The OMake user guide and reference manual

Jason Hickey, Aleksey Nogin, *et. al.*

15th March 2007

Contents

Chapter 1

Guide

If you are new to OMake, you the omake-quickstart presents a short introduction that describes how to set up a project. The omake-build-examples gives larger examples of build projects, and omake-language-examples presents programming examples.

Quickstart 2 A quickstart guide to using `omake`.

Build examples 3 Advanced build examples.

The OMake language 4 The `omake` language, including a description of objects, expressions, and values.

Language discussion 5 Further discussion on the language, including scoping, evaluation, and objects.

Language examples 6 Additional language examples.

Build rules 7 Defining and using rules to build programs.

Base builtin functions 8 Functions and variables in the core standard library.

System functions 9 Functions on files, input/output, and system commands.

Shell commands 10 Using the `omake` shell for command-line interpretation.

The standard objects 11 Pervasives defines the built-in objects.

Standard build definitions 12 The build specifications for programming languages in the OMake standard library.

Standard autoconfiguration functions and variables 13 The utilities provoded by the OMake standard library to simplify programming of autoconfiguration tests.

The interactive command interpreter 14 The `osh` command-line interpreter.

Appendices **OMake command-line options A** Command-line options for `omake`.

The OMake language grammar B A more precise specification of the OMake language.

All the documentation on a single page All the OMake documentation in a single page.

Chapter 2

OMake quickstart guide

2.1 Description

omake is designed for building projects that might have source files in several directories. Projects are normally specified using an OMakefile in each of the project directories, and an OMakeroot file in the root directory of the project. The OMakeroot file specifies general build rules, and the OMakefiles specify the build parameters specific to each of the subdirectories. When omake runs, it walks the configuration tree, evaluating rules from all of the OMakefiles. The project is then built from the entire collection of build rules.

2.1.1 Automatic dependency analysis

Dependency analysis has always been problematic with the make(1) program. omake addresses this by adding the .SCANNER target, which specifies a command to produce dependencies. For example, the following rule

```
.SCANNER: %.o: %.c
    $(CC) $(INCLUDE) -MM $<
```

is the standard way to generate dependencies for .c files. omake will automatically run the scanner when it needs to determine dependencies for a file.

2.1.2 Content-based dependency analysis

Dependency analysis in omake uses MD5 digests to determine whether files have changed. After each run, omake stores the dependency information in a file called .omakedb in the project root directory. When a rule is considered for execution, the command is not executed if the target, dependencies, and command sequence are unchanged since the last run of omake. As an optimization, omake does not recompute the digest for a file that has an unchanged modification time, size, and inode number.

2.2 For users already familiar with make

For users already familiar with the `make`(1) command, here is a list of differences to keep in mind when using `omake`.

- In `omake`, you are much less likely to define build rules of your own. The system provides many standard functions (like `StaticCLibrary` and `CProgram`), described in Chapter 12, to specify these builds more simply.
- Implicit rules using `.SUFFIXES` and the `.suf1.suf2:` are not supported. You should use wildcard patterns instead `%.suf2: %.suf1`.
- Scoping is significant: you should define variables and `.PHONY` targets (see Section 7.10) before they are used.
- Subdirectories are incorporated into a project using the `.SUBDIRS:` target (see Section 7.8).

2.3 Building a small C program

To start a new project, the easiest method is to change directories to the project root and use the command `omake --install` to install default `OMakefile`s.

```
$ cd ~/newproject
$ omake --install
*** omake: creating OMakeroot
*** omake: creating OMakefile
*** omake: project files OMakefile and OMakeroot have been installed
*** omake: you should edit these files before continuing
```

The default `OMakefile` contains sections for building C and OCaml programs. For now, we'll build a simple C project.

Suppose we have a C file called `hello_code.c` containing the following code:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

To build the program a program `hello` from this file, we can use the `CProgram` function. The `OMakefile` contains just one line that specifies that the program `hello` is to be built from the source code in the `hello_code.c` file (note that file suffixes are not passed to these functions).

```
CProgram(hello, hello_code)
```

Now we can run `omake` to build the project. Note that the first time we run `omake`, it both scans the `hello_code.c` file for dependencies, and compiles it using the `cc` compiler. The status line printed at the end indicates how many files were scanned, how many were built, and how many MD5 digests were computed.

```
$ omake hello
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.0 sec)
- scan . hello_code.o
+ cc -I. -MM hello_code.c
- build . hello_code.o
+ cc -I. -c -o hello_code.o hello_code.c
- build . hello
+ cc -o hello hello_code.o
*** omake: done (0.5 sec, 1/6 scans, 2/6 rules, 5/22 digests)
$ omake
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.1 sec)
*** omake: done (0.1 sec, 0/4 scans, 0/4 rules, 0/9 digests)
```

If we want to change the compile options, we can redefine the `CC` and `CFLAGS` variables *before* the `CProgram` line. In this example, we will use the `gcc` compiler with the `-g` option. In addition, we will specify a `.DEFAULT` target to be built by default. The `EXE` variable is defined to be `.exe` on `Win32` systems; it is empty otherwise.

```
CC = gcc
CFLAGS += -g
CProgram(hello, hello_code)
.DEFAULT: hello$(EXE)
```

Here is the corresponding run for `omake`.

```
$ omake
*** omake: reading OMakefiles
*** omake: finished reading OMakefiles (0.0 sec)
- scan . hello_code.o
+ gcc -g -I. -MM hello_code.c
- build . hello_code.o
+ gcc -g -I. -c -o hello_code.o hello_code.c
- build . hello
+ gcc -g -o hello hello_code.o
*** omake: done (0.4 sec, 1/7 scans, 2/7 rules, 3/22 digests)
```

We can, of course, include multiple files in the program. Suppose we write a new file `hello_helper.c`. We would include this in the project as follows.

```
CC = gcc
CFLAGS += -g
CProgram(hello, hello_code hello_helper)
.DEFAULT: hello$(EXE)
```

2.4 Larger projects

As the project grows it is likely that we will want to build libraries of code. Libraries can be built using the `StaticCLibrary` function. Here is an example of an `OMakefile` with two libraries.

```
CC = gcc
CFLAGS += -g

FOO_FILES = foo_a foo_b
BAR_FILES = bar_a bar_b bar_c

StaticCLibrary(libfoo, $(FOO_FILES))
StaticCLibrary(libbar, $(BAR_FILES))

# The hello program is linked with both libraries
LIBS = libfoo libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

2.5 Subdirectories

As the project grows even further, it is a good idea to split it into several directories. Suppose we place the `libfoo` and `libbar` into subdirectories.

In each subdirectory, we define an `OMakefile` for that directory. For example, here is an example `OMakefile` for the `foo` subdirectory.

```
INCLUDES += .. ../bar

FOO_FILES = foo_a foo_b
StaticCLibrary(libfoo, $(FOO_FILES))
```

Note the the `INCLUDES` variable is defined to include the other directories in the project.

Now, the next step is to link the subdirectories into the main project. The project `OMakefile` should be modified to include a `.SUBDIRS:` target.

```
# Project configuration
CC = gcc
```

```
CFLAGS += -g

# Subdirectories
.SUBDIRS: foo bar

# The libraries are now in subdirectories
LIBS = foo/libfoo bar/libbar

CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

Note that the variables `CC` and `CFLAGS` are defined *before* the `.SUBDIRS` target. These variables remain defined in the subdirectories, so that `libfoo` and `libbar` use `gcc -g`.

If the two directories are to be configured differently, we have two choices. The `OMakefile` in each subdirectory can be modified with its configuration (this is how it would normally be done). Alternatively, we can also place the change in the root `OMakefile`.

```
# Default project configuration
CC = gcc
CFLAGS += -g

# libfoo uses the default configuration
.SUBDIRS: foo

# libbar uses the optimizing compiler
CFLAGS += -O3
.SUBDIRS: bar

# Main program
LIBS = foo/libfoo bar/libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

Note that the way we have specified it, the `CFLAGS` variable also contains the `-O3` option for the `CProgram`, and `hello_code.c` and `hello_helper.c` file will both be compiled with the `-O3` option. If we want to make the change truly local to `libbar`, we can put the `bar` subdirectory in its own scope using the `section` form.

```
# Default project configuration
CC = gcc
CFLAGS += -g
```

```
# libfoo uses the default configuration
.SUBDIRS: foo

# libbar uses the optimizing compiler
section
    CFLAGS += -O3
    .SUBDIRS: bar

# Main program does not use the optimizing compiler
LIBS = foo/libfoo bar/libbar
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

Later, suppose we decide to port this project to `Win32`, and we discover that we need different compiler flags and an additional library.

```
# Default project configuration
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# libfoo uses the default configuration
.SUBDIRS: foo

# libbar uses the optimizing compiler
section
    CFLAGS += $(if $(equal $(OSTYPE), Win32), $(EMPTY), -O3)
    .SUBDIRS: bar

# Default libraries
LIBS = foo/libfoo bar/libbar

# We need libwin32 only on Win32
if $(equal $(OSTYPE), Win32)
   LIBS += win32/libwin32

   .SUBDIRS: win32
   export

# Main program does not use the optimizing compiler
```

```
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

Note the use of the `export` directives to export the variable definitions from the if-statements. Variables in `omake` are *scoped*—variables in nested blocks (blocks with greater indentation), are not normally defined in outer blocks. The `export` directive specifies that the variable definitions in the nested blocks should be exported to their parent block.

Finally, for this example, we decide to copy all libraries into a common `lib` directory. We first define a directory variable, and replace occurrences of the `lib` string with the variable.

```
# The common lib directory
LIB = $(dir lib)

# phony target to build just the libraries
.PHONY: makelibs

# Default project configuration
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# libfoo uses the default configuration
.SUBDIRS: foo

# libbar uses the optimizing compiler
section
    CFLAGS += $(if $(equal $(OSTYPE), Win32), $(EMPTY), -O3)
    .SUBDIRS: bar

# Default libraries
LIBS = $(LIB)/libfoo $(LIB)/libbar

# We need libwin32 only on Win32
if $(equal $(OSTYPE), Win32)
   LIBS += $(LIB)/libwin32

   .SUBDIRS: win32
   export
```

```
# Main program does not use the optimizing compiler
CProgram(hello, hello_code hello_helper)

.DEFAULT: hello$(EXE)
```

In each subdirectory, we modify the `OMakefile`s in the library directories to install them into the `$(LIB)` directory. Here is the relevant change to `foo/OMakefile`.

```
INCLUDES += .. ../bar

FOO_FILES = foo_a foo_b
StaticCLibraryInstall(makelib, $(LIB), libfoo, $(FOO_FILES))
```

Directory (and file names) evaluate to relative pathnames. Within the `foo` directory, the `$(LIB)` variable evaluates to `../lib`.

As another example, instead of defining the `INCLUDES` variable separately in each subdirectory, we can define it in the toplevel as follows.

```
INCLUDES = $(ROOT) $(dir foo bar win32)
```

In the `foo` directory, the `INCLUDES` variable will evaluate to the string `.. . ../bar ../win32`. In the `bar` directory, it would be `.. ../foo . ../win32`. In the root directory it would be `. foo bar win32`.

2.6 Other things to consider

`omake` also handles recursive subdirectories. For example, suppose the `foo` directory itself contains several subdirectories. The `foo/OMakefile` would then contain its own `.SUBDIRS` target, and each of its subdirectories would contain its own `OMakefile`.

2.7 Building OCaml programs

By default, `omake` is also configured with functions for building OCaml programs. The functions for OCaml program use the `OCaml` prefix. For example, suppose we reconstruct the previous example in OCaml, and we have a file called `hello_code.ml` that contains the following code.

```
open Printf

let () = printf "Hello world\n"
```

An example `OMakefile` for this simple project would contain the following.

```
# Use the byte-code compiler
BYTE_ENABLED = true
NATIVE_ENABLED = false
OCAMLCFLAGS += -g

# Build the program
OCamlProgram(hello, hello_code)
.DEFAULT: hello.run
```

Next, suppose the we have two library subdirectories: the `foo` subdirectory is written in C, the `bar` directory is written in OCaml, and we need to use the standard OCaml `Unix` module.

```
# Default project configuration
if $(equal $(OSTYPE), Win32)
    CC = cl /nologo
    CFLAGS += /DWIN32 /MT
    export
else
    CC = gcc
    CFLAGS += -g
    export

# Use the byte-code compiler
BYTE_ENABLED = true
NATIVE_ENABLED = false
OCAMLCFLAGS += -g

# library subdirectories
INCLUDES += $(dir foo bar)
OCAMLINCLUDES += $(dir foo bar)
.SUBDIRS: foo bar

# C libraries
LIBS = foo/libfoo

# OCaml libraries
OCAML_LIBS = bar/libbar

# Also use the Unix module
OCAML_OTHER_LIBS = unix

# The main program
OCamlProgram(hello, hello_code hello_helper)

.DEFAULT: hello
```

The `foo/OMakefile` would be configured as a C library.

```
FOO_FILES = foo_a foo_b
StaticCLibrary(libfoo, $(FOO_FILES))
```

The `bar/OMakefile` would build an ML library.

```
BAR_FILES = bar_a bar_b bar_c
OCamlLibrary(libbar, $(BAR_FILES))
```

2.8 The OMakefile and OMakeroot files

`OMake` uses the `OMakefile` and `OMakeroot` files for configuring a project. The syntax of these files is the same, but their role is slightly different. For one thing, every project must have exactly one `OMakeroot` file in the project root directory. This file serves to identify the project root, and it contains code that sets up the project. In contrast, a multi-directory project will often have an `OMakefile` in each of the project subdirectories, specifying how to build the files in that subdirectory.

Normally, the `OMakeroot` file is boilerplate. The following listing is a typical example.

```
include $(STDLIB)/build/Common
include $(STDLIB)/build/C
include $(STDLIB)/build/OCaml
include $(STDLIB)/build/LaTeX

# Redefine the command-line variables
DefineCommandVars(.)

# The current directory is part of the project
.SUBDIRS: .
```

The `include` lines include the standard configuration files needed for the project. The `$(STDLIB)` represents the `omake` library directory. The only required configuration file is `Common`. The others are optional; for example, the `$(STDLIB)/build/OCaml` file is needed only when the project contains programs written in OCaml.

The `DefineCommandVars` function defines any variables specified on the command line (as arguments of the form `VAR=<value>`). The `.SUBDIRS` line specifies that the current directory is part of the project (so the `OMakefile` should be read).

Normally, the `OMakeroot` file should be small and project-independent. Any project-specific configuration should be placed in the `OMakefiles` of the project.

2.9 Multiple version support

OMake version `0.9.6` introduced preliminary support for multiple, simultaneous versions of a project. Versioning uses the `vmount(dir1, dir2)` function, which defines a “virtual mount” of directory `dir1` over directory `dir2`. A “virtual mount” is like a transparent mount in Unix, where the files from `dir1` appear in the `dir2` namespace, but new files are created in `dir2`. More precisely, the filename `dir2/foo` refers to: a) the file `dir1/foo` if it exists, or b) `dir2/foo` otherwise.

The `vmount` function makes it easy to specify multiple versions of a project. Suppose we have a project where the source files are in the directory `src/`, and we want to compile two versions, one with debugging support and one optimized. We create two directories, `debug` and `opt`, and mount the `src` directory over them.

```
section
    CFLAGS += -g
    vmount(-l, src, debug)
    .SUBDIRS: debug

section
    CFLAGS += -O3
    vmount(-l, src, opt)
    .SUBDIRS: opt
```

Here, we are using `section` blocks to define the scope of the `vmount`—you may not need them in your project.

The `-l` option is optional. It specifies that files form the `src` directory should be linked into the target directories (or copied, if the system is Win32). The links are added as files are referenced. If no options are given, then files are not copied or linked, but filenames are translated to refer directly to the `src/` files.

Now, when a file is referenced in the `debug` directory, it is linked from the `src` directory if it exists. For example, when the file `debug/OMakefile` is read, the `src/OMakefile` is linked into the `debug/` directory.

The `vmount` model is fairly transparent. The `OMakefile`s can be written *as if* referring to files in the `src/` directory—they need not be aware of mounting. However, there are a few points to keep in mind.

2.10 Notes

- When using the `vmount` function for versioning, it wise to keep the source files distinct from the compiled versions. For example, suppose the source directory contained a file `src/foo.o`. When mounted, the `foo.o` file will be the same in all versions, which is probably not what you want. It is better to keep the `src/` directory pristine, containing no compiled code.

- When using the `vmount -l` option, files are linked into the version directory only if they are referenced in the project. Functions that examine the filesystem (like `$(ls ...)`) may produce unexpected results.

Chapter 3

Additional build examples

Let's explain the OMake build model a bit more. One issue that dominates this discussion is that OMake is based on global project analysis. That means you define a configuration for the *entire* project, and you run *one* instance of omake.

For single-directory projects this doesn't mean much. For multi-directory projects it means a lot. With GNU make, you would usually invoke the `make` program recursively for each directory in the project. For example, suppose you had a project with some project root directory, containing a directory of sources `src`, which in turn contains subdirectories `lib` and `main`. So your project looks like this nice piece of ASCII art.

```
my_project/
|--> Makefile
`--> src/
     |---> Makefile
     |---> lib/
     |     |---> Makefile
     |     `---> source files...
     `---> main/
           |---> Makefile
           `---> source files...
```

Typically, with GNU make, you would start an instance of `make` in `my_project/`; this would in term start an instance of `make` in the `src/` directory; and this would start new instances in `lib/` and `main/`. Basically, you count up the number of `Makefile`s in the project, and that is the number of instances of `make` processes that will be created.

The number of processes is no big deal with today's machines (sometimes contrary the the author's opinion, we no longer live in the 1970s). The problem with the scheme was that each `make` process had a separate configuration, and it took a lot of work to make sure that everything was consistent. Furthermore, suppose the programmer runs `make` in the `main/` directory, but the `lib/` is out-of-date. In this case, `make` would happily crank away, perhaps trying to rebuild files in `lib/`, perhaps just giving up.

With OMake this changes entirely. Well, not entirely. The source structure is quite similar, we merely add some Os to the ASCII art.

```
my_project/
|--> OMakeroot   (or Root.om)
|--> OMakefile
`--> src/
     |---> OMakefile
     |---> lib/
     |     |---> OMakefile
     |     `---> source files...
     `---> main/
           |---> OMakefile
           `---> source files...
```

The role of each `<dir>/OMakefile` plays the same role as each `<dir>/Makefile`: it describes how to build the source files in `<dir>`. The OMakefile retains much of syntax and structure of the Makefile, but in most cases it is much simpler.

One minor difference is the presence of the OMakeroot in the project root. The main purpose of this file is to indicate where the project root *is* in the first place (in case `omake` is invoked from a subdirectory). The `OMakeroot` serves as the bootstrap file; omake starts by reading this file first. Otherwise, the syntax and evaluation of `OMakeroot` is no different from any other `OMakefile`.

The *big* difference is that OMake performs a *global* analysis. Here is what happens when `omake` starts.

1. omake locates that OMakeroot file, and reads it.

2. Each OMakefile points to its subdirectory OMakefiles using the .SUBDIRS target. For example, `my_project/OMakefile` has a rule,

   ```
   .SUBDIRS: src
   ```

   and the `my_project/src/OMakefile` has a rule,

   ```
   .SUBDIRS: lib main
   ```

   `omake` uses these rules to read and evaluate every `OMakefile` in the project. Reading and evaluation is fast. This part of the process is cheap.

3. Now that the entire configuration is read, `omake` determines which files are out-of-date (using a global analysis), and starts the build process. This may take a while, depending on what exactly needs to be done.

There are several advantages to this model. First, since analysis is global, it is much easier to ensure that the build configuration is consistent–after all, there is only one configuration. Another benefit is that the build configuration is inherited, and can be re-used, down the hierarchy. Typically, the root `OMakefile` defines some standard boilerplate and configuration, and this is inherited by subdirectories that tweak and modify it (but do not need to restate it entirely). The disadvantage of course is space, since this is global analysis after all. In practice rarely seems to be a concern; omake takes up much less space than your web browser even on large projects.

Some notes to the GNU/BSD make user.

- OMakefiles are a lot like Makefiles. The syntax is similar, and there many of the builtin functions are similar. However, the two build systems are not the same. Some evil features (in the authors' opinions) have been dropped in OMake, and some new features have been added.
- OMake works the same way on all platforms, including Win32. The standard configuration does the right thing, but if you care about porting your code to multiple platforms, and you use some tricky features, you may need to condition parts of your build config on the `$(OSTYPE)` variable.
- A minor issue is that OMake dependency analysis is based on MD5 file digests. That is, dependencies are based on file *contents*, not file *modification times*. Say goodbye to false rebuilds based on spurious timestamp changes and mismatches between local time and fileserver time.

3.1 OMakeroot vs. OMakefile

Before we begin with examples, let's ask the first question, "What is the difference between the project root OMakeroot and OMakefile?" A short answer is, there is no difference, but you must have an OMakeroot file (or Root.om file).

However, the normal style is that OMakeroot is boilerplate and is more-or-less the same for all projects. The OMakefile is where you put all your project-specific stuff.

To get started, you don't have to do this yourself. In most cases you just perform the following step in your project root directory.

- Run `omake --install` in your project root.

This will create the initial OMakeroot and OMakefile files that you can edit to get started.

3.2 An example C project

To begin, let's start with a simple example. Let's say that we have a full directory tree, containing the following files.

