOCAL
Size .......... 0
KeepGoing ..... yes
History ....... yes
LogLevel ...... 2
ErrLevel ...... 3
WarnLevel ..... 2
HelpLevel ..... 1
VerifyLevel ... 2
```

An initial value for an Odin variable can be specified in an environment variable whose name is the the Odin variable name in capital letters preceded by the string `ODIN`. For example, the initial value for `MaxBuilds` is specified in the `ODINMAXBUILDS` environment variable.

4.4.1 Dir

The current working directory can be changed by assigning a new value to the `Dir` variable. The value of the current working directory is especially significant for Odin, since Odin identifies source objects by their absolute pathname, and the current working directory provides the absolute pathname for all relative names given to the Odin interpreter.

4.4.2 BuildHosts, MaxBuilds

The `BuildHosts` variable specifies the list of hosts that are used to execute the tools that generate the derived objects. A tool is executed on the first entry in the `BuildHosts` list that does not have a currently executing tool. The name `LOCAL` refers to the local host. The `MaxBuilds` variable specifies the maximum number of tools to execute in parallel.

The hosts in `BuildHosts` must have the same machine architecture and file namespace as the local host. If builds on non-equivalent machines are desired, this can be achieved by extending the tool packages (see Section B.1).

4.4.3 KeepGoing

When a build step reports errors, Odin will continue with build steps that do not depend on the failed build step. Setting the value of the `KeepGoing` variable to `no` will cause Odin to terminate the build when any build step reports an error (similar to the default behavior of Make).

4.4.4 History

The `History` variable specifies whether emacs-like history and command line editing is supported by Odin when it is used as an interactive command interpreter.

4.4.5 ErrLevel, WarnLevel, LogLevel

When an odin-command is executed, Odin indicates any errors or warnings associated with the odin-expressions specified in that odin-command. The `ErrLevel` and `WarnLevel` variables specify how detailed

this report is. In particular, the user can choose whether to see final status information, to see messages incrementally as they are produced by tools steps, or to see a summary of all relevant messages (including those from previously cached tool steps).

Odin can also produce a variety of information about the activities it is performing, such as a brief description of each tool that is invoked to satisfy a given request. The `LogLevel` variable specifies how detailed these messages are.

4.4.6 HelpLevel

The `HelpLevel` variable specifies what degree of detail is provided when the user asks for a list of possible file or parameter types (see Section 4.5). Normally, only commonly used types are described, but the `HelpLevel` can be increased to have all possible types described.

4.4.7 VerifyLevel

By default, Odin checks the modification dates of all relevant source files at the beginning of an Odin session and before each interactive odin-command. With the `VerifyLevel` variable set to `1` Odin will use Linux inotify to track changes. With the `VerifyLevel` variable set to `2` Odin expects all source file changes to be explicitly indicated via the `filename!:test` odin-command,

4.4.8 Size

The value of the `Size` variable indicates how much disk space (in kilobytes) is currently being used by derived files.

4.4.9 Environment Variables

Environment variables can be used in odin-commands. For example, if the environment variable `$HOME` has the value `/u/geoff`, then the two odin-commands:

```
-> $HOME/prog.c :exe
-> /u/geoff/prog.c :exe
```

are equivalent. The value of an environment variable can be quoted by immediately preceding it with a quoted identifier. For example, if the value of `$PATH` is `/bin:/usr/bin`, then the two odin-commands:

```
-> prog.c +path='/etc:'$PATH
-> prog.c +path='/etc:/bin:/usr/bin'
```

are equivalent.

The environment variable `$ODINCACHE` is set by Odin to be the location of the cache directory (see Section 1.2.2).

An environment variable is given a new value with a variable assignment odin-command of the form:

```
Variable = !Value
```

For example, the odin-command:

```
-> HOME = !/u/clemm
```

sets the value of the environment variable `$HOME` to be `/u/clemm`.

Although any environment variable can be used in odin-commands, only environment variables declared in a tool package (see Section B.6) can be used in `Odinfile` target definitions (see Section 2.1) The value of a declared environment variable is obtained from the environment at the time the current cache was created or reset. If the variable was not set in that environment, the default value specified in the derivation graph declaration of the variable is used.

The expressions `~` and `~name` are treated as if they were environment variables, bound respectively to the login directory of the current user and the login directory of the user with login `name`.

4.5 Help

A simple context-sensitive help facility is provided to describe the syntax of odin-commands and the currently available object types and parameter types. If a user types a question-mark anywhere in an odin-command, Odin provides a description of what could appear at that location in the odin-command. For example, the following odin-command provides a syntax summary:

```
-> ?
Control-D
OdinExpr
   HostFile
      + ParameterType
      + ParameterType = Identifier
      + ParameterType = (OdinExpr)
      : FileType
      / Key
      % Key
OdinExpr > OdinExpr
OdinExpr >
OdinExpr < OdinExpr
OdinExpr <
OdinExpr !HostCommand
OdinExpr !
OdinExpr !:test
!HostCommand
!:test
Variable =
Variable = Value
Variable = !Value
```

4.5.1 Source Type Help

If a list of the declared source object type-names is desired, a question-mark can be put in place of the extension for a file. For example, the odin-command:

```
-> prog?
Possible Base Types :
.c ............. C source code
.f ............. Fortran77 source code
.c.sm .......... C system model containing a list of filenames
,v ............. RCS version control file
```

describes the currently recognized type suffixes, where the message contents is derived from the installed tool packages (see Section B.3).

4.5.2 Derivation Help

If a list of possible derivations is desired, a question-mark can be put in place of the derivation name, and the Odin system responds with a list of the possible object types that can appear at that position. For example, the odin-command:

```
-> prog.c :fmt : ?
Possible Derivations from an Object of Type "fmt":
o .......... object code
```

```
exe ........ executable binary
fmt ........ formatted version
xref ....... cross reference listing
```

states that all of the following are acceptable odin-expressions:

```
prog.c :fmt :o
prog.c :fmt :exe
prog.c :fmt :fmt
prog.c :fmt :xref
```

4.5.3 Parameterization Help

If a list of the possible parameter types is desired, a question-mark can be put in place of the parameter, and the Odin system responds with a list of the possible parameter types that can appear at that position. For example, the odin-command:

```
-> prog.c :fmt + ?
Possible Parameters:
lib ........ object code library
debug ...... flag to generate debugging information
```

states that any of the following are acceptable odin-expressions:

```
prog.c :fmt +lib=(/usr/lib/network.a)
prog.c :fmt +debug
prog.c :fmt +debug +lib=(/usr/lib/network.a)
```

A more exact form of parameter help can be specified by indicating which derivation you intend to apply to the parameterized object. For example, the odin-command:

```
-> prog.c :fmt + ? :o
Possible Parameters:
debug ...... flag to generate debugging information
```

states that the following is an acceptable odin-expression:

```
prog.c :fmt +debug :o
```

Since the `+lib` parameter is not relevant to the derivation from `:fmt` to `:o`, it is not listed.

4.5.4 Variable Help

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

```
-> ? =
Dir MaxBuilds BuildHosts Size KeepGoing History
LogLevel ErrLevel WarnLevel HelpLevel VerifyLevel
```

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

```
Variable = ?
```

For example, the odin-command:

```
-> LogLevel = ?
0: No log information is generated.
1: Build commands are echoed.
2: And Odin commands.
3: And names of objects with errors.
4: And names of objects generated by tool scripts.
5: And names of objects generated by internal tools.
6: And names of objects deleted.
7: And names of objects touched by broadcast.
```

describes the possible values of the `LogLevel` variable.

Appendix A

Dependency Database Utilities

Odin provides the following utilities for browsing and modifying the dependency database: `:test`, `:redo`, `:inputs`, `:outputs`, `:elements`, `:element-of`, and `:dpath`. These utilities are specified following the exclamation point of an execute odin-command. For example, the odin-command:

```
-> prog.c :o ! :redo
```

would invoke the `:redo` utility with `prog.c:o` as its argument.

Unlike objects named in all other kinds of odin-commands, the object specified as an argument to a database utility is *not* made up-to-date before the database utility is executed. If you wish the database utility to be executed on an up-to-date object, first give the object as an argument to a query odin-command. For example:

```
-> prog.c :o
-> prog.c :o ! :redo
```

A.1 Testing Host Files for Changes

Normally Odin checks all host files for changes at the beginning of an Odin session and before each interactive odin-command (Odin actually performs these checks lazily, and will `fstat()` a file only when it is needed as input to a tool). It also checks a particular host file for changes after it is used as the destination of a copy odin-command.

Odin's runtime efficiency can be significantly improved by setting the Odin `VerifyLevel` variable to `1` (see Section 4.4.7). In this mode, the user (or a tool such as the user's editor) must notify Odin with the `:test` utility whenever a file changes other than through a copy odin-command.

For example, the odin-command:

```
-> /usr/src/prog.c ! :test
```

causes Odin to check to see if `/usr/src/prog.c` has changed, to determine whether objects derived from this file must be recomputed. If no source is specified for the `:test` utility, i.e. the odin-command is just:

```
-> ! :test
```

then all source files known to Odin are checked.

The Odin distribution includes an `odin.el` file that extends the gnu-emacs editor to issue the appropriate *filename*`!:test` message whenever a file is saved.

A.2 Redoing a Particular Derivation Step

The `:redo` utility allows the user to tell Odin that a particular step in a derivation should be recomputed when it is next requested, even if that step has already been computed and inputs to that step have not changed. For example, the odin-commands:

```
-> prog.c +debug :o ! :redo
-> prog.c +debug :o
```

causes Odin to redo the `prog.c+debug:o` step. This utility is useful when a transient error in a derivation step occurs, but the tool that implements the derivation step did not recognize it as a transient error.

A.3 Browsing the Dependency Database

The `:inputs` utility displays the direct inputs to the specified derived object. The `:outputs` utility displays the direct outputs of the specified object. The `:elements` utility displays the elements of the specified list. The `:element-of` utility displays the lists that contain the specified object as an element. For example, the odin-command:

```
-> prog.c +debug :exe !: inputs
```

displays the input objects used to compute the derived object `prog.c+debug:exe`.

The `:dpath` utility displays a dependency path from one object to another (if there is one). Normally, `:dpath` is applied to an object with a trailing `+depend` parameter. For example:

```
prog.c :exe +depend=(/usr/include/stdio.h) ! :dpath
```

would display the dependency path from `/usr/include/stdio.h` to `prog.c:exe`.

When browsing the dependency database, you may encounter certain internal object types that you have not explicitly declared. For example, if you have declared a tool with multiple outputs, there will be an object whose type ends with `*composite` which represents the set of output objects from a single tool run. Another example is a `:abstract` object which is created for each `:PIPE` and `:GENERIC` object.

Appendix B

Tool Package Libraries

In the Odin system, all information about tools, object types, and parameter types is specified in special directories called *tool package libraries*. When a cache (see Section 1.2.2) is created, *tool packages* from the tool package libraries are installed in the cache. A tool package is just a sub-directory of a tool package library, where the name of the package is the name of the sub-directory. The list of tool packages and the order in which they should be loaded is specified in a file named `PKGLST` (one package name per line) in the tool package library directory.

A default tool package library that is provided with the Odin distribution is installed into every cache. This default library is commonly located in `/usr/local/lib/Odin`. Do not add, delete, or modify packages from this library. This ensures that installation of newer versions of Odin will not delete or overwrite any local modifications.

A user can have the packages from additional libraries installed by specifying these libraries (separated by colons) in the `$ODINPATH` environment variable. For example, if a user sets the `$ODINPATH` variable to have the value `/project/lib/Odin:/home/geoff/my-pkg-lib`, then the packages from `/project/lib/Odin`, from `/home/geoff/my-pkg-lib`, and from the default library are installed into every cache the user creates. If a package with a given name occurs in several package libraries, the package from the library that occurs earliest in `$ODINPATH` is used.

The entries in `$ODINPATH` can also be individual tool packages rather than tool package libraries, in which case those packages are added to the list of packages to be installed into the cache.

B.1 Tool Packages

A tool package is a directory that contains a *derivation graph* file and any support files that implement the tools declared in the derivation graph. If `pkgx` is the name of the tool package, then `pkgx.dg` is the name of the derivation graph file.

The derivation graph is the mechanism for declaring object types, parameter types, environment varables, and tools for the tool package. It is described in detail in Section B.2.

B.1.1 Tool Package Selection

The `odin -R` command installs tool packages into the cache that it creates, listing on standard output the packages that it installs. By default, `odin -R` loads packages from odin's install directory e.g. `/usr/local/lib/Odin`. You can load packages from other directories using the `ODINPKG` environment variable.

`ODINPKG` is a colon-separated list of directories to search, and odin looks for a `PKGLST` file in each of these in turn. PKGLST lists, one per line, the packages to load from that directory.

Odin does not load "duplicate" packages, eg where a `PKGLST` file specifies package X, but package X has already been loaded via a `PKGLST` from a directory earlier in the `ODINPKG` list, the latter X is ignored. You can therefor "replace" standard odin packages with your own package of the same name.

Note that although you cannot tell odin not to load a particular standard package, you can achieve the affect by creating your own package of the same name with an empty .dg file.

Note that the standard Odin packages are not detailed in this manual, even though the tutorial makes use of them. Browse the .dg files to see the capabilities of the standard odin packages. Also look for their use in others' Odinfiles (including in the odin distribution).

B.1.2 Versioned Tool Packages

By default, a tool package is symbolically linked into each cache in which it is installed. Since the Odin system maintains the dependency of each build step on the package scripts used by that build step, a change to a build script will cause the recomputation of all build steps that use that script. In case the script change is an enhancement rather than a bug fix, it is often preferable to allow each user to decide when to upgrade his cache to use the new packages.

To provide this functionality, a tool package library can contain a file named `LIBVER` in addition to the tool package subdirectories. The `LIBVER` file contains text of the form `PKGVER.i`, where `i` is an integer. Each tool package then contains a file named `PKGVER.i` and a set of sub-directories, with one sub-directory for each version of the tool package. If changes to a package are made in the form of a new version, these changes are not seen by a user until he resets his cache with the `-R` option to the `odin` host-command.

The tool package version installed into a cache is the one specifed by the `PKGVER.i` file. For example, if `LIBVER` contained the text `PKGVER.7`, and if a tool package directory named `pkgx` contains three tool version directories named `pkgx/1.0`, `pkgx/1.1-beta`, and `pkgx/1.2`, and if the contents of the `pkgx/PKGVER.7` file is the text `1.1-beta`, then any new cache that installs `pkgx` installs the `1.1-beta` version of `pkgx`.

When a new version of the library is installed, the `i` in `PKGVER.i` is incremented, e.g. to `PKGVER.8`, and a `PKGVER.8` file is created in each of the package subdirectories. The library can be reverted back to an earlier version of the packages by editing the `LIBVER` file.

B.2 Derivation Graphs

A derivation graph consists of a sequence of source, object type, parameter type, environment variable, and tool declarations. A source declaration associates a filename pattern with an object type. An object type declaration associates an object type with a set of supertypes. A parameter type declaration associates a parameter type with an object type for the parameter values. An environment variable declaration specifies a default value for an environment variable. A tool declaration specifies the inputs and outputs of the tool, where the inputs are object types, parameter types, odin-expressions, and identifiers, and where the outputs are object types.

In the derivation graph, the following characters are special:

```
<whitespace> # \ ' : + = ( ) / % ; ? * < > @ & $
```

The quoting and commenting conventions for the derivation graph are the same as those for odin-expressions (see Section 2.2).

B.3 Source Declarations

A source declaration specifies the type for a host file based on the filename of the host file. It consists of a *filename expression*, a right arrow (composed from an equal-sign and a right-angle-bracket), and the name of a declared object type. A filename expression consists either of an identifier followed by an asterisk (i.e. all filenames that begin with that identifier), an asterisk followed by an identifier (i.e. all filenames that end with that identifier), or just an asterisk (i.e. all filenames). For example, the derivation graph entries:

```
*.c   => :c;
*.c.c => :cplusplus;
s.*   => :sccs;
```

```
*     => :FILE;
```

declare that the object type of a filename ending with `.c` is `:c`, the object type of a filename ending with `.c.c` is `:cplusplus`, the object type of a filename beginning with `s.` is `:sccs`, and the object type of any filename is `:FILE`.

The following disambiguation rules are used to ensure that every host file has a single object type: if a host filename matches both a prefix filename expression and a suffix filename expression, then the host file is an instance of the prefix filename expression; if a host filename matches two different prefix filename expressions or two different suffix filename expressions, then the host file is an instance of the longer filename expression. For example, with the previous source declarations, the following object types would be assigned:

```
FILENAME           OBJECT-TYPE
src/test.c         c
src/test.c.c       cplusplus
src/s.test.c.c     sccs
src/test           FILE
```

B.4 Object Type Declarations

An object type declaration consists of the object type being declared, a *help identifier*, a right arrow, and the *direct supertypes* of the type being declared. For example, the derivation graph entries:

```
:c 'C source code'? => :source;
:source 'source code' => :FILE;
:fmt 'formatted source code' => :FILE;
```

declare that the direct supertype of `:c` is `:source`, the direct supertype of `:source` is `:FILE`, and the direct supertype of `:fmt` is `:FILE`.

The *help identifier* is used by the Odin help system to generate messages about the installed types (see Section 4.5). The help identifier is followed by a question-mark to indicate that it should be displayed in help messages. The user can explictly request to see all help identifiers by increasing the value of the Odin `HelpLevel` variable (see Section 4.4.6).

An object type can be declared with multiple direct supertypes, either in a single declaration or in separate declarations. For example, the derivation graph entries:

```
:fmt.c 'formatted version of C code' => :c;
:fmt.c 'formatted version of C code' => :fmt;
```

and

```
:fmt.c 'formatted version of C code' => :c :fmt;
```

are equivalent.

The *supertypes* of an object type are the transitive closure of the direct supertypes (i.e. the supertypes are the direct supertypes and all supertypes of the direct supertypes). For example, from the preceding object type declarations, the supertypes of `:fmt.c` are `:c`, `:fmt`, `:source`, and `:FILE`.

The supertypes of an object type help determine what objects can be derived from an object of that type. In particular, if the type of the object `obj` is `:o`, then the odin-expression `obj:deriv` is valid if `:deriv` is a supertype of `:o`, if `:deriv` is the output of a tool whose inputs can be derived from `:o`, or if `:deriv` is a supertype of an object that can be derived from `:o`.

In some rare circumstances, a tool produces an object of a specified type, but this object should not be used to provide this type as an intermediate step in derivations. In this case, the output type of the tool should be declared to be a `base subtype` of the specified type. This is indicated by placing angle brackets around the supertype. For example, the derivation graph entry:

```
:dummy_main.c 'dummy main program for a C executable' => <:c>;
```

declares that the direct base supertype of `:dummy_main.c` is `:c`. This states that derivations appropriate to objects of type `:c` can be applied to objects of type `:dummy_main.c`, but that a derived object of type `:dummy_main.c` cannot be used for a derivation that must produce an object of type `:c`.

B.4.1 Built-In Supertypes

The following object type declarations are built into the Odin system:

```
:OBJECT 'any object'                                          => :OBJECT;
:FILE 'a file'                                                => :OBJECT;
:LIST 'a list'                                                => :OBJECT;
:VOID 'an object with no value'                               => :OBJECT;
:REFERENCE 'a reference'                                      => :OBJECT;
:EXECUTABLE 'a file that should be executed when queried' => :FILE;
:GENERIC 'a file whose type is based on its label'        => :FILE;
:PIPE 'a file produced by a type-preserving tool'         => :FILE;
:DERIVED-DIRECTORY 'a derived directory'                  => :FILE;
```

Every object type other than the built-in object types must be a subtype of exactly one of `:FILE`, `:LIST`, or `:VOID`.

If the type of an object is a subtype of `:VOID`, the the object does not have a value. These types are useful for declaring tools that do not generate an output file, but do have side effects such as printing a file or writing a tape.

If the type of an object is a subtype of `:REFERENCE`, then the object refers to another object. Whenever a reference object is used, such as when it is displayed or when it is given as input to a tool, it is automatically dereferenced by Odin.

If the type of an object is a subtype of `:EXECUTABLE`, when the object is specified in a query odin-command (see Section 4.1), the object is automatically executed after it has been made up-to-date. Executable types are analagous to executable targets (see Section 2.4.1).

If the type of an object is a subtype of `:GENERIC`, then the object is a derived file whose type is based on the root of the source file from which it is derived (see Section 2.2.2). A generic type is commonly produced as the result of a checkout operation from a source control file or as the result of an uncompress operation on a compressed file. For example, if the root of `prog.c.sm,v` is `prog.c.sm`, if the type of a file named `prog.c.sm` is `:c.sm`, and if `:checkout` is declared in the derivation graph entry:

```
:checkout 'version checked out of a version control file' => :GENERIC;
```

then the type of `prog.c.sm,v :checkout` is `:c.sm`.

If the type of an object is a subtype of `:PIPE`, then the object is a derived file whose type is the same as the input object to the tool that produced it. For example, if `:stdout` is declared in the derivation graph entry:

```
:stdout 'standard output file of a tool run' => :PIPE;
```

and if the type of `prog.c` is `:c`, then the type of `prog.c +cmd='sed s/a/b/' :stdout` is also `:c`.

If the type of an object is a subtype of `:DERIVED-DIRECTORY`, then the object must be a directory produced by a tool.

B.4.2 Built-In Derived Object Types

```
:err          'errors'                              => :FILE;
:warn         'warnings'                            => :FILE;
:targets      'targets in an Odinfile'              => :FILE;
:name         'name of an object'                   => :FILE;
:names        'names of the elements of a list'     => :FILE;
```

```
:filename     'filename of a file'                  => :FILE;
:ls           'filenames of the elements of a list' => :FILE;
:cat          'concatenation of the contents of the elements of a list'
                                                    => :FILE;
:first        'first element of a list'             => :OBJECT;
:union        'list union'                          => :LIST;
:id           'unique numeric id of an object'      => :FILE;
:label        'label of an object'                  => :FILE;
:labels       'labels of the elements of a list'    => :FILE;
:dir          'directory containing a file'         => :REFERENCE;
:vir_dir      'directory containing virtual files'  => :DERIVED-DIRECTORY;
:depend       'source dependencies'                 => :LIST;
:source_names 'source components of a system model' => :FILE;
:operation_names 'operation components of a system model'
                                                    => :FILE;
:view         'view from a view-path specification' => :LIST;
:expand_hooks 'expand hooks'                        => :FILE;
:apply        'apply operations'                    => :REFERENCE;
:odin_help    'top level help for odin'             => :EXECUTABLE;
```

The `:err` derivation produces a text file from an arbitrary input object. This text file contains all error messages generated by any tool used in the process of creating the input object. The `:warn` derivation produces a text file that contains both warning and error messages.

An `Odinfile` can contain nested target definitions (see Section 2.6). The `:targets` derivation produces a flattened version of the `Odinfile` containing all target definitions.

The `:name` derivation produces a text file that contains an odin-expression that specifies the input object. The `:names` derivation produces a text file from a list, where the text file contains a sequence of odin-expressions corresponding to the elements of the input object.

The `:filename` derivation produces a text file from a file, where the text file contains the host filename of the input object. The `:ls` derivation produces a text file from a list, where the text file contains a sequence of host filenames corresponding to the elements of the input object. The filename of a string object is the identifier that names the string object.

The `:cat` derivation produces a text file from a list, where the text file contains the concatenation of the contents of the elements of the input object. The contents of a string object is the identifier that names the string object.

The `:first` derivation produces an object from a list, by selecting the first element of the list. This derivations is primarily used in parameter type declarations.

The `:union` derivation produces a list from a list, where each element in the input list appears exactly once in the output list.

The `:id` derivation generates a text file containing a unique integer for the input object. The `:label` derivation generates a text file containing the label of the input object. The `:labels` derivation produces a text file from a list, where the text file contains a sequence of labels corresponding to the elements of the input object.

The `:dir` derivation produces a reference object that refers to the directory containing the input object. If the input object is a derived object, then `:dir` produces the directory that contains the source object from which the input object is derived.

The `:vir_dir` derivation produces a derived directory containing all the virtual targets defined in the input file. For the syntax of a virtual target definition, see Section 2.3.

The `:depend` derivation produces a list from an arbitrary input object. This list contains all source objects used in the process of creating the input object.

The `:source_names` derivation produces a text file from a text file. The format of the input text file is a sequence of odin-expressions naming file objects. The output text file contains just the initial source

filename component of each odin-expression (i.e. the first derivation or parameterization operator and all subsequent operations are removed). An example of such an input file would be:

```
../prog.c :fmt +debug
test.c +inc_sp=(..)
```

The result of applying `:source_names` to this input file would be:

```
../prog.c
test.c
```

The `:operation_names` derivation produces a text file from a text file. The format of the input text file is a sequence of odin-expressions naming file objects, one odin-expression per line. The output text file contains any text following the initial source filename component of each odin-expression (i.e. everything preceding the first derivation or parameterization operator is removed). An example of such an input file would be:

```
../prog.c :fmt +debug
test.c +inc_sp=(..)
```

The result of applying `:operation_names` to this input file would be:

```
:fmt +debug
+inc_sp=(..)
```

The `:view` derivation produces a list from a text file. The format of the text file is sequences of odin-expressions naming file objects, where each sequence is terminated by an odin-expression naming a string object. An example of such an input file would be:

```
# the search sequence for test.h
   test.h
   test.h,v :co
   RCS/test.h,v :co
   = 'test.h'
# the search sequence for sys.h
   sys.h
   SCCS/s.sys.h :co
   = 'sys.h'
```

The `:view` list contains the first file from each sequence whose status is greater than `ERROR`. If none of the files in a particular sequence have the appropriate status, the status of the `:view` object is set to `ERROR`, unless the sequence is terminated by the empty string odin-expression:

```
   = ''
```

in which case that sequence is just ignored. The `:view` derivation depends only on the file selected from a given sequence and on the files preceding the selected file in the given sequence. This means that error messages from these preceding files are included in `:error` reports for the `:view` object, but any derived files following a selected files from a given sequence are ignored when the `:view` object is made up-to-date.

The `:expand_hooks` derivation produces a text file from an input text file by expanding all *hooks* found in the input file. A hook is a way of linking a message back to an appropriate area in a source file that is relevant to understanding the given message.

A hook is text of the form:

```
(| odin-expression | hooktype | hookdata |)
```

where `odin-expression` is the name of the object referenced by the hook, `hooktype` is the type of the hook, and `hookdata` is the actual reference. In order to escape the delimiter '`(|`', '`|`', or '`|)`', precede it with a backslash.

The `:expand_hooks` derivation replaces all constructs of this form with the result of applying `:hooktype` to the object specified by `odin-expression`. The value of `hookdata` is placed in the `+hookvalue` parameter. Hooks appearing in the hookdata string (*nested hooks*) are passed unexpanded to the `:hooktype` tool. The result of the `:hooktype` derivation should be text that references the appropriate location in a source file. This text can in turn contain additional hooks that will be expanded (in particular, this allows for the expansion of nested hooks).

If `:hooktype` cannot be applied to the object specified by `odin-expression`, Odin attempts to apply `:hooktype` to any object from which this object can be derived. If all attempts to apply `:hooktype` fail, the result of hook expansion is the input text with the delimiters around the hook deleted.

The `:apply` derivation produces a reference object from an input object and a `+apply` parameter. The value of the `+apply` parameter is a file containing an arbitrary sequence of selection, derivation, and parameterization expressions, which are applied to the input object by the `:apply` derivation. For example, if the file `apply.expr` contained the text:

```
+debug :output /OUT.DAT
```

the two odin-expressions:

```
prog.c +debug :output /OUT.DAT
prog.c +apply=(apply.expr) :apply
```

would be equivalent.

The `:odin_help` derivation produces an executable file from the current working directory. This file is executed whenever the user requests help with the odin-command

```
-> ?
```

The `:odin_help` derivation can be overridden in a tool package to provide a different response to this help command.

B.5 Parameter Type Declarations

A parameter type declaration consists of the parameter type being declared, a help identifier, a right arrow, and an object type. When a parameter type is used as the input to a tool, the list of values for that parameter type is first derived to the specified object type. For example, the derivation graph entries:

```
+define 'a macro definition'     => :cat;
+lib    'an object code library' => :ls;
+debug  'debug flag'             => :first;
```

declare a `+define` parameter type whose values are `:cat`'ed, a `+lib` parameter type whose values are `:ls`'ed, and a `+debug` parameter type which is a flag.

B.6 Environment Variable Declarations

An environment variable declaration consists of the environment variable being declared, a help identifier, an equal-sign, and either an identifier or a parenthesized odin-expression. The value following the equal-sign specifies the default value for that variable. For example, the derivation graph entries:

```
$CC 'name of the C compiler' = 'gcc';

$CC_HOME 'directory containing the C compiler tools' = '/usr/local/bin';
```

specify default values for `$CC` and `$CC_HOME`. These values can then be overridden by corresponding values in the environment when a particular cache is created or reset. For example, if the value of `$CC_HOME` is defined to be `/usr/public/bin` in the environment of the process that resets a cache, then the value of `$CC_HOME` is `/usr/public/bin` whenever it is used in odin-expressions or odin-commands for that cache.

If the default value is a parenthesized odin-expression, the expression is evaluated and the resulting filename is used as the default value.

The current environment variables for a cache can be found in `$ODINCACHE/ENV`.

B.7 Tool Declarations

A tool declaration consists of a tool name, the tool inputs, a right arrow, and the tool outputs. The tool name is either `EXEC`, `COLLECT`, `READ-LIST`, or `READ-REFERENCE`. A tool input is an identifier or an *input-expression* in parentheses (see Section B.7.1). A tool output is an object type in parentheses.

For example, the derivation graph entry:

```
EXEC (/bin/yacc) -dv (:y) => (:y.tab.c) (:y.tab.h) (:y.output);
```

declares an `EXEC` tool that has inputs `(/bin/yacc)`, `-dv`, and `(:y)`, and that has outputs: `(:y.tab.c)`, `(:y.tab.h)`, and `(:y.output)`.

B.7.1 Input-Expressions

An *input-expression* is just like an odin-expression, except that a *derived input* or a *parameter input* can appear wherever a source filename can appear in an odin-expression. A *derived input* consists of a declared object type (e.g. `:o`), while a *parameter input* consists of a declared parameter type (e.g. `+debug`).

An input-expression can contain references to declared environment variables. For example, the following derivation graph entries declare and reference the environment variable `$YACC_HOME`:

```
$YACC_HOME 'directory containing yacc' = '/bin';

EXEC ($YACC_HOME/yacc) -dv (:y) => (:y.tab.c) (:y.tab.h) (:y.output);
```

An input-expression can contain *second-order derivations*. A second-order derivation consists of a *second-order object type* followed by an equal sign and a derivation expression. For example, the input-expression:

```
:lookup=:o
```

contains the second-order object type `:lookup` whose argument is the derivation `:o`.

The five second-order object types are `:map`, `:recurse`, `:extract`, `:delete`, and `:lookup`. The derivation `:map=:type` takes a list as input and produces a list containing the application of `:type` to each element of the input list. The derivation `:recurse=:type` takes a list as input and produces a list containing the application of `:type :recurse=:type` to each element of the input list to which `:type` can be applied, and just the original element otherwise. The derivation `:recurse=:type` can also take a file as input, in which case the input is treated as if it were a list containing that file as its only member. The derivation `:extract=:type` takes a list as input and produces a list containing each element of the input list that is a subtype of `:type`. The derivation `:delete=:type` takes a list as input and produces a list containing each element of the input list that is not a subtype of `:type`. The derivation `:lookup=:type` takes an object as input and produces a reference to an object which is the application of `:type` to the input object, except that each parameter value that is a reference or list is replaced by the sequence of files contained by the reference or list.

B.7.2 Ignoring the Status of a Tool Input

By default, a tool is not run if any of its inputs have `ERROR` status or less. In case the tool should ignore the status of an input object, an ampersand is appended to the declaration of that input. For example, the derivation graph entry:

```
EXEC (od.sh) (:FILE)& => (:FILE);
```

declares that the `od.sh` tool does not care what the status of the `:FILE` input object is.

B.7.3 Ignoring Changes to the Value of a Tool Input

If a tool only uses the name of an input file or only uses the names of the elements of an input list, then that input is marked with an at-sign. In this case Odin ignores changes to the value(s) of the input object, and only reruns the tool if the name(s) change. For example, consider a tool that takes all relative names specified in `#include` statements from an input file, and forms the cross product of those names with a set of directory names in a specified search path. A derivation graph entry for such a tool is:

```
EXEC (c_inc.sh) (:c) (+search_path)@ => (:c_inc);
```

This declares that `c_incs.sh` should be re-executed if the `:c` file or the names in the `+search_path` parameter change, but not if the contents of the directories in the search path change.

B.7.4 EXEC Tool

The `EXEC` tool passes its inputs to the Unix `execl()` system call and expects its outputs to be placed in current working directory. For example, the derivation graph entry:

```
EXEC (/bin/ld) -o exe (:o) => (:exe);
```

declares that a file named `exe` is produced in the current working directory by executing the C statement:

```
status = execl("/bin/ld", "/bin/ld", "-o", "exe", "/xxx/filename.o");
```

where `/xxx/filename.o` is the name of a file with type `:o`. When the `execl()` completes, Odin moves all output files into the cache.

An identifier input is passed to the `execl()` as a `C` string.

An input-expression is passed to the `execl()` as an absolute pathname of the file named by the input-expression. Any relative source filename in the input-expression refers to a file in the same directory as the derivation graph file containing the type expression. For example, if an `EXEC` declaration in the derivation graph file `/usr/local/Odin/misc/1.0/misc.dg` contains the input `(od.sh)`, then this input is passed to the `execl()` as `"/usr/local/Odin/misc/1.0/od.sh"`.

The value of a derived or parameter input in the input-expression depends on the input object to which the tool is being applied. For example, in the odin-expression:

```
prog.c +debug :exe
```

the input object to the `:exe` derivation is `prog.c+debug`.

Each derived input is replaced by a file that is the result of applying the derived input object type to the input object. For example, when the `:exe` object is being computed for `prog.c+debug:exe`, the `:o` derived input is replaced by the file `prog.c+debug:o`.

Each parameter input is replaced by the object resulting from applying the object type specified in the parameter declaration for that parameter type, to the list of values of all parameters of that type associated with the input object. As a special case, if no parameters of that type are associated with the input object, the parameter input is replaced by the empty string.

Each `execl()` is invoked in an empty directory provided by Odin, and the tool is expected to put its output files in that directory. The name of an output file should be the same as the type-name of the output. For example, the result of the `execl()` call for the output `(:exe)` should be an output file named `exe`.

In addition to the declared outputs, there are two standard output files named `ERRORS` and `WARNINGS`. Fatal errors should be written to `ERRORS`, while recoverable errors should be written to `WARNINGS`. The `ERRORS` and `WARNINGS` files are used by Odin to determine if error or warning status should be set for the output of the tool.

Normally, the standard input of the `execl()` is assigned to `/dev/null` and the standard output and error is assigned to a temporary file that is written to the standard output of the Odin client process after the `execl()` completes. To aid in debugging of an `EXEC` tool, if the Odin variable `MaxBuilds` is set to `1`, the standard input, output, and error of the `execl()` are those of the Odin client process.

If the exit status of the `execl()` call is non-zero, Odin ignores the results of the `execl()`, and aborts the current odin-command. Non-zero exit status should therefore only be used to indicate transient failures (such as memory or disk errors). This means that most standard Unix tools must be wrapped in shell scripts, so that normal error messages are sent to the file `ERRORS` and only transient errors cause a non-zero exit status.

Since the Odin system caches the results of `EXEC` tools and assumes the results are still valid if none of the input objects change, the tool writer must ensure that all files referenced during a tool run are specified as inputs to that tool. In addition to explicit input files, many tools use additional implicit input files whose names are specified in the text of the explicit input to the tool. It is necessary in these cases to provide a dependency scanning tool that produces as output a list containing the names of all of these additional input files. These additional inputs are specified in the tool declaration separated from the explicit inputs by the keyword `NEEDS`. For example, the derivation graph entry:

```
EXEC (cc.sh) (:c) (+cc_flags) NEEDS (:all_c_inc) => (:o);
```

declares that the script `cc.sh` uses as inputs an object of type `:c` and a parameter of type `+cc_flags`, and indirectly uses as input an object of type `:all_c_incs`.

B.7.5 READ-LIST Tool

The `READ-LIST` tool produces a list from an input object. If the input object is a directory, this list contains all the files in the directory. If the input object is a file, this list contains the objects named by the odin-expressions in the input file. For example, the derivation graph entry:

```
READ-LIST (:c.sm) => (:c.sm.list) ;
```

declares that `:c.sm.list` is the list specified by the contents of a file of type `:c.sm`.

B.7.6 COLLECT Tool

When the output type of a `COLLECT` tool declaration is a list, the `COLLECT` tool produces a list whose elements are the input objects. For example, the derivation graph entry:

```
COLLECT (:c_inc) (:c_inc :map=:all_c_inc) => (:all_c_inc);
:all_c_inc 'all c includes' => :LIST;
```

declares that `:all_c_inc` is a list that contains `:c_inc` and the result of applying `:all_c_inc` to each element of `:c_inc`.

When the output type of a `COLLECT` tool declaration is a reference, the `COLLECT` tool produces a reference to the input object. For example, the derivation graph entry:

```
COLLECT (:c +debug :o) => (:debug.o);
:debug.o 'C compile with debug on' => :REFERENCE;
```

declares that `:debug.o` is a reference to an object that results from applying `+debug:o` to a `:c` file.

B.7.7 READ-REFERENCE Tool

The `READ-REFERENCE` tool produces a reference to the object named by the single odin-expression in the input file. For example, the derivation graph entry:

```
READ-REFERENCE (:stub.name) => (:stub.ref) ;
```

declares that `:stub.ref` will be a reference to the object named by the contents of a file of type `:stub.name`.

Unlike `COLLECT`, `READ-REFERENCE` allows the referee to be selected and/or ratified programmatically by using an `EXEC` script with arbitrary inputs to produce the file containing the reference.

Appendix C

Derivation Graph Syntax

```
DerivationGraph -> DGEntry ';' DerivationGraph
                -> ;

DGEntry         -> 'BANNER' "Identifier"
                -> 'INCLUDE' "Identifier"
                -> Pattern '=' '>' ObjectType
                -> ObjectType Description '=' '>' SuperType+
                -> ParameterType Description '=' '>' ObjectType
                -> Variable Description '=' "Identifier"
                -> Tool Arg+ Needs '=' '>' Result+ ;

Pattern         -> "Identifier" '*'
                -> '*' "Identifier"
                -> '*' ;

ObjectType      -> ':' "Identifier" ;

ParameterType   -> '+' "Identifier" ;

Description     -> "Identifier" '?'
                -> "Identifier" ;

Tool            -> 'EXEC'
                -> 'COLLECT'
                -> 'READ-LIST'
                -> 'READ-REFERENCE' ;

SuperType       -> '<' ObjectType '>'
                -> ObjectType ;

Needs           -> 'NEEDS' Arg+
                -> ;

Result          -> '(' ObjectType ')' ;

Arg             -> Word
                -> FileArg
                -> FileArg '@'
```

```
                -> FileArg '&' ;

FileArg         -> '(' Root Operation* ')' ;

Root            -> Word
                -> ObjectType
                -> ParameterType
                -> '/'
                -> '/' Word
                -> '%' Word ;

Operation       -> Parameter
                -> ObjectType
                -> ObjectType '=' ObjectType
                -> '/' Word
                -> '/'
                -> '%' Word ;

Parameter       -> ParameterType
                -> ParameterType '=' PrmVal+ ;

PrmVal          -> Word
                -> FileArg ;

Word            -> "Identifier"
                -> Variable;

Variable        -> '$' "Identifier";
```

Appendix D

Derivation Graph Example

```
# Source Types

*.c.sm => :c.sm ;

*.c => :c ;

*.o => :o ;

# Object Types

:c.sm 'system model'? => :FILE ;

:c 'C code'? => :FILE ;

:o 'object module'? => :FILE ;

:dir 'directory' => :REFERENCE ;

:c_inc 'potential C-style included files' => :LIST ;

:all_c_inc 'C-style transitively included files' => :LIST ;

:c.list 'c source code list for tools wanting list input' => :c.sm.list ;

:c.sm.list 'list' => :LIST ;

:fmt 'formatted version of C code' => :PIPE ;

:exe 'executable binary'? => :FILE ;

# Parameter Types

+search_path 'name of a directory in an include search path'? => :ls ;

+ignore 'prefix of dependencies to ignore'? => :ls ;
```

```
+debug 'debug switch'? => :first ;

+prof 'profiling switch'? => :first ;

+gnu 'use gnu tools'? => :first ;

+cc_flags 'flags for cc'? => :cat ;

+ld_flags 'flags for ld'? => :cat ;


# Environment Variables

$CC 'name of C compiler' = 'gcc';

$CC_HOME 'directory containing C compiler tools' = '/usr/local/bin';


# Tools

READ-LIST (:c.sm)
   => (:c.sm.list) ;

EXEC (indent.sh) (:c)
   => (:fmt) ;

EXEC (c_inc.sh) (:FILE) (:FILE :dir)@ (+search_path) (+ignore)
   => (:c_inc) ;

COLLECT (:c_inc :map=:all_c_inc) (:c_inc)
   => (:all_c_inc) ;

EXEC (cc.sh) (:c) (:c :dir)@ (+search_path)
 (+debug) (+prof) (+gnu) (+cc_flags) NEEDS (:all_c_inc)& $CC $CC_HOME
   => (:o) ;

COLLECT (:c)
   => (:c.list) ;

EXEC (ld.sh) (:c.sm.list :map=:o) (+debug) (+gnu) (+prof) (+ld_flags)
   => (:exe) ;
```