```
my_project/
|--> OMakeroot
|--> OMakefile
`--> src/
     |---> OMakefile
     |---> lib/
     |     |---> OMakefile
     |     |---> ouch.c
     |     |---> ouch.h
     |     `---> bandaid.c
     `---> main/
           |---> OMakefile
           |---> horsefly.c
           |---> horsefly.h
           `---> main.c
```

Here is an example listing.

```
my_project/OMakeroot:
    # Include the standard configuration for C applications
    open build/C

    # Process the command-line vars
    DefineCommandVars()

    # Include the OMakefile in this directory.
    .SUBDIRS: .

my_project/OMakefile:
    # Set up the standard configuration
    CFLAGS += -g

    # Include the src subdirectory
    .SUBDIRS: src

my_project/src/OMakefile:
    # Add any extra options you like
    CFLAGS += -O2

    # Include the subdirectories
    .SUBDIRS: lib main

my_project/src/lib/OMakefile:
    # Build the library as a static library.
    # This builds libbug.a on Unix/OSX, or libbug.lib on Win32.
    # Note that the source files are listed _without_ suffix.
```

```
    StaticCLibrary(libbug, ouch bandaid)

my_project/src/main/OMakefile:
    # Some files include the .h files in ../lib
    INCLUDES += ../lib

    # Indicate which libraries we want to link against.
    LIBS[] +=
        ../lib/libbug

    # Build the program.
    # Builds horsefly.exe on Win32, and horsefly on Unix.
    # The first argument is the name of the executable.
    # The second argument is an array of object files (without suffix)
    # that are part of the program.
    CProgram(horsefly, horsefly main)

    # Build the program by default (in case omake is called
    # without any arguments).  EXE is defined as .exe on Win32,
    # otherwise it is empty.
    .DEFAULT: horsefly$(EXE)
```

Most of the configuration here is defined in the file `build/C.om` (which is part of the OMake distribution). This file takes care of a lot of work, including:

- Defining the `StaticCLibrary` and `CProgram` functions, which describe the canonical way to build C libraries and programs.
- Defining a mechanism for *scanning* each of the source programs to discover dependencies. That is, it defines .SCANNER rules for C source files.

Variables are inherited down the hierarchy, so for example, the value of CFLAGS in src/main/OMakefile is "`-g -O2`".

3.3 An example OCaml project

Let's repeat the example, assuming we are using OCaml instead of C. This time, the directory tree looks like this.

```
    my_project/
    |--> OMakeroot
    |--> OMakefile
    `--> src/
         |---> OMakefile
         |---> lib/
         |     |---> OMakefile
         |     |---> ouch.ml
```

```
         |     |---> ouch.mli
         |     `---> bandaid.ml
         `---> main/
               |---> OMakefile
               |---> horsefly.ml
               |---> horsefly.mli
               `---> main.ml
```

The listing is only a bit different.

```
my_project/OMakeroot:
    # Include the standard configuration for OCaml applications
    open build/OCaml

    # Process the command-line vars
    DefineCommandVars()

    # Include the OMakefile in this directory.
    .SUBDIRS: .

my_project/OMakefile:
    # Set up the standard configuration
    OCAMLFLAGS += -Wa

    # Do we want to use the bytecode compiler,
    # or the native-code one?  Let's use both for
    # this example.
    NATIVE_ENABLED = true
    BYTE_ENABLED = true

    # Include the src subdirectory
    .SUBDIRS: src

my_project/src/OMakefile:
    # Include the subdirectories
    .SUBDIRS: lib main

my_project/src/lib/OMakefile:
    # Let's do aggressive inlining on native code
    OCAMLOPTFLAGS += -inline 10

    # Build the library as a static library.
    # This builds libbug.a on Unix/OSX, or libbug.lib on Win32.
    # Note that the source files are listed _without_ suffix.
    OCamlLibrary(libbug, ouch bandaid)
```

```
my_project/src/main/OMakefile:
    # These files depend on the interfaces in ../lib
    OCAMLINCLUDES += ../lib

    # Indicate which libraries we want to link against.
    OCAML_LIBS[] +=
        ../lib/libbug

    # Build the program.
    # Builds horsefly.exe on Win32, and horsefly on Unix.
    # The first argument is the name of the executable.
    # The second argument is an array of object files (without suffix)
    # that are part of the program.
    OCamlProgram(horsefly, horsefly main)

    # Build the program by default (in case omake is called
    # without any arguments).  EXE is defined as .exe on Win32,
    # otherwise it is empty.
    .DEFAULT: horsefly$(EXE)
```

In this case, most of the configuration here is defined in the file `build/OCaml.om`. In this particular configuration, files in `my_project/src/lib` are compiled aggressively with the option `-inline 10`, but files in `my_project/src/lib` are compiled normally.

3.4 Handling new languages

The previous two examples seem to be easy enough, but they rely on the OMake standard library (the files `build/C` and `build/OCaml`) to do all the work. What happens if we want to write a build configuration for a language that is not already supported in the OMake standard library?

For this example, let's suppose we are adopting a new language. The language uses the standard compile/link model, but is not in the OMake standard library. Specifically, let's say we have the following setup.

- Source files are defined in files with a `.cat` suffix (for Categorical Abstract Terminology).
- `.cat` files are compiled with the `catc` compiler to produce `.woof` files (Wicked Object-Oriented Format).
- `.woof` files are linked by the `catc` compiler with the `-c` option to produce a `.dog` executable (Digital Object Group). The `catc` also defines a `-a` option to combine several `.woof` files into a library.
- Each `.cat` can refer to other source files. If a source file `a.cat` contains a line `open b`, then `a.cat` depends on the file `b.woof`, and `a.cat` must

be recompiled if `b.woof` changes. The `catc` function takes a `-I` option to define a search path for dependencies.

To define a build configuration, we have to do three things.

1. Define a `.SCANNER` rule for discovering dependency information for the source files.
2. Define a generic rule for compiling a `.cat` file to a `.woof` file.
3. Define a rule (as a function) for linking `.woof` files to produce a `.dog` executable.

Initially, these definitions will be placed in the project root `OMakefile`.

3.4.1 Defining a default compilation rule

Let's start with part 2, defining a generic compilation rule. We'll define the build rule as an *implicit* rule. To handle the include path, we'll define a variable `CAT_INCLUDES` that specifies the include path. This will be an array of directories. To define the options, we'll use a lazy variable (Section 6.5). In case there are any other standard flags, we'll define a `CAT_FLAGS` variable.

```
# Define the catc command, in case we ever want to override it
CATC = catc

# The default flags are empty
CAT_FLAGS =

# The directories in the include path (empty by default)
INCLUDES[] =

# Compute the include options from the include path
PREFIXED_INCLUDES[] = $`(mapprefix -I, $(INCLUDES))

# The default way to build a .woof file
%.woof: %.cat
    $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) -c $<
```

The final part is the build rule itself, where we call the `catc` compiler with the include path, and the `CAT_FLAGS` that have been defined. The `$<` variable represents the source file.

3.4.2 Defining a rule for linking

For linking, we'll define another rule describing how to perform linking. Instead of defining an implicit rule, we'll define a function that describes the linking step. The function will take two arguments; the first is the name of the executable (without suffix), and the second is the files to link (also without suffixes). Here is the code fragment.

```
# Optional link options
CAT_LINK_FLAGS =

# The function that defines how to build a .dog program
CatProgram(program, files) =
    # Add the suffixes
    file_names = $(addsuffix .woof, $(files))
    prog_name = $(addsuffix .dog, $(files))

    # The build rule
    $(prog_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -o $@ $+

    # Return the program name
    value $(prog_name)
```

The `CAT_LINK_FLAGS` variable is defined just in case we want to pass additional flags specific to the link step. Now that this function is defined, whenever we want to define a rule for building a program, we simply call the rule. The previous implicit rule specifies how to compile each source file, and the `CatProgram` function specifies how to build the executable.

```
# Build a rover.dog program from the source
# files neko.cat and chat.cat.
# Compile it by default.
.DEFAULT: $(CatProgram rover, neko chat)
```

3.4.3 Dependency scanning

That's it, almost. The part we left out was automated dependency scanning. This is one of the nicer features of OMake, and one that makes build specifications easier to write and more robust. Strictly speaking, it isn't required, but you definitely want to do it.

The mechanism is to define a `.SCANNER` rule, which is like a normal rule, but it specifies how to compute dependencies, not the target itself. In this case, we want to define a `.SCANNER` rule of the following form.

```
.SCANNER: %.woof: %.cat
    <commands>
```

This rule specifies that a `.woof` file may have additional dependencies that can be extracted from the corresponding `.cat` file by executing the `<commands>`. The *result* of executing the `<commands>` should be a sequence of dependencies in OMake format, printed to the standard output.

As we mentioned, each `.cat` file specifies dependencies on `.woof` files with an `open` directive. For example, if the `neko.cat` file contains a line `open chat`, then `neko.woof` depends on `chat.woof`. In this case, the `<commands>` should print the following line.

```
neko.woof: chat.woof
```

For an analogy that might make this clearer, consider the C programming language, where a `.o` file is produced by compiling a `.c` file. If a file `foo.c` contains a line like `#include "fum.h"`, then `foo.c` should be recompiled whenever `fum.h` changes. That is, the file `foo.o` *depends* on the file `fum.h`. In the OMake parlance, this is called an *implicit* dependency, and the `.SCANNER <commands>` would print a line like the following.

```
foo.o: fum.h
```

Now, returning to the animal world, to compute the dependencies of `neko.woof`, we should scan `neko.cat`, line-by-line, looking for lines of the form `open <name>`. We could do this by writing a program, but it is easy enough to do it in `omake` itself. We can use the builtin `awk` function to scan the source file. One slight complication is that the dependencies depend on the `INCLUDE` path. We'll use the `find-in-path` function to find them. Here we go.

```
.SCANNER: %.woof: %.cat
    section
        # Scan the file
        deps[] =
        awk($<)
        case $'^open'
            deps[] += $2
            export

        # Remove duplicates, and find the files in the include path
        deps = $(find-in-path $(INCLUDES), $(set $(deps)))

        # Print the dependencies
        println($"$@: $(deps)")
```

Let's look at the parts. First, the entire body is defined in a `section` because we are computing it internally, not as a sequence of shell commands.

We use the `deps` variable to collect all the dependencies. The `awk` function scans the source file (`$<`) line-by-line. For lines that match the regular expression `^open` (meaning that the line begins with the word `open`), we add the second word on the line to the `deps` variable. For example, if the input line is `open chat`, then we would add the `chat` string to the `deps` array. All other lines in the source file are ignored.

Next, the `$(set $(deps))` expression removes any duplicate values in the `deps` array (sorting the array alphabetically in the process). The `find-in-path` function then finds the actual location of each file in the include path.

The final step is print the result as the string `$"$@: $(deps)"` The quotations are added to flatten the `deps` array to a simple string.

3.4.4 Pulling it all together

To complete the example, let's pull it all together into a single project, much like our previous example.

```
    my_project/
    |--> OMakeroot
    |--> OMakefile
    `--> src/
         |---> OMakefile
         |---> lib/
         |     |---> OMakefile
         |     |---> neko.cat
         |     `---> chat.cat
         `---> main/
               |---> OMakefile
               `---> main.cat
```

The listing for the entire project is as follows. Here, we also include a function `CatLibrary` to link several `.woof` files into a library.

```
my_project/OMakeroot:
    # Process the command-line vars
    DefineCommandVars()

    # Include the OMakefile in this directory.
    .SUBDIRS: .

my_project/OMakefile:
   ########################################################################
   # Standard config for compiling .cat files
   #

   # Define the catc command, in case we ever want to override it
   CATC = catc

   # The default flags are empty
   CAT_FLAGS =

   # The directories in the include path (empty by default)
   INCLUDES[] =

   # Compute the include options from the include path
   PREFIXED_INCLUDES[] = $`(mapprefix -I, $(INCLUDES))

   # Dependency scanner for .cat files
   .SCANNER: %.woof: %.cat
```

```
    section
        # Scan the file
        deps[] =
        awk($<)
        case $'^open'
            deps[] += $2
            export

        # Remove duplicates, and find the files in the include path
        deps = $(find-in-path $(INCLUDES), $(set $(deps)))

        # Print the dependencies
        println($"$@: $(deps)")

# The default way to compile a .cat file
%.woof: %.cat
    $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) -c $<

# Optional link options
CAT_LINK_FLAGS =

# Build a library for several .woof files
CatLibrary(lib, files) =
    # Add the suffixes
    file_names = $(addsuffix .woof, $(files))
    lib_name = $(addsuffix .woof, $(lib))

    # The build rule
    $(lib_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -a $@ $+

    # Return the program name
    value $(lib_name)

# The function that defines how to build a .dog program
CatProgram(program, files) =
    # Add the suffixes
    file_names = $(addsuffix .woof, $(files))
    prog_name = $(addsuffix .dog, $(program))

    # The build rule
    $(prog_name): $(file_names)
        $(CATC) $(PREFIXED_INCLUDES) $(CAT_FLAGS) $(CAT_LINK_FLAGS) -o $@ $+

    # Return the program name
    value $(prog_name)
```

```
   ########################################################################
   # Now the program proper
   #

   # Include the src subdirectory
   .SUBDIRS: src

my_project/src/OMakefile:
   .SUBDIRS: lib main

my_project/src/lib/OMakefile:
   CatLibrary(cats, neko chat)

my_project/src/main/OMakefile:
   # Allow includes from the ../lib directory
   INCLUDES[] += ../lib

   # Build the program
   .DEFAULT: $(CatProgram main, main ../cats)
```

Some notes. The configuration in the project `OMakeroot` defines the standard configuration, including the dependency scanner, the default rule for compiling source files, and functions for building libraries and programs.

These rules and functions are inherited by subdirectories, so the `.SCANNER` and build rules are used automatically in each subdirectory, so you don't need to repeat them.

3.4.5 Finishing up

At this point we are done, but there are a few things we can consider.

First, the rules for building cat programs is defined in the project `OMakefile`. If you had another cat project somewhere, you would need to copy the `OMakeroot` (and modify it as needed). Instead of that, you should consider moving the configuration to a shared library directory, in a file like `Cat.om`. That way, instead of copying the code, you could include the shared copy with an OMake command `open Cat`. The share directory should be added to your `OMAKEPATH` environment variable to ensure that `omake` knows how to find it.

Better yet, if you are happy with your work, consider submitting it as a standard configuration (by sending a request to `omake@metaprl.org`) so that others can make use of it too.

3.5 Collapsing the hierarchy, .SUBDIRS bodies

Some projects have many subdirectories that all have the same configuration. For instance, suppose you have a project with many subdirectories, each containing a set of images that are to be composed into a web page. Apart from the specific images, the configuration of each file is the same.

To make this more concrete, suppose the project has four subdirectories `page1`, `page2`, `page3`, and `page4`. Each contains two files `image1.jpg` and `image2.jpg` that are part of a web page generated by a program `genhtml`.

Instead of of defining a `OMakefile` in each directory, we can define it as a body to the `.SUBDIRS` command.

```
.SUBDIRS: page1 page2 page3 page4
    index.html: image1.jpg image2jpg
        genhtml $+ > $@
```

The body of the `.SUBDIRS` is interpreted exactly as if it were the `OMakefile`, and it can contain any of the normal statements. The body is evaluated *in the subdirectory* for each of the subdirectories. We can see this if we add a statement that prints the current directory (`$(CWD)`).

```
  .SUBDIRS: page1 page2 page3 page4
      println($(absname $(CWD)))
      index.html: image1.jpg image2jpg
          genhtml $+ > $@
# prints
  /home/jyh/.../page1
  /home/jyh/.../page2
  /home/jyh/.../page3
  /home/jyh/.../page4
```

3.5.1 Using glob patterns

Of course, this specification is quite rigid. In practice, it is likely that each subdirectory will have a different set of images, and all should be included in the web page. One of the easier solutions is to use one of the directory-listing functions, like `glob` or `ls`. The `glob` function takes a shell pattern, and returns an array of file with matching filenames in the current directory.

```
.SUBDIRS: page1 page2 page3 page4
    IMAGES = $(glob *.jpg)
    index.html: $(IMAGES)
        genhtml $+ > $@
```

3.5.2 Simplified sub-configurations

Another option is to add a configuration file in each of the subdirectories that defines directory-specific information. For this example, we might define a file `BuildInfo.om` in each of the subdirectories that defines a list of images in that directory. The `.SUBDIRS` line is similar, but we include the BuildInfo file.

```
.SUBDIRS: page1 page2 page3 page4
    include BuildInfo   # Defines the IMAGES variable

    index.html: $(IMAGES)
        genhtml $+ > $@
```

Where we might have the following configurations.

```
page1/BuildInfo.om:
    IMAGES[] = image.jpg
page2/BuildInfo.om:
    IMAGES[] = ../common/header.jpg winlogo.jpg
page3/BuildInfo.om:
    IMAGES[] = ../common/header.jpg unixlogo.jpg daemon.jpg
page4/BuildInfo.om:
    IMAGES[] = fee.jpg fi.jpg foo.jpg fum.jpg
```

3.5.3 Computing the subdirectory list

The other hardcoded specification is the list of subdirectories `page1`, ..., `page4`. Rather than editing the project `OMakefile` each time a directory is added, we could compute it (again with `glob`).

```
.SUBDIRS: $(glob page*)
    index.html: $(glob *.jpg)
        genhtml $+ > $@
```

Alternately, the directory structure may be hierarchical. Instead of using `glob`, we could use the `subdirs` function, returns each of the directories in a hierarchy. For example, this is the result of evaluating the `subdirs` function in the omake project root. The `P` option, passed as the first argument, specifies that the listing is "proper," it should not include the `omake` directory itself.

```
osh> subdirs(P, .)
- : <array
        /home/jyh/.../omake/mk : Dir
        /home/jyh/.../omake/RPM : Dir
        ...
        /home/jyh/.../omake/osx_resources : Dir>
```

Using `subdirs`, our example is now as follows.

```
.SUBDIRS: $(subdirs P, .)
    index.html: $(glob *.jpg)
        genhtml $+ > $@
```

In this case, *every* subdirectory will be included in the project.

If we are using the `BuildInfo.om` option. Instead of including every subdirectory, we could include only those that contain a `BuildInfo.om` file. For this purpose, we can use the `find` function, which traverses the directory hierarchy looking for files that match a test expression. In our case, we want to search for files with the name `BuildInfo.om`. Here is an example call.

```
osh> FILES = $(find . -name BuildInfo.om)
- : <array
        /home/jyh/.../omake/doc/html/BuildInfo.om : File
        /home/jyh/.../omake/src/BuildInfo.om : File
        /home/jyh/.../omake/tests/simple/BuildInfo.om : File>
osh> DIRS = $(dirof $(FILES))
- : <array
        /home/jyh/.../omake/doc/html : Dir
        /home/jyh/.../omake/src : Dir
        /home/jyh/.../omake/tests/simple : Dir>
```

In this example, there are three `BuildInfo.om` files, in the `doc/html`, `src`, and `tests/simple` directories. The `dirof` function returns the directories for each of the files.

Returning to our original example, we modify it as follows.

```
.SUBDIRS: $(dirof $(find . -name BuildInfo.om))
    include BuildInfo   # Defines the IMAGES variable

    index.html: $(IMAGES)
        genhtml $+ > $@
```

3.5.4 Temporary directories

Sometimes, your project may include temporary directories–directories where you place intermediate results. these directories are deleted whenever the project is cleanup up. This means, in particular, that you can't place an `OMakefile` in a temporary directory, because it will be removed when the directory is removed.

Instead, if you need to define a configuration for any of these directories, you will need to define it using a `.SUBDIRS` body.

```
section
    CREATE_SUBDIRS = true

    .SUBDIRS: tmp
        # Compute an MD5 digest
        %.digest: %.comments
           echo $(digest $<) > $@

        # Extract comments from the source files
```

```
        %.comments: ../src/%.src
           grep '^#' $< > $@

        .DEFAULT: foo.digest

.PHONY: clean

clean:
    rm -rf tmp
```

In this example, we define the CREATE_SUBDIRS variable as true, so that the tmp directory will be created if it does not exist. The .SUBDIRS body in this example is a bit contrived, but it illustrates the kind of specification you might expect. The clean phony-target indicates that the tmp directory should be removed when the project is cleaned up.

Chapter 4

OMake concepts and syntax

Projects are specified to `omake` with `OMakefile`s. The `OMakefile` has a format similar to a `Makefile`. An `OMakefile` has three main kinds of syntactic objects: variable definitions, function definitions, and rule definitions.

4.1 Variables

Variables are defined with the following syntax. The name is any sequence of alphanumeric characters, underscore `_`, and hyphen `-`.

```
<name> = <value>
```

Values are defined as a sequence of literal characters and variable expansions. A variable expansion has the form `$(<name>)`, which represents the value of the `<name>` variable in the current environment. Some examples are shown below.

```
CC = gcc
CFLAGS = -Wall -g
COMMAND = $(CC) $(CFLAGS) -O2
```

In this example, the value of the `COMMAND` variable is the string `gcc -Wall -g -O2`.

Unlike `make`(1), variable expansion is *eager* and *functional* (see also the section on Scoping). That is, variable values are expanded immediately and new variable definitions do not affect old ones. For example, suppose we extend the previous example with following variable definitions.

```
X = $(COMMAND)
COMMAND = $(COMMAND) -O3
Y = $(COMMAND)
```

In this example, the value of the `X` variable is the string `gcc -Wall -g -O2` as before, and the value of the `Y` variable is `gcc -Wall -g -O2 -O3`.

4.2 Adding to a variable definition

Variables definitions may also use the += operator, which adds the new text to an existing definition. The following two definitions are equivalent.

```
# Add options to the CFLAGS variable
CFLAGS = $(CFLAGS) -Wall -g

# The following definition is equivalent
CFLAGS += -Wall -g
```

4.3 Arrays

Arrays can be defined by appending the `[]` sequence to the variable name and defining initial values for the elements as separate lines. Whitespace is significant on each line. The following code sequence prints `c d e`.

```
X[] =
    a b
    c d e
    f

println($(nth 2, $(X)))
```

4.4 Special characters and quoting

The following characters are special to `omake`: `$():,=#\`. To treat any of these characters as normal text, they should be escaped with the backslash character `\`.

```
DOLLAR = \$
```

Newlines may also be escaped with a backslash to concatenate several lines.

```
FILES = a.c\
        b.c\
        c.c
```

Note that the backslash is *not* an escape for any other character, so the following works as expected (that is, it preserves the backslashes in the string).

```
DOSTARGET = C:\WINDOWS\control.ini
```

An alternative mechanism for quoting special text is the use `$"..."` escapes. The number of double-quotations is arbitrary. The outermost quotations are not included in the text.

```
A = $""String containing "quoted text" ""
B = $"""Multi-line
    text.
    The # character is not special"""
```

4.5 Function definitions

Functions are defined using the following syntax.

```
<name>(<params>) =
   <indented-body>
```

The parameters are a comma-separated list of identifiers, and the body must be placed on a separate set of lines that are indented from the function definition itself. For example, the following text defines a function that concatenates its arguments, separating them with a colon.

```
ColonFun(a, b) =
    return($(a):$(b))
```

The `return` expression can be used to return a value from the function. A `return` statement is not required; if it is omitted, the returned value is the value of the last expression in the body to be evaluated. NOTE: as of version `0.9.6`, `return` is a control operation, causing the function to immediately return. In the following example, when the argument `a` is true, the function `f` immediately returns the value 1 without evaluating the print statement.

```
f(a) =
   if $(a)
      return 1
   println(The argument is false)
   return 0
```

In many cases, you may wish to return a value from a section or code block without returning from the function. In this case, you would use the `value` operator. In fact, the `value` operator is not limited to functions, it can be used any place where a value is required. In the following definition, the variable `X` is defined as 1 or 2, depending on the value of a, then result is printed, and returned from the function.

```
f_value(a) =
   X =
      if $(a)
         value 1
      else
         value 2
   println(The value of X is $(X))
   value $(X)
```

Functions are called using the GNU-make syntax, `$(<name> <args>)`, where `<args>` is a comma-separated list of values. For example, in the following program, the variable `X` contains the value `foo:bar`.

```
X = $(ColonFun foo, bar)
```

If the value of a function is not needed, the function may also be called using standard function call notation. For example, the following program prints the string "She says: Hello world".

```
Printer(name) =
    println($(name) says: Hello world)

Printer(She)
```

4.6 Comments

Comments begin with the `#` character and continue to the end of the line.

4.7 File inclusion

Files may be included with the `include` or `open` form. The included file must use the same syntax as an `OMakefile`.

```
include $(Config_file)
```

The `open` operation is similar to an `include`, but the file is included at most once.

```
open Config

# Repeated opens are ignored, so this
# line has no effect.
open Config
```

If the file specified is not an absolute filenmame, both `include` and `open` operations search for the file based on the `OMAKEPATH` variable. In case of the `open` directive, the search is performed at *parse* time, and the argument to `open` may not contain any expressions.

4.8 Scoping, sections

Scopes in `omake` are defined by indentation level. When indentation is increased, such as in the body of a function, a new scope is introduced.

The `section` form can also be used to define a new scope. For example, the following code prints the line `X = 2`, followed by the line `X = 1`.

```
X = 1
section
    X = 2
    println(X = $(X))

println(X = $(X))
```

This result may seem surprising–the variable definition within the `section` is not visible outside the scope of the `section`.

The `export` form, which will be described in detail in Section 5.3, can be used to circumvent this restriction by exporting variable values from an inner scope. It must be the final expression in a scope. For example, if we modify the previous example by adding an `export` expression, the new value for the `X` variable is retained, and the code prints the line `X = 2` twice.

```
X = 1
section
    X = 2
    println(X = $(X))
    export

println(X = $(X))
```

There are also cases where separate scoping is quite important. For example, each `OMakefile` is evaluated in its own scope. Since each part of a project may have its own configuration, it is important that variable definitions in one `OMakefile` do not affect the definitions in another.

To give another example, in some cases it is convenient to specify a separate set of variables for different build targets. A frequent idiom in this case is to use the `section` command to define a separate scope.

```
section
   CFLAGS += -g
   %.c: %.y
       $(YACC) $<
   .SUBDIRS: foo

.SUBDIRS: bar baz
```

In this example, the `-g` option is added to the `CFLAGS` variable by the `foo` subdirectory, but not by the `bar` and `baz` directories. The implicit rules are scoped as well and in this example, the newly added yacc rule will be inherited by the `foo` subdirectory, but not by the `bar` and `baz` ones; furthermore this implicit rule will not be in scope in the current directory.

4.9 Conditionals

Top level conditionals have the following form.

```
if <test>
   <true-clause>
elseif <text>
   <elseif-clause>
else
   <else-clause>
```

The `<test>` expression is evaluated, and if it evaluates to a *true* value (see Section 8.2 for more information on logical values, and Boolean functions), the code for the `<true-clause>` is evaluated; otherwise the remaining clauses are evaluated. There may be multiple `elseif` clauses; both the `elseif` and `else` clauses are optional. Note that the clauses are indented, so they introduce new scopes.

When viewed as a predicate, a value corresponds to the Boolean *false*, if its string representation is the empty string, or one of the strings `false`, `no`, `nil`, `undefined`, or `0`. All other values are *true*.

The following example illustrates a typical use of a conditional. The `OSTYPE` variable is the current machine architecture.

```
# Common suffixes for files
if $(equal $(OSTYPE), Win32)
   EXT_LIB = .lib
   EXT_OBJ = .obj
   EXT_ASM = .asm
   EXE = .exe
   export
elseif $(mem $(OSTYPE), Unix Cygwin)
   EXT_LIB = .a
   EXT_OBJ = .o
   EXT_ASM = .s
   EXE =
   export
else
   # Abort on other architectures
   eprintln(OS type $(OSTYPE) is not recognized)
   exit(1)
```

4.10 Matching

Pattern matching is performed with the `switch` and `match` forms.

```
switch <string>
```

```
case <pattern1>
    <clause1>
case <pattern2>
    <clause2>
...
default
   <default-clause>
```

The number of cases is arbitrary. The `default` clause is optional; however, if it is used it should be the last clause in the pattern match.

For `switch`, the string is compared with the patterns literally.

```
switch $(HOST)
case mymachine
    println(Building on mymachine)
default
    println(Building on some other machine)
```

Patterns need not be constant strings. The following function tests for a literal match against `pattern1`, and a match against `pattern2` with `##` delimiters.

```
Switch2(s, pattern1, pattern2) =
   switch $(s)
   case $(pattern1)
       println(Pattern1)
   case $"##$(pattern2)##"
       println(Pattern2)
   default
       println(Neither pattern matched)
```

For `match` the patterns are `egrep`(1)-style regular expressions. The numeric variables `$1, $2, ...` can be used to retrieve values that are matched by `\(...\)` expressions.

```
match $(NODENAME)@$(SYSNAME)@$(RELEASE)
case $"mymachine.*@\(.*\)@\(.*\)"
    println(Compiling on mymachine; sysname $1 and release $2 are ignored)

case $".*@Linux@.*2\.4\.\(.*\)"
    println(Compiling on a Linux 2.4 system; subrelease is $1)

default
    eprintln(Machine configuration not implemented)
    exit(1)
```

4.11 Objects

OMake is an object-oriented language. Generally speaking, an object is a value that contains fields and methods. An object is defined with a `.` suffix for a variable. For example, the following object might be used to specify a point $(1, 5)$ on the two-dimensional plane.

```
Coord. =
    x = 1
    y = 5
    print(message) =
       println($"$(message): the point is ($(x), $(y)")

# Define X to be 5
X = $(Coord.x)

# This prints the string, "Hi: the point is (1, 5)"
Coord.print(Hi)
```

The fields `x` and `y` represent the coordinates of the point. The method `print` prints out the position of the point.

4.12 Classes

We can also define *classes*. For example, suppose we wish to define a generic `Point` class with some methods to create, move, and print a point. A class is really just an object with a name, defined with the `class` directive.

```
Point. =
    class Point

    # Default values for the fields
    x = 0
    y = 0

    # Create a new point from the coordinates
    new(x, y) =
       this.x = $(x)
       this.y = $(y)
       return $(this)

    # Move the point to the right
    move-right() =
       x = $(add $(x), 1)
       return $(this)
```

```
      # Print the point
      print() =
         println($"The point is ($(x), $(y)")

p1 = $(Point.new 1, 5)
p2 = $(p1.move-right)

# Prints "The point is (1, 5)"
p1.print()

# Prints "The point is (2, 5)"
p2.print()
```

Note that the variable `$(this)` is used to refer to the current object. Also, classes and objects are *functional*—the `new` and `move-right` methods return new objects. In this example, the object `p2` is a different object from `p1`, which retains the original $(1, 5)$ coordinates.

4.13 Inheritance

Classes and objects support inheritance (including multiple inheritance) with the `extends` directive. The following definition of `Point3D` defines a point with `x`, `y`, and `z` fields. The new object inherits all of the methods and fields of the parent classes/objects.

```
Z. =
   z = 0

Point3D. =
   extends $(Point)
   extends $(Z)
   class Point3D

   print() =
      println($"The 3D point is ($(x), $(y), $(z))")

# The "new" method was not redefined, so this
# defines a new point (1, 5, 0).
p = $(Point3D.new 1, 5)
```

4.14 Special objects/sections

Objects provide one way to manage the OMake namespace. There are also four special objects that are further used to control the namespace.

4.15 private.

The `private.` section is used to define variables that are private to the current file/scope. The values are not accessible outside the scope. Variables defined in a `private.` object can be accessed only from within the section where they are defined.

```
Obj. =
   private. =
      X = 1

   print() =
      println(The value of X is: $(X))

# Prints:
#    The private value of X is: 1
Obj.print()

# This is an error--X is private in Obj
y = $(Obj.X)
```

In addition, private definitions do not affect the global value of a variable.

```
# The public value of x is 1
x = 1
f() =
    println(The public value of x is: $(x))

# This object uses a private value of x
Obj. =
    private. =
       x = 2

    print() =
       x = 3
       println(The private value of x is: $(x))
       f()

# Prints:
#    The private value of x is: 3
#    The public value of x is: 1
Obj.print()
```

Private variables have two additional properties.

1. Private variables are local to the file in which they are defined.

2. Private variables are not exported by the `export` directive, unless they are mentioned explicitly.

```
private. =
   FLAG = true

section
   FLAG = false
   export

# FLAG is still true
section
   FLAG = false
   export FLAG

# FLAG is now false
```

4.16 protected.

The `protected.` object is used to define fields that are local to an object. They can be accessed as fields, but they are not passed dynamically to other functions. The purpose of a protected variable is to prevent a variable definition within the object from affecting other parts of the project.

```
X = 1
f() =
   println(The public value of X is: $(X))

# Prints:
#    The public value of X is: 2
section
   X = 2
   f()

# X is a protected field in the object
Obj. =
   protected. =
      X = 3

   print() =
      println(The protected value of X is: $(X))
      f()

# Prints:
```

```
#    The protected value of X is: 3
#    The public value of X is: 1
Obj.print()

# This is legal, it defines Y as 3
Y = $(Obj.X)
```

In general, it is a good idea to define object variables as protected. The resulting code is more modular because variables in your object will not produce unexpected clashes with variables defined in other parts of the project.

4.17 public.

The `public.` object is used to specify public dynamically-scoped variables. In the following example, the `public.` object specifies that the value `X = 4` is to be dynamically scoped. Public variables *are not* defined as fields of an object.

```
X = 1
f() =
   println(The public value of X is: $(X))

# Prints:
#    The public value of X is: 2
section
   X = 2
   f()

Obj. =
   protected. =
      X = 3

   print() =
      println(The protected value of X is: $(X))
      public. =
         X = 4
      f()

# Prints:
#    The protected value of X is: 3
#    The public value of X is: 4
Obj.print()
```

4.18 static.

The `static.` object is used to specify values that are persistent across runs of OMake. They are frequently used for configuring a project. Configuring a project can be expensive, so the `static.` object ensure that the configuration is performed just once. In the following (somewhat trivial) example, a `static` section is used to determine if the LaTeX command is available. The `$(where latex)` function returns the full pathname for `latex`, or `false` if the command is not found.

```
static. =
   LATEX_ENABLED = false
   print(--- Determining if LaTeX is installed )
   if $(where latex)
       LATEX_ENABLED = true
       export

   if $(LATEX_ENABLED)
      println($'(enabled)')
   else
      println($'(disabled)')
```

The OMake standard library provides a number of useful functions for programming the `static.` tests, as described in Chapter 13. Using the standard library, the above can be rewritten as

```
open configure/Configure
static. =
   LATEX_ENABLED = $(CheckProg latex)
```

As a matter of style, a `static.` section that is used for configuration should print what it is doing using the `ConfMsgChecking` and `ConfMsgResult` functions (of couse, most of helper functions in the standard library would do that automatically).

4.19 Short syntax for scoping objects

The usual dot-notation can be used for private, protected, and public variables (but not static variables).

```
# Public definition of X
public.X = 1

# Private definition of X
private.X = 2

# Prints:
#    The public value of X is: 1
#    The private value of X is: 2
println(The public value of X is: $(public.X))
println(The private value of X is: $(private.X))
```

4.20 Modular programming

The scoping objects help provide a form of modularity. When you write a new file or program, explicit scoping declarations can be used to define an explicit interface for your code, and help avoid name clashes with other parts of the project. Variable definitions are public by default, but you can control this with private definitions.

```
# These variables are private to this file
private. =
   FILES = foo1 foo2 foo3
   SUFFIX = .o
   OFILES = $(addsuffix $(SUFFIX), $(FILES))

# These variables are public
public. =
   CFLAGS += -g

# Build the files with the -g option
$(OFILES):
```

Chapter 5

Expressions and values

omake provides a full programming-language including many system and IO functions. The language is object-oriented – everything is an object, including the base values like numbers and strings. However, the omake language differs from other scripting languages in three main respects.

- Scoping is dynamic.
- Apart from IO, the language is entirely functional – there is no assignment operator in the language.
- Evaluation is normally eager – that is, expressions are evaluated as soon as they are encountered.

To illustrate these features, we will use the osh(1) omake program shell. The osh(1) program provides a toploop, where expressions can be entered and the result printed. osh(1) normally interprets input as command text to be executed by the shell, so in many cases we will use the value form to evaluate an expression directly.

```
osh> 1
*** omake error: File -: line 1, characters 0-1 command not found: 1
osh> value 1
- : "1" : Sequence
osh> ls -l omake
-rwxrwxr-x  1 jyh jyh 1662189 Aug 25 10:24 omake*
```

5.1 Dynamic scoping

Dynamic scoping means that the value of a variable is determined by the most recent binding of the variable in scope at runtime. Consider the following program.

```
OPTIONS = a b c
f() =
   println(OPTIONS = $(OPTIONS))
g() =
   OPTIONS = d e f
   f()
```

If `f()` is called without redefining the `OPTIONS` variable, the function should print the string `OPTIONS = a b c`.

In contrast, the function `g()` redefines the `OPTIONS` variable and evaluates `f()` in that scope, which now prints the string `OPTIONS = d e f`.

The body of `g` defines a local scope – the redefinition of the `OPTIONS` variable is local to `g` and does not persist after the function terminates.

```
osh> g()
OPTIONS = d e f
osh> f()
OPTIONS = a b c
```

Dynamic scoping can be tremendously helpful for simplifying the code in a project. For example, the `OMakeroot` file defines a set of functions and rules for building projects using such variables as `CC`, `CFLAGS`, etc. However, different parts of a project may need different values for these variables. For example, we may have a subdirectory called `opt` where we want to use the `-O3` option, and a subdirectory called `debug` where we want to use the `-g` option. Dynamic scoping allows us to redefine these variables in the parts of the project without having to redefine the functions that use them.

```
section
   CFLAGS = -O3
   .SUBDIRS: opt
section
   CFLAGS = -g
   .SUBDIRS: debug
```

However, dynamic scoping also has drawbacks. First, it can become confusing: you might have a variable that is intended to be private, but it is accidentally redefined elsewhere. For example, you might have the following code to construct search paths.

```
PATHSEP = :
make-path(dirs) =
   return $(concat $(PATHSEP), $(dirs))

make-path(/bin /usr/bin /usr/X11R6/bin)
- : "/bin:/usr/bin:/usr/X11R6/bin" : String
```

However, elsewhere in the project, the `PATHSEP` variable is redefined as a directory separator `/`, and your function suddenly returns the string `/bin//usr/bin//usr/X11R6/bin`, obviously not what you want.

The `private` block is used to solve this problem. Variables that are defined in a `private` block use static scoping – that is, the value of the variable is determined by the most recent definition in scope in the source text.

```
private
   PATHSEP = :
make-path(dirs) =
   return $(concat $(PATHSEP), $(dirs))

PATHSEP = /
make-path(/bin /usr/bin /usr/X11R6/bin)
- : "/bin:/usr/bin:/usr/X11R6/bin" : String
```

5.2 Functional evaluation

Apart from I/O, `omake` programs are entirely functional. This has two parts:

- There is no assignment operator.
- Functions are values, and may be passed as arguments, and returned from functions just like any other value.

The second item is straightforward. For example, the following program defines an increment function by returning a function value.

```
incby(n) =
   g(i) =
      return $(add $(i), $(n))
   return $(g)

f = $(incby 5)

value $(f 3)
- : 8 : Int
```

The first item may be the most confusing initially. Without assignment, how is it possible for a subproject to modify the global behavior of the project? In fact, the omission is intentional. Build scripts are much easier to write when there is a guarantee that subprojects do not interfere with one another.

However, there are times when a subproject needs to propagate information back to its parent object, or when an inner scope needs to propagate information back to the outer scope.

5.3 Exporting the environment

The `export` directive can be used to propagate all or part of an inner scope back to its parent. The `export` directive should be the last statement in a block. If used without arguments, the entire scope is propagated back to the parent; otherwise the arguments specify which part of the environment to propagate. The most common usage is to export the definitions in a conditional block. In the following example, the variable `B` is bound to 2 after the conditional. The `A` variable is not redefined.

```
if $(test)
   A = 1
   B = $(add $(A), 1)
   export B
else
   B = 2
   export
```

If the `export` directive is used without an argument, all of the following is exported:

- The values of all the dynamically scoped variables (as described in Section 4.17).
- The current working directory.
- The current Unix environment.
- The current implicit rules and implicit dependencies (see also Section 7.11.1).
- The current set of "phony" target declarations (see Sections 7.10 and 7.11.3).

If the `export` directive is used with an argument, the argument expression is evaluated and the resulting value is interpreted as follows:

- If the value is empty, everything is exported, as described above.
- If the value represents a environment (or a partial environment) captured using the `export` function, then the corresponding environment or partial environment is exported.
- Otherwise, the value must be a sequence of strings specifying which items are to be propagated back. The following strings have special meaning:
  - `.RULE` — implicit rules and implicit dependencies.
  - `.PHONY` — the set of "phony" target declarations.

  All other strings are interpreted as names of the variables that need to be propagated back.

For example, in the following (somewhat artificial) example, the variables A and B will be exported, and the implicit rule will remain in the environment after the section ends, but the variable TMP and the target tmp_phony will remain unchanged.

```
WANT_TO_EXPORT[] = A B .RULE
section
   A = 1
   B = 2
   TMP = $(add $(A), $(B))

   .PHONY: tmp_phony

   tmp_phony:
      prepare_foo

   %.foo: %.bar tmp_phony
      compute_foo $(TMP) $< $@
   export $(WANT_TO_EXPORT)
```

5.4 Eager evaluation

Evaluation in omake is eager. That is, expressions are evaluated as soon as they are encountered by the evaluator. One effect of this is that the right-hand-side of a variable definition is expanded when the variable is defined.

```
osh> A = 1
- : "1"
osh> A = $(A)$(A)
- : "11"
```

In the second definition, A = (A)(A), the right-hand-side is evaluated first, producing the sequence 11. Then the variable A is *redefined* as the new value. When combined with dynamic scoping, this has many of the same properties as conventional imperative programming.

```
osh> A = 1
- : "1"
osh> printA() =
    println($"A = $A")
osh> A = $(A)$(A)
- : "11"
osh> printA()
11
```

In this example, the print function is defined in the scope of A. When it is called on the last line, the dynamic value of A is 11, which is what is printed.

However, dynamic scoping and imperative programming should not be confused. The following example illustrates a difference. The second `printA` is not in the scope of the definition `A = x$(A)$(A)x`, so it prints the original value, `1`.

```
osh> A = 1
- : "1"
osh> printA() =
    println($"A = $A")
osh> section
         A = x$(A)$(A)x
         printA()
x11x
osh> printA()
1
```

See also Section 6.5 for further ways to control the evaluation order through the use of "lazy" expressions.

5.5 Objects

`omake` is an object-oriented language. Everything is an object, including base values like numbers and strings. In many projects, this may not be so apparent because most evaluation occurs in the default toplevel object, the `Pervasives` object, and few other objects are ever defined.

However, objects provide additional means for data structuring, and in some cases judicious use of objects may simplify your project.

Objects are defined with the following syntax. This defines `name` to be an object with several methods an values.

```
name. =                         # += may be used as well
   extends parent-object        # optional
   class class-name             # optional

   # Fields
   X = value
   Y = value

   # Methods
   f(args) =
      body
   g(arg) =
      body
```

An `extends` directive specifies that this object inherits from the specified `parent-object`. The object may have any number of `extends` directives. If there is more than on `extends` directive, then fields and methods are inherited from all parent objects. If there are name conflicts, the later definitions override the earlier definitions.

The `class` directive is optional. If specified, it defines a name for the object that can be used in `instanceof` operations, as well as `::` scoping directives discussed below.

The body of the object is actually an arbitrary program. The variables defined in the body of the object become its fields, and the functions defined in the body become its methods.

5.6 Field and method calls

The fields and methods of an object are named using `object.name` notation. For example, let's define a one-dimensional point value.

```
Point. =
   class Point

   # Default value
   x = $(int 0)

   # Create a new point
   new(x) =
      x = $(int $(x))
      return $(this)

   # Move by one
   move() =
      x = $(add $(x), 1)
      return $(this)

osh> p1 = $(Point.new 15)
osh> value $(p1.x)
- : 15 : Int

osh> p2 = $(p1.move)
osh> value $(p2.x)
- : 16 : Int
```

The `$(this)` variable always represents the current object. The expression `$(p1.x)` fetches the value of the `x` field in the `p1` object. The expression `$(Point.new 15)` represents a method call to the `new` method of the `Point` object, which returns a new object with 15 as its initial value. The expression `$(p1.move)` is also a method call, which returns a new object at position 16.

Note that objects are functional — it is not possible to modify the fields or methods of an existing object in place. Thus, the `new` and `move` methods return new objects.

5.7 Method override

Suppose we wish to create a new object that moves by 2 units, instead of just 1. We can do it by overriding the `move` method.

```
Point2. =
   extends $(Point)

   # Override the move method
   move() =
      x = $(add $(x), 2)
      return $(this)

osh> p2 = $(Point2.new 15)
osh> p3 = $(p2.move)
osh> value $(p3.x)
- : 17 : Int
```

However, by doing this, we have completely replaced the old `move` method.

5.8 Super calls

Suppose we wish to define a new `move` method that just calls the old one twice. We can refer to the old definition of move using a super call, which uses the notation `$(classname::name <args>)`. The `classname` should be the name of the superclass, and `name` the field or method to be referenced. An alternative way of defining the `Point2` object is then as follows.

```
Point2. =
   extends $(Point)

   # Call the old method twice
   move() =
      this = $(Point::move)
      return $(Point::move)
```

Note that the first call to `$(Point::move)` redefines the current object (the `this` variable). This is because the method returns a new object, which is re-used for the second call.

Chapter 6

Additional language examples

In this section, we'll explore the core language through a series of examples (examples of the build system are the topic of the Chapter 3).

For most of these examples, we'll use the `osh` command interpreter. For simplicity, the values printed by `osh` have been abbreviated.

6.1 Strings and arrays

The basic OMake values are strings, sequences, and arrays of values. Sequences are like arrays of values separated by whitespace; the sequences are split on demand by functions that expect arrays.

```
osh> X = 1 2
- : "1 2" : Sequence
osh> addsuffix(.c, $X)
- : <array 1.c 2.c> : Array
```

Sometimes you want to define an array explicitly. For this, use the `[]` brackets after the variable name, and list each array entry on a single indented line.

```
osh> A[] =
        Hello world
        $(getenv HOME)
- : <array "Hello world" "/home/jyh"> : Array
```

One central property of arrays is that whitespace in the elements is significant. This can be useful, especially for filenames that contain whitespace.

```
# List the current files in the directory
 osh> ls -Q
```

```
"fee"  "fi"  "foo"  "fum"
osh> NAME[] =
        Hello world
- : <array "Hello world"> : Array
osh> touch $(NAME)
osh> ls -Q
"fee"  "fi"  "foo"  "fum"  "Hello world"
```

6.2 Quoted strings

A `String` is a single value; whitespace is significant in a string. Strings are introduced with quotes. There are four kinds of quoted elements; the kind is determined by the opening quote. The symbols `'` (single-quote) and `"` (double-quote) introduce the normal shell-style quoted elements. The quotation symbols are *included* in the result string. Variables are always expanded within a quote of this kind. Note that the `osh`(1) (Chapter 14) printer escapes double-quotes within the string; these are only for printing, they are not part of the string itself.

```
osh> A = 'Hello "world"'
- : "'Hello \"world\"'" : String
osh> B = "$(A)"
- : "\"'Hello \"world\"'\"" : String
osh> C = 'Hello \'world\''
- : "'Hello 'world''" : String
```

A second kind of quote is introduced with the `$'` and `$"` quotes. The number of opening and closing quote symbols is arbitrary. These quotations have several properties:

- The quote delimiters are not part of the string.
- Backslash \ symbols within the string are treated as normal characters.
- The strings may span several lines.
- Variables are expanded within `$"` sequences, but not within `$'` sequences.

```
osh> A = $'''Here $(IS) an '''' \(example\) string['''
- : "Here $(IS) an '''' \\(example\\) string[" : String
osh> B = $""""A is "$(A)" """"
- : "A is \"Here $(IS) an '''' \\(example\\) string[\" " : String
osh> value $(A.length)
- : 38 : Int
osh> value $(A.nth 5)
- : "$" : String
osh> value $(A.rev)
- : "[gnirts )\\elpmaxe(\\ '''' na )SI($ ereH" : String
```

Strings and sequences both have the property that they can be merged with adjacent non-whitespace text.

```
osh> A = a b c
- : "a b c" : Sequence
osh> B = $(A).c
- : <sequence "a b c" : Sequence ".c" : Sequence> : Sequence
osh> value $(nth 2, $(B))
- : "c.c" : String
osh> value $(length $(B))
- : 3 : Int
```

Arrays are different. The elements of an array are never merged with adjacent text of any kind. Arrays are defined by adding square brackets `[]` after a variable name and defining the elements with an indented body. The elements may include whitespace.

```
osh> A[] =
        a b
        foo bar
- : <array
       "a b" : Sequence
       "foo bar" : Sequence>
       : Array
osh> echo $(A).c
a b foo bar .c
osh> value $(A.length)
- : 2 : Int
osh> value $(A.nth 1)
- : "foo bar" : Sequence
```

Arrays are quite helpful on systems where filenames often contain whitespace.

```
osh> FILES[] =
         c:\Documents and Settings\jyh\one file
         c:\Program Files\omake\second file

osh> CFILES = $(addsuffix .c, $(FILES))
osh> echo $(CFILES)
c:\Documents and Settings\jyh\one file.c c:\Program Files\omake\second file.c
```

6.3 Files and directories

OMake projects usually span multiple directories, and different parts of the project execute commands in different directories. There is a need to define a location-independent name for a file or directory.

This is done with the `$(file <names>)` and `$(dir <names>)` functions.

```
osh> mkdir tmp
osh> F = $(file fee)
osh> section:
         cd tmp
         echo $F
../fee
osh> echo $F
fee
```

Note the use of a `section:` to limit the scope of the `cd` command. The section temporarily changes to the `tmp` directory where the name of the file is `../fee`. Once the section completes, we are still in the current directory, where the name of the file is `fee`.

One common way to use the file functions is to define proper file names in your project `OMakefile`, so that references within the various parts of the project will refer to the same file.

```
osh> cat OMakefile
ROOT = $(dir .)
TMP  = $(dir tmp)
BIN  = $(dir bin)
...
```

6.4 Iteration, mapping, and foreach

Most builtin functions operate transparently on arrays.

```
osh> addprefix(-D, DEBUG WIN32)
- : -DDEBUG -DWIN32 : Array
osh> mapprefix(-I, /etc /tmp)
- : -I /etc -I /tmp : Array
osh> uppercase(fee fi foo fum)
- : FEE FI FOO FUM : Array
```

The `mapprefix` and `addprefix` functions are slightly different (the `addsuffix` and `mapsuffix` functions are similar). The `addprefix` adds the prefex to each array element. The `mapprefix` doubles the length of the array, adding the prefix as a new array element before each of the original elements.

Even though most functions work on arrays, there are times when you will want to do it yourself. The `foreach` function is the way to go. The `foreach` function has two forms, but the form with a body is most useful. In this form, the function takes two arguments and a body. The second argument is an array, and the first is a variable. The body is evaluated once for each element of the array, where the variable is bound to the element. Let's define a function to add 1 to each element of an array of numbers.

```
osh> add1(l) =
         foreach(i, $l):
             add($i, 1)
osh> add1(7 21 75)
- : 8 22 76 : Array
```

Sometimes you have an array of filenames, and you want to define a rule for each of them. Rules are not special, you can define them anywhere a statement is expected. Say we want to write a function that describes how to process each file, placing the result in the `tmp/` directory.

```
TMP = $(dir tmp)

my-special-rule(files) =
   foreach(name, $(files))
      $(TMP)/$(name): $(name)
         process $< > $@
```

Later, in some other part of the project, we may decide that we want to use this function to process some files.

```
# These are the files to process in src/lib
MY_SPECIAL_FILES[] =
    fee.src
    fi.src
    file with spaces in its name.src
my-special-rule($(MY_SPECIAL_FILES))
```

The result of calling `my-special-rule` is exactly the same as if we had written the following three rules explicitly.

```
$(TMP)/fee.src: fee.src
    process fee > $@
$(TMP)/fi.src: fi.src
    process fi.src > $@
$(TMP)/$"file with spaces in its name.src": $"file with spaces in its name.src"
    process $< > $@
```

Of course, writing these rules is not nearly as pleasant as calling the function. The usual properties of function abstraction give us the usual benefits. The code is less redundant, and there is a single location (the `my-special-rule` function) that defines the build rule. Later, if we want to modify/update the rule, we need do so in only one location.

6.5 Lazy expressions

Evaluation in `omake` is normally eager. That is, expressions are evaluated as soon as they are encountered by the evaluator. One effect of this is that the right-hand-side of a variable definition is expanded when the variable is defined.

There are two ways to control this behavior. The `$`(v)` form introduces lazy behavior, and the `$,(v)` form restores eager behavior. Consider the following sequence.

```
osh> A = 1
- : "1" : Sequence
osh> B = 2
- : "2" : Sequence
osh> C = $`(add $(A), $,(B))
- : $(apply add $(apply A) "2" : Sequence)
osh> println(C = $(C))
C = 3
osh> A = 5
- : "5" : Sequence
osh> B = 6
- : "6" : Sequence
osh> println(C = $(C))
C = 7
```

The definition `C = $`(add $(A), $,(B))` defines a lazy application. The `add` function is not applied in this case until its value is needed. Within this expression, the value `$,(B)` specifies that `B` is to be evaluated immediately, even though it is defined in a lazy expression.

The first time that we print the value of `C`, it evaluates to 3 since `A` is 1 and `B` is 2. The second time we evaluate `C`, it evaluates to 7 because `A` has been redefined to `5`. The second definition of `B` has no effect, since it was evaluated at definition time.

6.5.1 A larger example of lazy expressions

Lazy expressions are not evaluated until their result is needed. Some people, including this author, frown on overuse of lazy expressions, mainly because it is difficult to know when evaluation actually happens. However, there are cases where they pay off.

One example comes from option processing. Consider the specification of "include" directories on the command line for a C compiler. If we want to include files from /home/jyh/include and ../foo, we specify it on the command line with the options `-I/home/jyh/include -I../foo`.

Suppose we want to define a generic rule for building C files. We could define a `INCLUDES` array to specify the directories to be included, and then define a generic implicit rule in our root OMakefile.

```
# Generic way to compile C files.
CFLAGS = -g
INCLUDES[] =
%.o: %.c
   $(CC) $(CFLAGS) $(INCLUDES) -c $<
```

```
# The src directory builds my_widget+ from 4 source files.
# It reads include files from the include directory.
.SUBDIRS: src
    FILES = fee fi foo fum
    OFILES = $(addsuffix .o, $(FILES))
    INCLUDES[] += -I../include
    my_widget: $(OFILES)
       $(CC) $(CFLAGS) -o $@ $(OFILES)
```

But this is not quite right. The problem is that INCLUDES is an array of options, not directories. If we later wanted to recover the directories, we would have to strip the leading `-I` prefix, which is a hassle. Furthermore, we aren't using proper names for the directories. The solution here is to use a lazy expression. We'll define INCLUDES as a directory array, and a new variable `PREFIXED_INCLUDES` that adds the -I prefix. The `PREFIXED_INCLUDES` is computed lazily, ensuring that the value uses the most recent value of the INCLUDES variable.

```
# Generic way to compile C files.
CFLAGS = -g
INCLUDES[] =
PREFIXED_INCLUDES[] = $`(addprefix -I, $(INCLUDES))
%.o: %.c
   $(CC) $(CFLAGS) $(PREFIXED_INCLUDES) -c $<

# For this example, we define a proper name for the include directory
STDINCLUDE = $(dir include)

# The src directory builds my_widget+ from 4 source files.
# It reads include files from the include directory.
.SUBDIRS: src
    FILES = fee fi foo fum
    OFILES = $(addsuffix .o, $(FILES))
    INCLUDES[] += $(STDINCLUDE)
    my_widget: $(OFILES)
       $(CC) $(CFLAGS) -o $@ $(OFILES)
```

Note that there is a close connection between lazy values and functions. In the example above, we could equivalently define `PREFIXED_INCLUDES` as a function with zero arguments.

```
PREFIXED_INCLUDES() =
    addprefix(-I, $(INCLUDES))
```

6.6 Scoping and exports

The OMake language is functional (apart from IO and shell commands). This comes in two parts: functions are first-class, and variables are immutable (there is no assignment operator). The latter property may seem strange to users used to GNU make, but it is actually a central point of OMake. Since variables can't be modified, it is impossible (or at least hard) for one part of the project to interfere with another.

To be sure, pure functional programming can be awkward. In OMake, each new indentation level introduces a new scope, and new definitions in that scope are lost when the scope ends. If OMake were overly strict about scoping, we would wind up with a lot of convoluted code.

```
osh> X = 1
osh> setenv(BOO, 12)
osh> if $(equal $(OSTYPE), Win32)
         setenv(BOO, 17)
         X = 2
osh> println($X $(getenv BOO))
1 12
```

The `export` command presents a way out. It takes care of "exporting" a value (or the entire variable environment) from an inner scope to an outer one.

```
osh> X = 1
osh> setenv(BOO, 12)
osh> if $(equal $(OSTYPE), Win32)
         setenv(BOO, 17)
         X = 2
         export
osh> println($X $(getenv BOO))
2 17
```

Exports are especially useful in loop to export values from one iteration of a loop to the next.

```
# Ok, let's try to add up the elements of the array
osh>sum(l) =
        total = 0
        foreach(i, $l)
            total = $(add $(total), $i)
        value $(total)
osh>sum(1 2 3)
- : 0 : Int

# Oops, that didn't work!
osh>sum(l) =
```

```
        total = 0
        foreach(i, $l)
            total = $(add $(total), $i)
            export
        value $(total)
osh>sum(1 2 3)
- : 6 : Int
```

A `while` loop is another form of loop, with an auto-export.

```
osh>i = 0
osh>total = 0
osh>while $(lt $i, 10)
        total = $(add $(total), $i)
        i = $(add $i, 1)
osh>println($(total))
45
```

6.7 Shell aliases

Sometimes you may want to define an *alias*, an OMake command that masquerades as a real shell command. You can do this by adding your function as a method to the `Shell` object.

For an example, suppose we use the `awk` function to print out all the comments in a file.

```
osh>cat comment.om
# Comment function
comments(filename) =
    awk($(filename))
    case $'^#'
        println($0)
# File finished
osh>include comment
osh>comments(comment.om)
# Comment function
# File finished
```

To add it as an alias, add the method (using += to preserve the existing entries in the Shell).

```
osh>Shell. +=
        printcom(argv) =
            comments($(nth 0, $(argv)))
osh>printcom comment.om > output.txt
osh>cat output.txt
```

```
# Comment function
# File finished
```

A shell command is passed an array of arguments `argv`. This does *not* include the name of the alias.

6.8 Input/output redirection on the cheap

As it turns out, scoping also provides a nice alternate way to perform redirection. Suppose you have already written a lot of code that prints to the standard output channel, but now you decide you want to redirect it. One way to do it is using the technique in the previous example: define your function as an alias, and then use shell redirection to place the output where you want.

There is an alternate method that is easier in some cases. The variables `stdin`, `stdout`, and `stderr` define the standard I/O channels. To redirect output, redefine these variables as you see fit. Of course, you would normally do this in a nested scope, so that the outer channels are not affected.

```
osh>f() =
        println(Hello world)
osh>f()
Hello world
osh>section:
        stdout = $(fopen output.txt, w)
        f()
        close($(stdout))
osh>cat output.txt
Hello world
```

This also works for shell commands. If you like to gamble, you can try the following example.

```
osh>f() =
        println(Hello world)
osh>f()
Hello world
osh>section:
        stdout = $(fopen output.txt, w)
        f()
        cat output.txt
        close($(stdout))
osh>cat output.txt
Hello world
Hello world
```

Chapter 7

Rules

Rules are used by OMake to specify how to build files. At its simplest, a rule has the following form.

```
<target>: <dependencies>
    <commands>
```

The `<target>` is the name of a file to be built. The `<dependencies>` are a list of files that are needed before the `<target>` can be built. The `<commands>` are a list of indented lines specifying commands to build the target. For example, the following rule specifies how to compile a file `hello.c`.

```
hello.o: hello.c
    $(CC) $(CFLAGS) -c -o hello.o hello.c
```

This rule states that the `hello.o` file depends on the `hello.c` file. If the `hello.c` file has changed, the command `$(CC) $(CFLAGS) -c -o hello.o hello.c` is to be executed to update the target file `hello.o`.

A rule can have an arbitrary number of commands. The individual command lines are executed independently by the command shell. The commands do not have to begin with a tab, but they must be indented from the dependency line.

In addition to normal variables, the following special variables may be used in the body of a rule.

- `$*`: the target name, without a suffix.
- `$@`: the target name.
- `$^`: a list of the sources, in alphabetical order, with duplicates removed.
- `$+`: all the sources, in the original order.
- `$<`: the first source.

For example, the above `hello.c` rule may be simplified as follows.

```
hello.o: hello.c
    $(CC) $(CFLAGS) -c -o $@ $<
```

Unlike normal values, the variables in a rule body are expanded lazily, and binding is dynamic. The following function definition illustrates some of the issues.

```
CLibrary(name, files) =
    OFILES = $(addsuffix .o, $(files))

    $(name).a: $(OFILES)
        $(AR) cq $@ $(OFILES)
```

This function defines a rule to build a program called `$(name)` from a list of `.o` files. The files in the argument are specified without a suffix, so the first line of the function definition defines a variable `OFILES` that adds the `.o` suffix to each of the file names. The next step defines a rule to build a target library `$(name).a` from the `$(OFILES)` files. The expression `$(AR)` is evaluated when the function is called, and the value of the variable `AR` is taken from the caller's scope (see also the section on Scoping).

7.1 Implicit rules

Rules may also be implicit. That is, the files may be specified by wildcard patterns. The wildcard character is `%`. For example, the following rule specifies a default rule for building `.o` files.

```
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $*.c
```

This rule is a template for building an arbitrary `.o` file from a `.c` file.

By default, implicit rules are only used for the targets in the current directory. However subdirectories included via the `.SUBDIRS` rules inherit all the implicit rules that are in scope (see also the section on Scoping).

7.2 Bounded implicit rules

Implicit rules may specify the set of files they apply to. The following syntax is used.

```
<targets>: <pattern>: <dependencies>
    <commands>
```

For example, the following rule applies only to the files `a.o` and `b.o`.

```
a.o b.o: %.o: %.c
    $(CC) $(CFLAGS) -DSPECIAL -c $*.c
```

7.3 section

Frequently, the commands in a rule body are expressions to be evaluated by the shell. `omake` also allows expressions to be evaluated by `omake` itself.

The syntax of these "computed rules" uses the `section` expression. The following rule uses the `omake` IO functions to produce the target `hello.c`.

```
hello.c:
    section
        FP = fopen(hello.c, w)
        fprintln($(FP), $""#include <stdio.h> int main() { printf("Hello world\n"); }"")
        close($(FP))
```

This example uses the quotation `$""...""` (see also Section B.1.6) to quote the text being printed. These quotes are not included in the output file. The `fopen`, `fprintln`, and `close` functions perform file IO as discussed in the IO section.

In addition, commands that are function calls, or special expressions, are interpreted correctly. Since the `fprintln` function can take a file directly, the above rule can be abbreviated as follows.

```
hello.c:
   fprintln($@, $""#include <stdio.h> int main() { printf("Hello world\n"); }"")
```

7.4 section rule

Rules can also be computed using the `section rule` form, where a rule body is expected instead of an expression. In the following rule, the file `a.c` is copied onto the `hello.c` file if it exists, otherwise `hello.c` is created from the file `default.c`.

```
hello.c:
    section rule
       if $(target-exists a.c)
          hello.c: a.c
             cat a.c > hello.c
       else
          hello.c: default.c
             cp default.c hello.c
```

7.5 Special dependencies

7.5.1 :exists:

In some cases, the contents of a dependency do not matter, only whether the file exists or not. In this case, the `:exists:` qualifier can be used for the dependency.

```
foo.c: a.c :exists: .flag
   if $(test -e .flag)
       $(CP) a.c $@
```

7.5.2 :effects:

Some commands produce files by side-effect. For example, the `latex`(1) command produces a `.aux` file as a side-effect of producing a `.dvi` file. In this case, the `:effects:` qualifier can be used to list the side-effect explicitly. `omake` is careful to avoid simultaneously running programs that have overlapping side-effects.

```
paper.dvi: paper.tex :effects: paper.aux
    latex paper
```

7.5.3 :value:

The `:value:` dependency is used to specify that the rule execution depends on the value of an expression. For example, the following rule

```
a: b c :value: $(X)
    ...
```

specifies that "a" should be recompiled if the value of `$(X)` changes (X does not have to be a filename). This is intended to allow greater control over dependencies.

In addition, it can be used instead of other kinds of dependencies. For example, the following rule:

```
a: b :exists: c
    commands
```

is the same as

```
a: b :value: $(target-exists c)
    commands
```

Notes:

- The values are arbitrary (they are not limited to variables)
- The values are evaluated at rule expansion time, so expressions containing variables like `$@`, `$^`, etc are legal.

7.6 .SCANNER rules

Scanner rules define a way to specify automatic dependency scanning. A `.SCANNER` rule has the following form.

```
.SCANNER: target: dependencies
    commands
```

The rule is used to compute additional dependencies that might be defined in the source files for the specified target. The result of executing the scanner commands *must* be a sequence of dependencies in OMake format, printed to the standard output. For example, on GNU systems the `gcc -MM foo.c` produces dependencies for the file `foo.c` (based on `#include` information).

We can use this to specify a scanner for C files that adds the scanned dependencies for the `.o` file. The following scanner specifies that dependencies for a file, say `foo.o` can be computed by running `gcc -MM foo.c`. Furthermore, `foo.c` is a dependency, so the scanner should be recomputed whenever the `foo.c` file changes.

```
.SCANNER: %.o: %.c
    gcc -MM $<
```

Let's suppose that the command `gcc -MM foo.c` prints the following line.

```
foo.o: foo.h /usr/include/stdio.h
```

The result is that the files `foo.h` and `/usr/include/stdio.h` are considered to be dependencies of `foo.o`—that is, `foo.o` should be rebuilt if either of these files changes.

This works, to an extent. One nice feature is that the scanner will be re-run whenever the `foo.c` file changes. However, one problem is that dependencies in C are *recursive*. That is, if the file `foo.h` is modified, it might include other files, establishing further dependencies. What we need is to re-run the scanner if `foo.h` changes too.

We can do this with a *value* dependency. The variable `$&` is defined as the dependency results from any previous scan. We can add these as dependencies using the `digest` function, which computes an MD5 digest of the files.

```
.SCANNER: %.o: %.c :value: $(digest $&)
    gcc -MM $<
```

Now, when the file `foo.h` changes, its digest will also change, and the scanner will be re-run because of the value dependency (since `$&` will include `foo.h`).

This still is not quite right. The problem is that the C compiler uses a *search-path* for include files. There may be several versions of the file `foo.h`, and the one that is chosen depends on the include path. What we need is to base the dependencies on the search path.

The `$(digest-in-path-optional ...)` function computes the digest based on a search path, giving us a solution that works.

```
.SCANNER: %.o: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
   gcc -MM $(addprefix -I, $(INCLUDES)) $<
```

The standard output of the scanner rules will be captured by OMake and is not allowed to contain any content that OMake will not be able to parse as a dependency. The output is allowed to contain dependency specifications for unrelated targets, however such dependencies will be ignored. The scanner rules are allowed to produce arbitrary output on the standard error channel — such output will be handled in the same way as the output of the ordinary rules (in other words, it will be presented to the user, when dictated by the `--output-`... options enabled).

Additional examples of the `.SCANNER` rules can be found in Section 3.4.3.

7.6.1 Named scanners, and the `:scanner:` dependencies

Sometimes it may be useful to specify explicitly which scanner should be used in a rule. For example, we might compile `.c` files with different options, or (heaven help us) we may be using both `gcc` and the Microsoft Visual C++ compiler `cl`. In general, the target of a `.SCANNER` is not tied to a particular target, and we may name it as we like.

```
.SCANNER: scan-gcc-%.c: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
    gcc -MM $(addprefix -I, $(INCLUDES)) $<

.SCANNER: scan-cl-%.c: %.c :value: $(digest-in-path-optional $(INCLUDES), $&)
    cl --scan-dependencies-or-something $(addprefix /I, $(INCLUDES)) $<
```

The next step is to define explicit scanner dependencies. The `:scanner:` dependency is used for this. In this case, the scanner dependencies are specified explicitly.

```
$(GCC_FILES): %.o: %.c :scanner: scan-gcc-%c
    gcc ...

$(CL_FILES): %.obj: %.c :scanner: scan-cl-%c
    cl ...
```

Explicit `:scanner:` scanner specification may also be used to state that a single `.SCANNER` rule should be used to generate dependencies for more than one target. For example,

```
.SCANNER: scan-all-c: $(GCC_FILES) :value: $(digest-in-path-optional $(INCLUDES), $&)
    gcc -MM $(addprefix -I, $(INCLUDES)) $(GCC_FILES)

$(GCC_FILES): %.o: %.c :scanner: scan-all-c
    ...
```

The above has the advantage of only running `gcc` once and a disadvantage that when a single source file changes, all the files will end up being re-scanned.

7.6.2 Notes

In most cases, you won't need to define scanners of your own. The standard installation includes default scanners (both explicitly and implicitly named ones) for C, OCaml, and LaTeX files.

The `SCANNER_MODE` variable controls the usage of implicit scanner dependencies.

The explicit `:scanner:` dependencies reduce the chances of scanner misspecifications. In large complicated projects it might be a good idea to set `SCANNER_MODE` to `error` and use only the named `.SCANNER` rules and explicit `:scanner:` specifications.

7.7 .DEFAULT

The `.DEFAULT` target specifies a target to be built by default if `omake` is run without explicit targets. The following rule instructs `omake` to build the program `hello` by default

```
.DEFAULT: hello
```

7.8 .SUBDIRS

The `.SUBDIRS` target is used to specify a set of subdirectories that are part of the project. Each subdirectory should have its own `OMakefile`, which is evaluated in the context of the current environment.

```
.SUBDIRS: src doc tests
```

This rule specifies that the `OMakefile`s in each of the `src`, `doc`, and `tests` directories should be read.

In some cases, especially when the `OMakefile`s are very similar in a large number of subdirectories, it is inconvenient to have a separate `OMakefile` for each directory. If the `.SUBDIRS` rule has a body, the body is used instead of the `OMakefile`.

```
.SUBDIRS: src1 src2 src3
   println(Subdirectory $(CWD))
   .DEFAULT: lib.a
```

In this case, the `src1`, `src2`, and `src3` files do not need `OMakefile`s. Furthermore, if one exists, it is ignored. The following includes the file if it exists.

```
.SUBDIRS: src1 src2 src3
    if $(file-exists OMakefile)
       include OMakefile
    .DEFAULT: lib.a
```

7.9 .INCLUDE

The `.INCLUDE` target is like the `include` directive, but it specifies a rule to build the file if it does not exist.

```
.INCLUDE: config
    echo "CONFIG_READ = true" > config

 echo CONFIG_READ is $(CONFIG_READ)
```

You may also specify dependencies to an `.INCLUDE` rule.

```
.INCLUDE: config: config.defaults
   cp config.defaults config
```

A word of caution is in order here. The usual policy is used for determining when the rule is out-of-date. The rule is executed if any of the following hold.

- the target does not exist,
- the rule has never been executed before,
- any of the following have changed since the last time the rule was executed,
  - the target,
  - the dependencies,
  - the commands-text.

In some of the cases, this will mean that the rule is executed even if the target file already exists. If the target is a file that you expect to edit by hand (and therefore you don't want to overwrite it), you should make the rule evaluation conditional on whether the target already exists.

```
.INCLUDE: config: config.defaults
    # Don't overwrite my carefully hand-edited file
    if $(not $(file-exists config))
       cp config.defaults config
```

7.10 .PHONY

A "phony" target is a target that is not a real file, but exists to collect a set of dependencies. Phony targets are specified with the `.PHONY` rule. In the following example, the `install` target does not correspond to a file, but it corresponds to some commands that should be run whenever the `install` target is built (for example, by running `omake install`).

```
.PHONY: install

install: myprogram.exe
   cp myprogram.exe /usr/bin
```

7.11 Rule scoping

As we have mentioned before, `omake` is a *scoped* language. This provides great flexibility—different parts of the project can define different configurations without interfering with one another (for example, one part of the project might be compiled with `CFLAGS=-O3` and another with `CFLAGS=-g`).

But how is the scope for a target file selected? Suppose we are building a file `dir/foo.o`. `omake` uses the following rules to determine the scope.

- First, if there is an *explicit* rule for building `dir/foo.o` (a rule with no wildcards), the context for that rule determines the scope for building the target.
- Otherwise, the directory `dir/` must be part of the project. This normally means that a configuration file `dir/OMakefile` exists (although, see the `.SUBDIRS` section for another way to specify the `OMakefile`). In this case, the scope of the target is the scope at the end of the `dir/OMakefile`.

To illustrate rule scoping, let's go back to the example of a "Hello world" program with two files. Here is an example `OMakefile` (the two definitions of `CFLAGS` are for illustration).

```
# The executable is compiled with debugging
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# Redefine CFLAGS
CFLAGS += -O3
```

In this project, the target `hello` is *explicit*. The scope of the `hello` target is the line beginning with `hello:`, where the value of `CFLAGS` is `-g`. The other two targets, `hello_code.o` and `hello_lib.o` do not appear as explicit targets, so their scope is at the end of the `OMakefile`, where the `CFLAGS` variable is defined to be `-g -O3`. That is, `hello` will be linked with `CFLAGS=-g` and the `.o` files will be compiled with `CFLAGS=-g -O3`.

We can change this behavior for any of the targets by specifying them as explicit targets. For example, suppose we wish to compile `hello_lib.o` with a preprocessor variable `LIBRARY`.

```
# The executable is compiled with debugging
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# Compile hello_lib.o with CFLAGS = -g -DLIBRARY
section
```

```
        CFLAGS += -DLIBRARY
        hello_lib.o:

    # Redefine CFLAGS
    CFLAGS += -O3
```

In this case, `hello_lib.o` is also mentioned as an explicit target, in a scope where `CFLAGS=-g -DLIBRARY`. Since no rule body is specified, it is compiled using the usual implicit rule for building `.o` files (in a context where `CFLAGS=-g -DLIBRARY`).

7.11.1 Scoping of implicit rules

Implicit rules (rules containing wildcard patterns) are *not* global, they follow the normal scoping convention. This allows different parts of a project to have different sets of implicit rules. If we like, we can modify the example above to provide a new implicit rule for building `hello_lib.o`.

```
# The executable is compiled with debugging
CFLAGS = -g
hello: hello_code.o hello_lib.o
   $(CC) $(CFLAGS) -o $@ $+

# Compile hello_lib.o with CFLAGS = -g -DLIBRARY
section
    %.o: %.c
        $(CC) $(CFLAGS) -DLIBRARY -c $<
    hello_lib.o:

# Redefine CFLAGS
CFLAGS += -O3
```

In this case, the target `hello_lib.o` is built in a scope with a new implicit rule for building `%.o` files. The implicit rule adds the `-DLIBRARY` option. This implicit rule is defined only for the target `hello_lib.o`; the target `hello_code.o` is built as normal.

7.11.2 Scoping of `.SCANNER` rules

Scanner rules are scoped the same way as normal rules. If the `.SCANNER` rule is explicit (containing no wildcard patterns), then the scope of the scan target is the same as the the rule. If the `.SCANNER` rule is implicit, then the environment is taken from the `:scanner:` dependency.

```
# The executable is compiled with debugging
CFLAGS = -g
hello: hello_code.o hello_lib.o
```

```
   $(CC) $(CFLAGS) -o $@ $+

# scanner for .c files
.SCANNER: scan-c-%.c: %.c
   $(CC) $(CFLAGS) -MM $<

# Compile hello_lib.o with CFLAGS = -g -DLIBRARY
section
    CFLAGS += -DLIBRARY
    hello_lib.o: hello_lib.c :scanner: scan-c-hello_lib.c
       $(CC) $(CFLAGS) -c $<

# Compile hello_code.c with CFLAGS = -g -O3
section
    CFLAGS += -O3
    hello_code.o: hello_code.c :scanner: scan-c-hello_code.c
       $(CC) $(CFLAGS) -c $<
```

Again, this is for illustration—it is unlikely you would need to write a complicated configuration like this! In this case, the `.SCANNER` rule specifies that the C-compiler should be called with the `-MM` flag to compute dependencies. For the target `hello_lib.o`, the scanner is called with `CFLAGS=-g -DLIBRARY`, and for `hello_code.o` it is called with `CFLAGS=-g -O3`.

7.11.3 Scoping for `.PHONY` targets

Phony targets (targets that do not correspond to files) are defined with a `.PHONY:` rule. Phony targets are scoped as usual. The following illustrates a common mistake, where the `.PHONY` target is declared *after* it is used.

```
# !!This example is broken!!
all: hello

hello: hello_code.o hello_lib.o
    $(CC) $(CFLAGS) -o $@ $+

.PHONY: all
```

This doesn't work as expected because the `.PHONY` declaration occurs too late. The proper way to write this example is to place the `.PHONY` declaration first.

```
# Phony targets must be declared before being used
.PHONY: all

all: hello
```

```
hello: hello_code.o hello_lib.o
    $(CC) $(CFLAGS) -o $@ $+
```

Phony targets are passed to subdirectories. As a practical matter, it is wise to declare all `.PHONY` targets in your root `OMakefile`, before any `.SUBDIRS`. This will ensure that 1) they are considered as phony targets in each of the subdirectories, and 2) you can build them from the project root.

```
.PHONY: all install clean

.SUBDIRS: src lib clib
```

Note that when a `.PHONY` target is inherited by a subdirectory via a `.SUBDIRS`, a whole hierarchy of `.PHONY` target (that are a part of the global one) is created, as described in Section 7.12.2 below.

7.12 Running OMake from a subdirectory

Running `omake foo` asks OMake to build the file `foo` in context of the *whole* project, even when running from a subdirectory of the project. Therefore, if `bar/baz` is a regular target (not a `.PHONY` one), then running `omake bar/baz` and running `(cd bar; omake baz)` are usually equivalent.

There are two noteworthy exceptions to the above rule:

- If the subdirectory is not a part of the project (there is no `.SUBDIRS`) for it, then OMake will complain if you try to run it in that directory.
- If a subdirectory contains an `OMakeroot` of its own, this would designate the subdirectory as a separate project (which is usually a bad idea and is not recommended).

7.12.1 Phony targets in a subdirectory

Suppose you have a `.PHONY: clean` declared in your root `OMakefile` and both the root `OMakefile` and the `OMakefile` in some of the subdirectories contain `clean:` rules. In this case

- Running `omake clean` in the root directory will execute all the rules (each in the appropriate directory);
- Running `omake clean` in the subdirectory will execute just its local one, as well as the ones from the subdirectories of the current directory.

The above equally applies to the built-in `.PHONY` targets, including `.DEFAULT`. Namely, if OMake is executed (without argument) in the root directory of a project, all the `.DEFAULT` targets in the project will be built. On the other hand, when OMake is executed (without argument) in a subdirectory, only the `.DEFAULT` targets defined in and under that subdirectory will be built.

The following Section explains the underlying semantics that gives rise to the above behavior.

7.12.2 Hierarchy of `.PHONY` targets

When the the root `OMakefile` contains a `.PHONY: clean` directive, it creates:

- A "global" phony target `/.PHONY/clean` (note the leading "`/`");
- A "relative" phony target attached to the current directory — `.PHONY/clean` (note the lack of the leading "`/`");
- A dependency `/.PHONY/clean: .PHONY/clean`.

All the `clean: ...` rules in the root `OMakefile` following this `.PHONY: clean` declaration would be interpreted as rules for the `.PHONY/clean` target.

Now when OMake then comes across a `.SUBDIRS: foo` directive (when it is in scope of the above `.PHONY: clean` declaration), it does the following:

- Creates a new `.PHONY/foo/clean` "relative" phony target;
- Creates the dependency `.PHONY/clean: .PHONY/foo/clean`;
- Processes the body of the `.SUBDIRS: foo` directive, or reads the `foo/OMakefile` file, if the body is empty. While doing that, it interprets its instructions relative to the `foo` directory. In particular, all the `clean: ...` rules will be taken to apply to `.PHONY/foo/clean`.

Now when you run `omake clean` in the root directory of the project, it is interpreted as `omake .PHONY/clean` (similar to how it happens with the normal targets), so both the rules for `.PHONY/clean` are executed and the rules for its dependency `.PHONY/foo/clean`. Running `(cd foo; omake clean)` is, as for normal targets, equivalent to running `omake .PHONY/foo/clean` and only those rules that apply to `.PHONY/foo/clean` will be executed.

7.13 Pathnames in rules

In rules, the targets and dependencies are first translated to *file* values (as in the `file` function). They are then translated to strings for the command line. This can cause some unexpected behavior. In the following example, the `absname` function is the absolute pathname for the file `a`, but the rule still prints the relative pathname.

```
.PHONY: demo
demo: $(absname a)
    echo $<

# omake demo
a
```

There is arguably a good reason for this. On Win32 systems, the `/` character is viewed as an "option specifier." The pathname separator is the `\` character. OMake translates the filenames automatically so that things work as expected on both systems.

```
demo: a/b
    echo $<

# omake demo (on a Unix system)
a/b
# omake demo (on a Win32 system)
a\b
```

Sometimes you may wish that target strings to be passed literally to the commands in the rule. One way to do this is to specify them literally.

```
SRC = a/b $(absname c/d)
demo: $(SRC)
    echo $(SRC)

# omake demo (on a Win32 system)
a/b c:\...\c\d
```

Alternately, you might wish that filenames be automatically expanded to absolute pathnames. For example, this might be useful when parsing the OMake output to look for errors. For this, you can use the `--absname` option (Section A.3.20). If you call `omake` with the `--absname` option, all filenames will be expanded to absolute names.

```
# omake --absname demo (on a Unix system)
/home/.../a/b /home/.../c/d
```

Alternately, the `--absname` option is scoped. If you want to use it for only a few rules, you can use the `OMakeFlags` function to control how it is applied.

```
section
   OMakeFlags(--absname)
   demo: a
       echo $<

# omake demo
/home/.../a
```

N.B. The `--absname` option is currently an experimental feature.

Chapter 8

Base library

8.1 Builtin variables

OMAKE_VERSION Version of OMake.

STDLIB The directory where the OMake standard library files reside. At startup, the default value is determined as follows.

- The value of the `OMAKELIB` environment variable, if set (must contain an absolute path, if set), otherwise
- On Windows, the registry keys `HKEY_CURRENT_USER\SOFTWARE\MetaPRL\OMake\OMAKELIB` and `HKEY_LOCAL_MACHINE\SOFTWARE\MetaPRL\OMake\OMAKELIB` are looked up and the value is used, if exist.
- Otherwise a compile-time default it used.

The current default value may be accessed by running `omake --version`

OMAKEPATH An array of directories specifying the lookup path for the `include` and `open` directives (see Section 4.7). The default value is an array of two elements — `.` and `$(STDLIB)`.

OSTYPE Set to the machine architecture `omake` is running on. Possible values are `Unix` (for all Unix versions, including Linux and Mac OS X), `Win32` (for MS-Windows, OMake compiled with MSVC++ or Mingw), and `Cygwin` (for MS-Windows, OMake compiled with Cygwin).

SYSNAME The name of the operating system for the current machine.

NODENAME The hostname of the current machine.

OS_VERSION The operating system release.

MACHINE The machine architecture, e.g. `i386`, `sparc`, etc.

HOST Same as `NODENAME`.

USER The login name of the user executing the process.

HOME The home directory of the user executing the process.

PID The OMake process id.

TARGETS The command-line target strings. For example, if OMake is invoked with the following command line,

```
omake CFLAGS=1 foo bar.c
```

then `TARGETS` is defined as `foo bar.c`.

BUILD_SUMMARY The `BUILD_SUMMARY` variable refers to the file that `omake` uses to summarize a build (the message that is printed at the very end of a build). The file is empty when the build starts. If you wish to add additional messages to the build summary, you can edit/modify this file during the build.

For example, if you want to point out that some action was taken, you can append a message to the build summary.

```
foo: boo
    echo "The file foo was built" >> $(BUILD_SUMMARY)
    ...build foo...
```

VERBOSE Whether certain commands should be verbose. A boolean flag that is `false` by default and is set to `true` when OMake is invoked with the `--verbose` option.

8.2 Logic, Boolean functions, and control flow

Boolean values in omake are represented by case-insensitive strings. The *false* value can be represented by the strings `false`, `no`, `nil`, `undefined` or `0`, and everything else is true.

8.2.1 not

```
$(not e) : String
   e : String
```

The `not` function negates a Boolean value.

For example, `$(not false)` expands to the string `true`, and `$(not hello world)` expands to `false`.

8.2.2 equal

```
$(equal e1, e2) : String
   e1 : String
   e2 : String
```

The `equal` function tests for equality of two values.

For example `$(equal a, b)` expands to `false`, and `$(equal hello world, hello world)` expands to `true`.

8.2.3 and

```
$(and e1, ..., en) : String
   e1, ..., en: Sequence
```

The `and` function evaluates to the conjunction of its arguments.

For example, in the following code, `X` is true, and `Y` is false.

```
A = a
B = b
X = $(and $(equal $(A), a) true $(equal $(B), b))
Y = $(and $(equal $(A), a) true $(equal $(A), $(B)))
```

8.2.4 or

```
$(or e1, ..., en) : String
   e1, ..., en: String Sequence
```

The `or` function evaluates to the disjunction of its arguments.

For example, in the following code, `X` is true, and `Y` is false.

```
A = a
B = b
X = $(or $(equal $(A), a) false $(equal $(A), $(B)))
Y = $(or $(equal $(A), $(B)) $(equal $(A), b))
```

8.2.5 if

```
 $(if e1, e2[, e3]) : value
    e1 : String
    e2, e3 : value
```

The `if` function represents a conditional based on a Boolean value. For example `$(if $(equal a, b), c, d)` evaluates to `d`.

Conditionals may also be declared with an alternate syntax.

```
if e1
   body1
elseif e2
   body2
...
else
   bodyn
```

If the expression `e1` is not false, then the expressions in `body1` are evaluated and the result is returned as the value of the conditional. Otherwise, if `e1` evaluates to false, the evaluation continues with the `e2` expression. If none of the conditional expressions is true, then the expressions in `bodyn` are evaluated and the result is returned as the value of the conditional.

There can be any number of `elseif` clauses; the `else` clause is optional.

Note that each branch of the conditional defines its own scope, so variables defined in the branches are normally not visible outside the conditional. The `export` command may be used to export the variables defined in a scope. For example, the following expression represents a common idiom for defining the C compiler configuration.

```
if $(equal $(OSTYPE), Win32)
   CC = cl
   CFLAGS += /DWIN32
   export
else
   CC = gcc
   CFLAGS += -g -O2
   export
```

8.2.6 switch, match

The `switch` and `match` functions perform pattern matching.

`$(switch <arg>, <pattern_1>, <value_1>, ..., <pattern_n>, <value_n>)`
`$(match <arg>, <pattern_1>, <value_1>, ..., <pattern_n>, <value_n>)`

The number of `<pattern>/<value>` pairs is arbitrary. They strictly alternate; the total number of arguments to `<match>` must be odd.

The `<arg>` is evaluated to a string, and compared with `<pattern_1>`. If it matches, the result of the expression is `<value_1>`. Otherwise evaluation continues with the remaining patterns until a match is found. If no pattern matches, the value is the empty string.

The `switch` function uses string comparison to compare the argument with the patterns. For example, the following expression defines the `FILE` variable to be either `foo`, `bar`, or the empty string, depending on the value of the `OSTYPE` variable.

```
FILE = $(switch $(OSTYPE), Win32, foo, Unix, bar)
```

The `match` function uses regular expression patterns (see the `grep` function). If a match is found, the variables `$1, $2, ...` are bound to the substrings matched between `\(` and `\)` delimiters. The `$0` variable contains the entire match, and `$*` is an array of the matched substrings. to the matched substrings.

```
FILE = $(match foo_xyz/bar.a, foo_\\\(.*\\\)/\\\(.*\\\)\.a, foo_$2/$1.o)
```

The `switch` and `match` functions also have an alternate (more usable) form.

```
match e
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

If the value of expression `e` matches `pattern_i` and no previous pattern, then `body_i` is evaluated and returned as the result of the `match`. The `switch` function uses string comparison; the `match` function uses regular expression matching.

```
match $(FILE)
case $".*\(\.[^\/.]*\)"
   println(The string $(FILE) has suffix $1)
default
   println(The string $(FILE) has no suffix)
```

8.2.7 try

```
try
   try-body
catch class1(v1)
   catch-body
when expr
   when-body
...
finally
   finally-body
```

The `try` form is used for exception handling. First, the expressions in the `try-body` are evaluated.

If evaluation results in a value `v` without raising an exception, then the expressions in the `finally-body` are evaluated and the value `v` is returned as the result.

If evaluation of the `try-body` results in a exception object `obj`, the `catch` clauses are examined in order. When examining `catch` clause `catch class(v)`, if the exception object `obj` is an instance of the class name `class`, the variable `v` is bound to the exception object, and the expressions in the `catch-body` are evaluated.

If a `when` clause is encountered while a `catch` body is being evaluated, the predicate `expr` is evaluated. If the result is true, evaluation continues with the expressions in the `when-body`. Otherwise, the next `catch` clause is considered for evaluation.

If evaluation of a `catch-body` or `when-body` completes successfully, returning a value `v`, without encountering another `when` clause, then the expressions in the `finally-body` are evaluated and the value `v` is returned as the result.

There can be any number of `catch` clauses; the `finally` clause is optional.

8.2.8 raise

```
raise exn
   exn : Exception
```

The `raise` function raises an exception. The `exn` object can be any object. However, the normal convention is to raise an `Exception` object.

If the exception is never caught, the whole object will be verbosely printed in the error message. However, if the object is an `Exception` one and contains a `message` field, only that field will be included in the error message.

8.2.9 exit

```
exit(code)
   code : Int
```

The `exit` function terminates `omake` abnormally.

`$(exit <code>)`

The `exit` function takes one integer argument, which is exit code. Non-zero values indicate abnormal termination.

8.2.10 defined

```
$(defined sequence) : String
   sequence : Sequence
```

The `defined` function test whether all the variables in the sequence are currently defined. For example, the following code defines the `X` variable if it is not already defined.

```
if $(not $(defined X))
   X = a b c
   export
```

8.2.11 defined-env

```
$(defined-env sequence) : String
   sequence : String
```

The `defined-env` function tests whether a variable is defined as part of the process environment.

For example, the following code adds the `-g` compile option if the environment variable `DEBUG` is defined.

```
if $(defined-env DEBUG)
    CFLAGS += -g
    export
```

8.2.12 getenv

```
$(getenv name) : String
$(getenv name, default) : String
```

The `getenv` function gets the value of a variable from the process environment. The function takes one or two arguments.

In the single argument form, an exception is raised if the variable variable is not defined in the environment. In the two-argument form, the second argument is returned as the result if the value is not defined.

For example, the following code defines the variable `X` to be a space-separated list of elements of the `PATH` environment variable if it is defined, and to `/bin /usr/bin` otherwise.

```
X = $(split $(PATHSEP), $(getenv PATH, /bin:/usr/bin))
```

You may also use the alternate form.

```
getenv(NAME)
    default
```

8.2.13 setenv

```
setenv(name, value)
   name : String
   value : String
```

The `setenv` function sets the value of a variable in the process environment. Environment variables are scoped like normal variables.

8.2.14 unsetenv

```
unsetenv(names)
   names : String Array
```

The `unsetenv` function removes some variable definitions from the process environment. Environment variables are scoped like normal variables.

8.2.15 get-registry

```
get-registry(hkey, key, field) : String
get-registry(hkey, key, field, default) : String
    hkey : String
    key : String
    field : String
```

The `get-registry` function retrieves a string value from the system registry on Win32. On other architectures, there is no registry.

The `hive` (I think that is the right word), indicates which part of the registry to use. It should be one of the following values.

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

Refer to the Microsoft documentation if you want to know what these mean.

The `key` is the field you want to get from the registry. It should have a form like `A\B\C` (if you use forward slashes, they will be converted to backslashes). The field is the sub-field of the key.

In the 4-argument form, the `default` is returned on failure. You may also use the alternate form.

```
get-registry(hkey, key, field)
   default
```

8.2.16 getvar

```
$(getvar name) : String
```

The `getvar` function gets the value of a variable.

An exception is raised if the variable variable is not defined.

For example, the following code defines X to be the string abc.

```
NAME = foo
foo_1 = abc
X = $(getvar $(NAME)_1)
```

8.2.17 setvar

```
setvar(name, value)
   name : String
   value : String
```

The `setvar` function defines a new variable. For example, the following code defines the variable `X` to be the string `abc`.

```
NAME = X
setvar($(NAME), abc)
```

8.3 Arrays and sequences

8.3.1 array

```
$(array elements) : Array
   elements : Sequence
```

The `array` function creates an array from a sequence. If the `<arg>` is a string, the elements of the array are the whitespace-separated elements of the string, respecting quotes.

In addition, array variables can be declared as follows.

```
A[] =
   <val1>
   ...
   <valn>
```

In this case, the elements of the array are exactly `<val1>`, ..., `<valn>`, and whitespace is preserved literally.

8.3.2 split

```
$(split sep, elements) : Array
   sep : String
   elements : Sequence
```

The `split` function takes two arguments, a string of separators, and a string argument. The result is an array of elements determined by splitting the elements by all occurrence of the separator in the `elements` sequence.

For example, in the following code, the `X` variable is defined to be the array `/bin /usr/bin /usr/local/bin`.

```
PATH = /bin:/usr/bin:/usr/local/bin
X = $(split :, $(PATH))
```

The `sep` argument may be omitted. In this case `split` breaks its arguments along the white space. Quotations are not split.

8.3.3 concat

```
$(concat sep, elements) : String
   sep : String
   elements : Sequence
```

The `concat` function takes two arguments, a separator string, and a sequence of elements. The result is a string formed by concatenating the elements, placing the separator between adjacent elements.

For example, in the following code, the `X` variable is defined to be the string `foo_x_bar_x_baz`.

```
X = foo  bar     baz
Y = $(concat _x_, $(X))
```

8.3.4 length

```
$(length sequence) : Int
   sequence : Sequence
```

The `length` function returns the number of elements in its argument.

For example, the expression `$(length a  b "c d")` evaluates to 3.

8.3.5 nth

```
$(nth i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

The `nth` function returns the nth element of its argument, treated as a list. Counting starts at 0. An exception is raised if the index is not in bounds.

For example, the expression `$(nth 1, a "b c" d)` evaluates to `"b c"`.

8.3.6 nth-hd

```
$(nth-hd i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

The `nth-hd` function returns the first `i` elements of the sequence. An exception is raised if the sequence is not at least `i` elements long.

For example, the expression `$(nth-hd 2, a "b c" d)` evaluates to `a "b c"`.

8.3.7 nth-tl

```
$(nth-tl i, sequence) : value
   i : Int
   sequence : Sequence
raises RuntimeException
```

The `nth-tl` function skips `i` elements of the sequence and returns the rest. An exception is raised if the sequence is not at least `i` elements long.

For example, the expression `$(nth-tl 1, a "b c" d)` evaluates to `"b c" d`.

8.3.8 subrange

```
$(subrange off, len, sequent) : value
   off : Int
   len : Int
   sequence : Sequence
raises RuntimeException
```

The `subrange` function returns a subrange of the sequence. Counting starts at 0. An exception is raised if the specified range is not in bounds.

For example, the expression `$(subrange 1, 2, a "b c" d e)` evaluates to `"b c" d`.

8.3.9 rev

```
$(rev sequence) : Sequence
   sequence : Sequence
```

The `rev` function returns the elements of a sequence in reverse order. For example, the expression `$(rev a "b c" d)` evaluates to `d "b c" a`.

8.3.10 join

```
$(join sequence1, sequence2) : Sequence
   sequence1 : Sequence
   sequence2 : Sequence
```

The `join` function joins together the elements of the two sequences. For example, `$(join a b c, .c .cpp .h)` evaluates to `a.c b.cpp c.h`. If the two input sequences have different lengths, the remainder of the longer sequence is copied at the end of the output unmodified.

8.3.11 string

```
$(string sequence) : String
   sequence : Sequence
```

The `string` function flattens a sequence into a single string. This is similar to the `concat` function, but the elements are separated by whitespace. The result is treated as a unit; whitespace is significant.

8.3.12 string-escaped, ocaml-escaped, html-escaped, html-pre-escaped, c-escaped, id-escaped

```
$(string-escaped sequence) : String Array
$(ocaml-escaped sequence) : String Array
$(html-escaped sequence) : String Array
$(html-pre-escaped sequence) : String Array
$(c-escaped sequence) : String Array
$(hex-escaped sequence) : StringArray
   sequence : Array
```

The `string-escaped` function converts each element of its argument to a string, escaping it, if it contains symbols that are special to OMake. The special characters include `:()\,$'"#` and whitespace. This function can be used in scanner rules to escape file names before printing then to `stdout`.

The `ocaml-escaped` function converts each element of its argument to a string, escaping characters that are special to OCaml.

The `c-escaped` function converts a string to a form that can be used as a string constant in C.

The `id-escaped` function turns a string into an identifier that may be used in OMake.

The `html-escaped` function turns a literal string into a form acceptable as HTML. The `html-pre-escaped` function is similar, but it does not translate newlines into `<br>`.

```
println($(string $(string-escaped $"a b" $"y:z")))
a\ b y\:z
```

8.3.13 decode-uri, encode-uri

```
$(decode-uri sequence) : sequence
    sequence : Sequence
```

These two functions perform URI encoding, where special characters are represented by hexadecimal characters.

```
osh> s = $(encode-uri $'a b~c')
"a+b%7ec"
osh> decode-uri($s)
"a b~c"
```

8.3.14 quote

```
$(quote sequence) : String
   sequence : Sequence
```

The `quote` function flattens a sequence into a single string and adds quotes around the string. Inner quotation symbols are escaped.

For example, the expression `$(quote a "b c" d)` evaluates to `"a \"b c\" d"`, and `$(quote abc)` evaluates to `"abc"`.

8.3.15 quote-argv

```
$(quote-argv sequence) : String
   sequence : Sequence
```

The `quote-argv` function flattens a sequence into a single string, and adds quotes around the string. The quotation is formed so that a command-line parse can separate the string back into its components.

8.3.16 html-string

```
$(html-string sequence) : String
   sequence : Sequence
```

The `html-string` function flattens a sequence into a single string, and escaped special HTML characters. This is similar to the `concat` function, but the elements are separated by whitespace. The result is treated as a unit; whitespace is significant.

8.3.17 addsuffix

```
$(addsuffix suffix, sequence) : Array
   suffix : String
   sequence : Sequence
```

The `addsuffix` function adds a suffix to each component of sequence. The number of elements in the array is exactly the same as the number of elements in the sequence.

For example, `$(addsuffix .c, a b "c d")` evaluates to `a.c b.c "c d".c`.

8.3.18 mapsuffix

```
$(mapsuffix suffix, sequence) : Array
   suffix : value
   sequence : Sequence
```

The `mapsuffix` function adds a suffix to each component of sequence. It is similar to `addsuffix`, but uses array concatenation instead of string concatenation. The number of elements in the array is twice the number of elements in the sequence.

For example, `$(mapsuffix .c, a b "c d")` evaluates to `a .c b .c "c d" .c`.

8.3.19 addsuffixes

```
$(addsuffixes suffixes, sequence) : Array
   suffixes : Sequence
   sequence : Sequence
```

The `addsuffixes` function adds all suffixes in its first argument to each component of a sequence. If `suffixes` has `n` elements, and `sequence` has `m` elements, the the result has `n * m` elements.

For example, the `$(addsuffixes .c .o, a b c)` expressions evaluates to `a.c a.o b.c b.o c.o c.a`.

8.3.20 removeprefix

```
$(removeprefix prefix, sequence) : Array
   prefix : String
   sequence : Array
```

The `removeprefix` function removes a prefix from each component of a sequence.

8.3.21 removesuffix

```
$(removesuffix sequence) : Array
   sequence : String
```

The `removesuffix` function removes the suffixes from each component of a sequence.

For example, `$(removesuffix a.c b.foo "c d")` expands to `a b "c d"`.

8.3.22 replacesuffixes

```
$(replacesuffixes old-suffixes, new-suffixes, sequence) : Array
   old-suffixes : Sequence
   new-suffixes : Sequence
   sequence : Sequence
```

The `replacesuffixes` function modifies the suffix of each component in sequence. The `old-suffixes` and `new-suffixes` sequences should have the same length.

For example, `$(replacesuffixes .h .c, .o .o, a.c b.h c.z)` expands to `a.o b.o c.z`.

8.3.23 addprefix

```
$(addprefix prefix, sequence) : Array
   prefix : String
   sequence : Sequence
```

The `addprefix` function adds a prefix to each component of a sequence. The number of element in the result array is exactly the same as the number of elements in the argument sequence.

For example, `$(addprefix foo/, a b "c d")` evaluates to `foo/a foo/b foo/"c d"`.

8.3.24 mapprefix

```
$(mapprefix prefix, sequence) : Array
   prefix : String
   sequence : Sequence
```

The `mapprefix` function adds a prefix to each component of a sequence. It is similar to `addprefix`, but array concatenation is used instead of string concatenation. The result array contains twice as many elements as the argument sequence.

For example, `$(mapprefix foo, a b "c d")` expands to `foo a foo b foo "c d"`.

8.3.25 add-wrapper

```
$(add-wrapper prefix, suffix, sequence) : Array
   prefix : String
   suffix : String
   sequence : Sequence
```

The `add-wrapper` functions adds both a prefix and a suffix to each component of a sequence. For example, the expression `$(add-wrapper dir/, .c, a b)` evaluates to `dir/a.c dir/b.c`. String concatenation is used. The array result has the same number of elements as the argument sequence.

8.3.26 set

```
$(set sequence) : Array
   sequence : Sequence
```

The `set` function sorts a set of string components, eliminating duplicates.

For example, `$(set z y z "m n" w a)` expands to `"m n" a w y z`.

8.3.27 mem

```
$(mem elem, sequence) : Boolean
   elem : String
   sequence : Sequence
```

The `mem` function tests for membership in a sequence.

For example, `$(mem "m n", y z "m n" w a)` evaluates to `true`, while `$(mem m n, y z "m n" w a)` evaluates to `false`.

8.3.28 intersection

```
$(intersection sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

The `intersection` function takes two arguments, treats them as sets of strings, and computes their intersection. The order of the result is undefined, and it may contain duplicates. Use the `set` function to sort the result and eliminate duplicates in the result if desired.

For example, the expression `$(intersection c a b a, b a)` evaluates to `a b a`.

8.3.29 intersects

```
$(intersects sequence1, sequence2) : Boolean
   sequence1 : Sequence
   sequence2 : Sequence
```

The `intersects` function tests whether two sets have a non-empty intersection. This is slightly more efficient than computing the intersection and testing whether it is empty.

For example, the expression `$(intersects a b c, d c e)` evaluates to `true`, and `$(intersects a b c a, d e f)` evaluates to `false`.

8.3.30 set-diff

```
$(set-diff sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

The `set-diff` function takes two arguments, treats them as sets of strings, and computes their difference (all the elements of the first set that are not present in the second one). The order of the result is undefined and it may contain duplicates. Use the `set` function to sort the result and eliminate duplicates in the result if desired.

For example, the expression `$(set-diff c a b a e, b a)` evaluates to `c e`.

8.3.31 filter

```
$(filter patterns, sequence) : Array
   patterns : Sequence
   sequence : Sequence
```

The `filter` function picks elements from a sequence. The patterns is a non-empty sequence of patterns, each may contain one occurrence of the wildcard `%` character.

For example `$(filter %.h %.o, a.c x.o b.h y.o "hello world".c)` evaluates to `x.o b.h y.o`.

8.3.32 filter-out

```
$(filter-out patterns, sequence) : Array
   patterns : Sequence
   sequence : Sequence
```

The `filter-out` function removes elements from a sequence. The patterns is a non-empty sequence of patterns, each may contain one occurrence of the wildcard `%` character.

For example `$(filter-out %.c %.h, a.c x.o b.h y.o "hello world".c)` evaluates to `x.o y.o`.

8.3.33 capitalize

```
$(capitalize sequence) : Array
   sequence : Sequence
```

The `capitalize` function capitalizes each word in a sequence. For example, `$(capitalize through the looking Glass)` evaluates to `Through The Looking Glass`.

8.3.34 uncapitalize

```
$(uncapitalize sequence) : Array
   sequence : Sequence
```

The `uncapitalize` function uncapitalizes each word in its argument.

For example, `$(uncapitalize through the looking Glass)` evaluates to `through the looking glass`.

8.3.35 uppercase

```
$(uppercase sequence) : Array
   sequence : Sequence
```

The `uppercase` function converts each word in a sequence to uppercase. For example, `$(uppercase through the looking Glass)` evaluates to `THROUGH THE LOOKING GLASS`.

8.3.36 lowercase

```
$(lowercase sequence) : Array
   sequence : Sequence
```

The `lowercase` function reduces each word in its argument to lowercase.

For example, `$(lowercase through tHe looking Glass)` evaluates to `through the looking glass`.

8.3.37 system

```
system(s)
   s : Sequence
```

The `system` function is used to evaluate a shell expression. This function is used internally by `omake` to evaluate shell commands.

For example, the following program is equivalent to the expression `system(ls foo)`.

```
ls foo
```

8.3.38 shell

```
$(shell command) : Array
$(shella command) : Array
$(shell-code command) : Int
   command : Sequence
```

The `shell` function evaluates a command using the command shell, and returns the whitespace-separated words of the standard output as the result.

The `shella` function acts similarly, but it returns the lines as separate items in the array.

The `shell-code` function returns the exit code. The output is not diverted.

For example, if the current directory contains the files `OMakeroot`, `OMakefile`, and `hello.c`, then `$(shell ls)` evaluates to `hello.c OMakefile OMakeroot` (on a Unix system).

8.3.39 export

The `export` function allows one to capture the current environment in a variable.

For example, the following code:

```
A = 1
B = 1
C = 1
SAVE_ENV = $(export A B)
A = 2
B = 2
C = 2
export $(SAVE_ENV)
println($A $B $C)
```

will print `1 1 2`.

The arguments to this function are interpreted the exact same way as the arguments to the `export` special form (see Section 5.3).

8.3.40 while

```
while <test>
   <body>
```

–or–

```
while <test>
case <test1>
   <body1>
...
case <testn>
   <bodyn>
default
   <bodyd>
```

The loop is executed while the test is true. In the first form, the `<body>` is executed on every loop iteration. In the second form, the body `<bodyI>` is selected, as the first case where the test `<testI>` is true. If none apply, the optional default case is evaluated. If no cases are true, the loop exits. The environment is automatically exported.

Examples.

Iterate for `i` from `0` to `9`.

```
i = 0
while $(lt $i, 10)
   echo $i
   i = $(add $i, 1)
```

The following example is equivalent.

```
i = 0
while true
case $(lt $i, 10)
   echo $i
   i = $(add $i, 1)
```

The following example is similar, but some special cases are printed. value is printed.

```
i = 0
while $(lt $i, 10)
case $(equal $i, 0)
   echo zero
```

```
    case $(equal $i, 1)
       echo one
    default
       echo $i
```

The `break` function can be used to break out of the `while` loop early.

8.3.41 break

```
break
```

Terminate execution of the innermost loop, returning the current state.

8.3.42 random, random-init

```
random-init(i)
    i : Int
random() : Int
```

Produce a random number. The numbers are pseudo-random, and are not cryptographically secure.

The generator is initialized form semi-random system data. Subsequent runs should produce different results. The `rando-init` function can be used to return the generator to a known state.

8.4 Arithmetic

8.4.1 int

The `int` function can be used to create integers. It returns an `Int` object.
`$(int 17)`.

8.4.2 float

The `float` function can be used to create floating-point numbers. It returns a `Float` object.
`$(float 3.1415926)`.

8.4.3 Basic arithmetic

The following functions can be used to perform basic arithmetic.

- `$(neg <numbers>)`: arithmetic inverse
- `$(add <numbers>)`: addition.
- `$(sub <numbers>)`: subtraction.

- `$(mul <numbers>)`: multiplication.
- `$(div <numbers>)`: division.
- `$(mod <numbers>)`: remainder.
- `$(lnot <numbers>)`: bitwise inverse.
- `$(land <numbers>)`: bitwise and.
- `$(lor <numbers>)`: bitwise or.
- `$(lxor <numbers>)`: bitwise exclusive-or.
- `$(lsl <numbers>)`: logical shift left.
- `$(lsr <numbers>)`: logical shift right.
- `$(asr <numbers>)`: arithmetic shift right.
- `$(min <numbers>)`: smallest element.
- `$(max <numbers>)`: largest element.

8.4.4 Comparisons

The following functions can be used to perform numerical comparisons.

- `$(lt <numbers>)`: less then.
- `$(le <numbers>)`: no more than.
- `$(eq <numbers>)`: equal.
- `$(ge <numbers>)`: no less than.
- `$(gt <numbers>)`: greater than.
- `$(ult <numbers>)`: unsigned less than.
- `$(ule <numbers>)`: unsigned greater than.
- `$(uge <numbers>)`: unsigned greater than or equal.
- `$(ugt <numbers>)`: unsigned greater than.

8.5 First-class functions

8.5.1 fun

The `fun` form introduces anonymous functions.

`$(fun <v1>, ..., <vn>, <body>)`

The last argument is the body of the function. The other arguments are the parameter names.

The three following definitions are equivalent.

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))

F = $(fun X, Y, $(addsuffix $(Y), $(X)))

F =
   fun(X, Y)
      value $(addsuffix $(Y), $(X))
```

8.5.2 apply

The `apply` operator is used to apply a function.

`$(apply <fun>, <args>)`

Suppose we have the following function definition.

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))
```

The the two expressions below are equivalent.

```
X = F(a b c, .c)
X = $(apply $(F), a b c, .c)
```

8.5.3 applya

The `applya` operator is used to apply a function to an array of arguments.

`$(applya <fun>, <args>)`

For example, in the following program, the value of `Z` is `file.c`.

```
F(X, Y) =
   return($(addsuffix $(Y), $(X)))
args[] =
   file
   .c
Z = $(applya $(F), $(args))
```

8.5.4 create-map, create-lazy-map

The `create-map` is a simplified form for creating `Map` objects. The `create-map` function takes an even number of arguments that specify key/value pairs. For example, the following values are equivalent.

```
X = $(create-map name1, xxx, name2, yyy)

X. =
    extends $(Map)
    $|name1| = xxx
    $|name2| = yyy
```

The `create-lazy-map` function is similar, but the values are computed lazily. The following two definitions are equivalent.

```
Y = $(create-lazy-map name1, $(xxx), name2, $(yyy))

Y. =
    extends $(Map)
    $|name1| = $`(xxx)
    $|name2| = $`(yyy)
```

The `create-lazy-map` function is used in rule construction.

8.6 Iteration and mapping

8.6.1 foreach

The `foreach` function maps a function over a sequence.

```
$(foreach <fun>, <args>)

foreach(<var>, <args>)
   <body>
```

For example, the following program defines the variable `X` as an array `a.c b.c c.c`.

```
X =
   foreach(x, a b c)
      value $(x).c

# Equivalent expression
X = $(foreach $(fun x, $(x).c), abc)
```

There is also an abbreviated syntax.

The `export` form can also be used in a `foreach` body. The final value of `X` is `a.c b.c c.c`.

```
X =
foreach(x, a b c)
   X += $(x).c
   export
```

The `break` function can be used to break out of the loop early.

Chapter 9

File, I/O and system operations

9.1 File names

9.1.1 file, dir

```
$(file sequence) : File Sequence
   sequence : Sequence
$(dir sequence) : Dir Sequence
   sequence : Sequence
```

The `file` and `dir` functions define location-independent references to files and directories. In `omake`, the commands to build a target are executed in the target's directory. Since there may be many directories in an `omake` project, the build system provides a way to construct a reference to a file in one directory, and use it in another without explicitly modifying the file name. The functions have the following syntax, where the name should refer to a file or directory.

For example, we can construct a reference to a file `foo` in the current directory.

```
FOO = $(file foo)
.SUBDIRS: bar
```

If the `FOO` variable is expanded in the `bar` subdirectory, it will expand to `../foo`.

These commands are often used in the top-level OMakefile to provide location-independent references to top-level directories, so that build commands may refer to these directories as if they were absolute.

```
ROOT = $(dir .)
LIB  = $(dir lib)
BIN  = $(dir bin)
```

Once these variables are defined, they can be used in build commands in subdirectories as follows, where `$(BIN)` will expand to the location of the `bin` directory relative to the command being executed.

```
   install: hello
cp hello $(BIN)
```

9.1.2 tmpfile

```
$(tmpfile prefix) : File
$(tmpfile prefix, suffix) : File
    prefix : String
    suffix : String
```

The `tmpfile` function returns the name of a fresh temporary file in the temporary directory.

9.1.3 in

```
$(in dir, exp) : String Array
   dir : Dir
   exp : expression
```

The `in` function is closely related to the `dir` and `file` functions. It takes a directory and an expression, and evaluates the expression in that effective directory. For example, one common way to install a file is to define a symbol link, where the value of the link is relative to the directory where the link is created.

The following commands create links in the `$(LIB)` directory.

```
FOO = $(file foo)
install:
   ln -s $(in $(LIB), $(FOO)) $(LIB)/foo
```

Note that the `in` function only affects the expansion of `Node` (`File` and `Dir`) values.

9.1.4 basename

```
$(basename files) : String Sequence
   files : String Sequence
```

The `basename` function returns the base names for a list of files. The basename is the filename with any leading directory components removed.

For example, the expression `$(basename dir1/dir2/a.out /etc/modules.conf /foo.ml)` evaluates to `a.out modules.conf foo.ml`.

9.1.5 dirname

```
$(dirname files) : String Sequence
   files : String Sequence
```

The `dirname` function returns the directory name for a list of files. The directory name is the filename with the basename removed. If a name does not have a directory part, the directory is “.”

For example, the expression `$(dirname dir1\dir2\a.out /etc/modules.conf /foo.ml bar.ml)` evaluates to `dir1/dir2 /etc / .`.

Note: this function is different from the `dirof` function. The function `dirname` is simple a function over strings, while `dirof` is a function on filenames.

9.1.6 rootname

```
$(rootname files) : String Sequence
   files : String Sequence
```

The `rootname` function returns the root name for a list of files. The rootname is the filename with the final suffix removed.

For example, the expression `$(rootname dir1/dir2/a.out /etc/a.b.c /foo.ml)` evaluates to `dir1/dir2/a /etc/a.b /foo`.

9.1.7 dirof

```
$(dirof files) : Dir Sequence
   files : File Sequence
```

The `dirof` function returns the directory for each of the listed files.

For example, the expression `$(dirof dir/dir2/a.out /etc/modules.conf /foo.ml)` evaluates to the directories `dir1/dir2 /etc /`.

9.1.8 fullname

```
$(fullname files) : String Sequence
   files : File Sequence
```

The `fullname` function returns the pathname relative to the project root for each of the files or directories.

9.1.9 absname

```
$(absname files) : String Sequence
   files : File Sequence
```

The `absname` function returns the absolute pathname for each of the files or directories.

9.1.10 homename

```
$(homename files) : String Sequence
   files : File Sequence
```

The `homename` function returns the name of a file in tilde form, if possible. The unexpanded forms are computed lazily: the `homename` function will usually evaluate to an absolute pathname until the first tilde-expansion for the same directory.

9.1.11 suffix

```
$(suffix files) : String Sequence
   files : StringSequence
```

The `suffix` function returns the suffixes for a list of files. If a file has no suffix, the function returns the empty string.

For example, the expression `$(suffix dir1/dir2/a.out /etc/a /foo.ml)` evaluates to `.out .ml`.

9.2 Path search

9.2.1 which

```
$(which files) : File Sequence
   files : String Sequence
```

The `which` function searches for executables in the current command search path, and returns `file` values for each of the commands. It is an error if a command is not found.

9.2.2 where

The `where` function is similar to which, except it returns the list of all the locations of the given executable (in the order in which the corresponding directories appear in `$PATH`). In case a command is handled internally by the `Shell` object, the first string in the output will describe the command as a built-in function.

```
% where echo
echo is a Shell object method (a built-in function)
/bin/echo
```

9.2.3 rehash

```
rehash()
```

The `rehash` function resets all search paths.

9.2.4 exists-in-path

```
$(exists-in-path files) : String
   files : String Sequence
```

The `exists-in-path` function tests whether all executables are present in the current search path.

9.2.5 digest

```
$(digest files) : String Array
   file : File Array
raises RuntimeException

$(digest-optional files) : String Array
   file : File Array
```

The `digest` and `digest-optional` functions compute MD5 digests of files. The `digest` function raises an exception if a file does no exist. The `digest-optional` returns `false` if a file does no exist. MD5 digests are cached.

9.2.6 find-in-path

```
$(find-in-path path, files) : File Array
   path : Dir Array
   files : String Array
raises RuntimeException

$(find-in-path-optional path, files) : File Array
```

The `find-in-path` function searches for the files in a search path. Only the tail of the filename is significant. The `find-in-path` function raises an exception if the file can't be found. The `find-in-path-optional` function silently removes files that can't be found.

9.2.7 digest-path

```
$(digest-in-path path, files) : String/File Array
   path : Dir Array
   files : String Array
raises RuntimeException

$(digest-in-path-optional path, files) : String/File Array
```

The `digest-in-path` function searches for the files in a search path and returns the file and digest for each file. Only the tail of the filename is significant. The `digest-in-path` function raises an exception if the file can't be found. The `digest-in-path-optional` function silently removes elements that can't be found.

9.3 File stats

9.3.1 file-exists, target-exists, target-is-proper

```
$(file-exists files) : String
$(target-exists files) : String
$(target-is-proper files) : String
    files : File Sequence
```

The `file-exists` function checks whether the files listed exist. The `target-exists` function is similar to the `file-exists` function. However, it returns true if the file exists *or* if it can be built by the current project. The `target-is-proper` returns true only if the file can be generated in the current project.

9.3.2 stat-reset

```
$(stat-reset files) : String
    files : File Sequence
```

OMake uses a stat-cache. The `stat-reset` function reset the `stat` information for the given files, forcing the `stat` information to be recomputed the next time it is requested.

9.3.3 filter-exists, filter-targets, filter-proper-targets

```
$(filter-exists files) : File Sequence
$(filter-targets files) : File Sequence
$(filter-proper-targets) : File Sequence
   files : File Sequence
```

The `filter-exists`, `filter-targets`, and `filter-proper-targets` functions remove files from a list of files.

- `filter-exists`: the result is the list of files that exist.
- `filter-targets`: the result is the list of files either exist, or can be built by the current project.
- `filter-proper-targets`: the result is the list of files that can be built in the current project.

One way to create a simple "clean" rule that removes generated files from the project is by removing all files that can be built in the current project. CAUTION: you should be careful before you do this. The rule removes *any* file that can *potentially* be reconstructed. There is no check to make sure that the commands to rebuild the file would actually succeed. Also, note that no file outside the current project will be deleted.

```
.PHONY: clean

clean:
    rm $(filter-proper-targets $(ls R, .))
```

See the `dependencies-proper` function to see an alternate method for removing intermediate files.

If you use CVS, you may wish to use the `cvs_realclean` program that is distributed with `omake`.

9.3.4 find-targets-in-path, find-targets-in-path-optional

```
$(find-targets-in-path path files) : File Array
$(find-targets-in-path-optional path, files) : File Array
    path : Dir Array
    files : File Sequence
```

The `find-target-in-path` function searches for targets in the search path. For each file `file` in the file list, the path is searched sequentially for a directory `dir` such that the target `dir/file` exists. If so, the file `dir/file` is returned.

For example, suppose you are building a C project, and project contains a subdirectory `src/` containing only the files `fee.c` and `foo.c`. The following expression evaluates to the files `src/fee.o` `src/foo.o` even if the files have not already been built.

```
$(find-targets-in-path lib src, fee.o foo.o)

# Evaluates to
src/fee.o src/foo.o
```

The `find-targets-in-path` function raises an exception if the file can't be found. The `find-targets-in-path-optional` function silently removes targets that can't be found.

```
$(find-targets-in-path-optional lib src, fee.o foo.o fum.o)

# Evaluates to
src/fee.o src/foo.o
```

9.3.5 file-sort

```
$(file-sort order, files) : File Sequence
  order : String
  files : File Sequence
```

The `file-sort` function sorts a list of filenames by build order augmented by a set of sort rules. Sort rules are declared using the `.ORDER` target. The `.BUILDORDER` defines the default order.

`$(file-sort <order>, <files>)`

For example, suppose we have the following set of rules.

```
a: b c
b: d
c: d

.DEFAULT: a b c d
   echo $(file-sort .BUILDORDER, a b c d)
```

In the case, the sorter produces the result `d b c a`. That is, a target is sorted *after* its dependencies. The sorter is frequently used to sort files that are to be linked by their dependencies (for languages where this matters).

There are three important restrictions to the sorter:

- The sorter can be used only within a rule body. The reason for this is that *all* dependencies must be known before the sort is performed.
- The sorter can only sort files that are buildable in the current project.
- The sorter will fail if the dependencies are cyclic.

9.3.5.1 sort rule

It is possible to further constrain the sorter through the use of sort rules. A sort rule is declared in two steps. The target must be listed as an `.ORDER` target; and then a set of sort rules must be given. A sort rule defines a pattern constraint.

```
.ORDER: .MYORDER

.MYORDER: %.foo: %.bar
.MYORDER: %.bar: %.baz

.DEFAULT: a.foo b.bar c.baz d.baz
   echo $(sort .MYORDER, a.foo b.bar c.baz d.baz)
```

In this example, the `.MYORDER` sort rule specifies that any file with a suffix `.foo` should be placed after any file with suffix `.bar`, and any file with suffix `.bar` should be placed after a file with suffix `.baz`.

In this example, the result of the sort is `d.baz c.baz b.bar a.foo`.

9.3.6 file-check-sort

```
file-check-sort(files)
   files : File Sequence
raises RuntimeException
```

The `file-check-sort` function checks whether a list of files is in sort order. If so, the list is returned unchanged. If not, the function raises an exception.

`$(file-check-sort <order>, <files>)`

9.4 Globbing and file listings

OMake commands are "glob-expanded" before being executed. That is, names may contain *patterns* that are expanded to sequences of file and directory names. The syntax follows the standard bash(1), csh(1), syntax, with the following rules.

- A *pathname* is a sequence of directory and file names separated by one of the `/` or `\` characters. For example, the following pathnames refer to the same file: `/home/jyh/OMakefile` and `/home\jyh/OMakefile`.
- Glob-expansion is performed on the components of a path. If a path contains occurrences of special characters (listed below), the path is viewed as a pattern to be matched against the actual files in the system. The expansion produces a sequence of all file/directory names that match.

  For the following examples, suppose that a directory `/dir` contains files named `a`, `-a`, `a.b`, and `b.c`.

  `*` Matches any sequence of zero-or-more characters. For example, the pattern `/dir/a*` expands to `/dir/a /dir/aa /dir/a.b`.

  `?` Matches exactly one character. The pattern `/dir/?a` expands the filename `/dir/-a`.

  `[...]` Square brackets denote character sets and ranges in the ASCII character set. The pattern may contain individual characters c or character ranges c_1-c_2. The pattern matches any of the individual characters specified, or any characters in the range. A leading "hat" inverts the send of the pattern. To specify a pattern that contains the literal characters `-`, the `-` should occur as the first character in the range.

Pattern	Expansion
`/dir/[a-b]*`	`/dir/a /dir/a.b /dir/b.c`
`/dir/[-a-b]*`	`/dir/a /dir/-a /dir/a.b /dir/b.c`
`/dir/[-a]*`	`/dir/a /dir/-a /dir/a.b`

  {`s1,...,sN`} Braces indicate brace-expansion. The braces delimit a sequence of strings separated by commas. Given N strings, the result produces N copies of the pattern, one for each of the strings s_i.

Pattern	Expansion
`a{b,c,d}`	`ab ac ad`
`a{b{c,d},e}`	`abc abd ae`
`a{?{[A-Z],d},*}`	`a?[A-Z] a?d a*`

  The tilde is used to specify home directories. Depending on your system, these might be possible expansions.

Pattern	Expansion
`~jyh`	`/home/jyh`
`~bob/*.c`	`c:\Documents and Settings\users\bob`

The `\` character is both a pathname separator and an escape character. If followed by a special glob character, the `\` changes the sense of the following character to non-special status. Otherwise, `\` is viewed as a pathname separator.

Pattern	Expansion
`~jyh/\*`	`~jyh/*` (`*` is literal)
`/dir/\[a-z?`	`/dir/[a-z?` (`[` is literal, `?` is a pattern).
`c:\Program Files\[A-z]`	`c:\Program Files[A-z]*`

Note that the final case might be considered to be ambiguous (where `\` should be viewed as a pathname separator, not as an escape for the subsequent `[` character. If you want to avoid this ambiguity on Win32, you should use the forward slash `/` even for Win32 pathnames (the `/` is translated to `\` in the output).

Pattern	Expansion
`c:/Program Files/[A-z]*`	`c:\Program Files\WindowsUpdate ...`

9.4.1 glob

```
$(glob strings) : Node Array
   strings : String Sequence
$(glob options, strings) : Node Array
   options : String
   strings : String Sequence
```

The `glob` function performs glob-expansion.
The . and .. entries are always ignored.
The options are:

b Do not perform `csh`(1)-style brace expansion.

e The `\` character does not escape special characters.

n If an expansion fails, return the expansion literally instead of aborting.

i If an expansion fails, it expands to nothing.

. Allow wildcard patterns to match files beginning with a .

A Return all files, including files that begin with a .

F Match only normal files (any file that is not a directory).

D Match only directory files.

C Ignore files according to `cvs`(1) rules.

P Include only proper subdirectories.

In addition, the following variables may be defined that affect the behavior of `glob`.

GLOB_OPTIONS A string containing default options.

GLOB_IGNORE A list of shell patterns for filenames that `glob` should ignore.

GLOB_ALLOW A list of shell patterns. If a file does not match a pattern in `GLOB_ALLOW`, it is ignored.

The returned files are sorted by name.

9.4.2 ls

```
$(ls files) : Node Array
   files : String Sequence
$(ls options, files) : Node Array
   files : String Sequence
```

The `ls` function returns the filenames in a directory.
The . and .. entries are always ignored. The patterns are shell-style patterns, and are glob-expanded.
The options include all of the options to the `glob` function, plus the following.

R Perform a recursive listing.

The `GLOB_ALLOW` and `GLOB_IGNORE` variables can be defined to control the globbing behavior. The returned files are sorted by name.

9.4.3 subdirs

```
$(subdirs dirs) : Dir Array
   dirs : String Sequence
$(subdirs options, dirs) : Dir Array
   options : String
   dirs : String Sequence
```

The `subdirs` function returns all the subdirectories of a list of directories, recursively.
The possible options are the following:

A Return directories that begin with a .

C Ignore files according to `.cvsignore` rules.

P Include only proper subdirectories.

9.5 Filesystem operations

9.5.1 mkdir

```
mkdir(mode, node...)
   mode : Int
   node : Node
raises RuntimeException

mkdir(node...)
   node : Node
raises RuntimeException
```

The `mkdir` function creates a directory, or a set of directories. The following options are supported.

-m mode Specify the permissions of the created directory.

-p Create parent directories if they do not exist.

– Interpret the remaining names literally.

9.5.2 Stat

The `Stat` object represents an information about a filesystem node, as returned by the `stat` and `lstat` functions. It contains the following fields.

dev : the device number.

ino : the inode number.

kind : the kind of the file, one of the following: `REG` (regular file), `DIR` (directory), `CHR` (character device), `BLK` (block device), `LNK` (symbolic link), `FIFO` (named pipe), `SOCK` (socket).

perm : access rights, represented as an integer.

nlink : number of links.

uid : user id of the owner.

gid : group id of the file's group.

rdev : device minor number.

size : size in bytes.

atime : last access time, as a floating point number.

mtime : last modification time, as a floating point number.

ctime : last status change time, as a floating point number.

Not all of the fields will have meaning on all operating systems.

9.5.3 stat, lstat

```
$(stat node...) : Stat
   node : Node or Channel
$(lstat node...) : Stat
   node : Node or Channel
raises RuntimeException
```

The `stat` functions return file information. If the file is a symbolic link, the `stat` function refers to the destination of the link; the `lstat` function refers to the link itself.

9.5.4 unlink

```
$(unlink file...)
   file : File
#(rm file...)
   file : File
$(rmdir dir...)
   dir : Dir
raises RuntimeException
```

The `unlink` and `rm` functions remove a file. The `rmdir` function removes a directory.

The following options are supported for `rm` and `rmdir`.

-f ignore nonexistent files, never prompt.

-i prompt before removal.

-r remove the contents of directories recursively.

-v explain what is going on.

– the rest of the values are interpreted literally.

9.5.5 rename

```
rename(old, new)
   old : Node
   new : Node
mv(nodes... dir)
   nodes : Node Sequence
   dir   : Dir
cp(nodes... dir)
   nodes : Node Sequence
   dir   : Dir
raises RuntimeException
```

The `rename` function changes the name of a file or directory named `old` to `new`.

The `mv` function is similar, but if `new` is a directory, and it exists, then the files specified by the sequence are moved into the directory. If not, the behavior of `mv` is identical to `rename`. The `cp` function is similar, but the original file is not removed.

The `mv` and `cp` functions take the following options.

-f Do not prompt before overwriting.

-i Prompt before overwriting.

-v Explain what it happening.

-r Copy the contents of directories recursively.

– Interpret the remaining arguments literally.

9.5.6 link

```
link(src, dst)
   src : Node
   dst : Node
raises RuntimeException
```

The `link` function creates a hard link named `dst` to the file or directory `src`.
Hard links may work under Win32 when NTFS is used.
Normally, only the superuser can create hard links to directories.

9.5.7 symlink

```
symlink(src, dst)
   src : Node
   dst : Node
raises RuntimeException
```

The `symlink` function creates a symbolic link `dst` that points to the `src` file.

The link name is computed relative to the target directory. For example, the expression `$(symlink a/b, c/d)` creates a link named `c/d -> ../a/b`.

Symbolic links are not supported in Win32. Consider using the `ln-or-cp` `Shell` alias for cross-platform portable linking/copying.

9.5.8 readlink

```
$(readlink node...) : Node
   node : Node
```

The `readlink` function reads the value of a symbolic link.

9.5.9 chmod

```
chmod(mode, dst...)
   mode : Int
   dst : Node or Channel
chmod(mode dst...)
   mode : String
   dst : Node Sequence
raises RuntimeException
```

The `chmod` function changes the permissions of the targets.
Options:

-v Explain what is happening.

-r Change files and directories recursively.

-f Continue on errors.

– Interpret the remaining argument literally.

9.5.10 chown

```
chown(uid, gid, node...)
   uid : Int
   gid : Int
   node : Node or Channel
chown(uid, node...)
   uid : Int
   node : Node or Channel
raises RuntimeException
```

The `chown` function changes the user and group id of the file. If the `gid` is not specified, it is not changed. If either id is -1, that id is not changed.

9.5.11 truncate

```
truncate(length, node...)
    length : Int
    node : Node or Channel
raises RuntimeException
```

The `truncate` function truncates a file to the given length.

9.5.12 umask

```
$(umask mode) : Int
   mode : Int
raises RuntimeException
```

Sets the file mode creation mask. The previous mask is returned. This value is not scoped, changes have global effect.

9.6 vmount

9.6.1 vmount

```
vmount(src, dst)
   src, dst : Dir
vmount(flags, src, dst)
   flags : String
   src, dst : Dir
```

"Mount" the `src` directory on the `dst` directory. This is a virtual mount, changing the behavior of the `$(file ...)` function. When the `$(file str)` function is used, the resulting file is taken relative to the `src` directory if the file exists. Otherwise, the file is relative to the current directory.

The main purpose of the `vmount` function is to support multiple builds with separate configurations or architectures.

The options are as follows.

l Create symbolic links to files in the `src` directory.

c Copy files from the `src` directory.

Mount operations are scoped.

9.6.2 add-project-directories

```
add-project-directories(dirs)
   dirs : Dir Array
```

Add the directories to the set of directories that omake considers to be part of the project. This is mainly used to avoid omake complaining that the current directory is not part of the project.

9.6.3 remove-project-directories

```
remove-project-directories(dirs)
   dirs : Dir Array
```

Removed the directories from the set of directories that omake considers to be part of the project. This is mainly used to cancel a `.SUBDIRS` from including a directory if it is determined that the directory does not need to be compiled.

9.7 File predicates

9.7.1 test

```
test(exp) : Bool
   exp : String Sequence
```

The *expression* grammar is as follows:

- `!` *expression* : *expression* is not true
- *expression1* `-a` *expression2* : both expressions are true
- *expression1* `-o` *expression2* : at least one expression is true
- `(` *expression* `)` : *expression* is true

The base expressions are:

- `-n` *string* : The *string* has nonzero length
- `-z` *string* : The *string* has zero length
- *string* `=` *string* : The strings are equal
- *string* `!=` *string* : The strings are not equal
- *int1* `-eq` *int2* : The integers are equal
- *int1* `-ne` *int2* : The integers are not equal
- *int1* `-gt` *int2* : *int1* is larger than *int2*
- *int1* `-ge` *int2* : *int2* is not larger than *int1*
- *int1* `-lt` *int2* : *int1* is smaller than *int2*
- *int1* `-le` *int2* : *int1* is not larger than *int2*
- *file1* `-ef` *file2* : On Unix, *file1* and *file2* have the same device and inode number. On Win32, *file1* and *file2* have the same name.
- *file1* `-nt` *file2* : *file1* is newer than *file2*
- *file1* `-ot` *file2* : *file1* is older than *file2*
- `-b` *file* : The file is a block special file
- `-c` *file* : The file is a character special file
- `-d` *file* : The file is a directory
- `-e` *file* : The file exists

- `-f` *file* : The file is a normal file
- `-g` *file* : The set-`group-id` bit is set on the file
- `-G` *file* : The file's group is the current effective group
- `-h` *file* : The file is a symbolic link (also `-L`)
- `-k` *file* : The file's sticky bit is set
- `-L` *file* : The file is a symbolic link (also `-h`)
- `-O` *file* : The file's owner is the current effective user
- `-p` *file* : The file is a named pipe
- `-r` *file* : The file is readable
- `-s` *file* : The file is empty
- `-S` *file* : The file is a socket
- `-u` *file* : The set-`user-id` bit is set on the file
- `-w` *file* : The file is writable
- `-x` *file* : The file is executable

A *string* is any sequence of characters; leading `-` characters are allowed.

An *int* is a *string* that can be interpreted as an integer. Unlike traditional versions of the test program, the leading characters may specify an arity. The prefix `0b` means the numbers is in binary; the prefix `0o` means the number is in octal; the prefix `0x` means the number is in hexadecimal. An *int* can also be specified as `-l` *string*, which evaluates to the length of the *string*.

A *file* is a *string* that represents the name of a file.

The syntax mirrors that of the `test`(1) program. If you are on a Unix system, the man page explains more. Here are some examples.

```
# Create an empty file
osh> touch foo
# Is the file empty?
osh> test(-e foo)
- : true
osh> test(! -e foo)
- : false
# Create another file
osh> touch boo
# Is the newer file newer?
osh> test(boo -nt foo)
- : true
# A more complex query
```

```
# boo is newer than foo, and foo is empty
osh> test(\( boo -nt foo \) -a -e foo)
- : true
```

9.7.2 find

```
find(exp) : Node Array
   exp : String Sequence
```

The `find` function searches a directory recursively, returning the files for which the expression evaluates to true.

The expression argument uses the same syntax as the `test` function, with the following exceptions.

1. The expression may begin with a directory. If not specified, the current directory is searched.
2. The `{}` string expands to the current file being examined.

The syntax of the expression is the same as `test`, with the following additions.

- `-name` *string* : The current file matches the glob expression (see Section 9.4).

The `find` function performs a recursive scan of all subdirectories. The following call is being run from the root of the `omake` source directory.

```
osh> find(. -name fo* )
- : <array
        /home/jyh/.../omake/mk/.svn/format
        /home/jyh/.../omake/RPM/.svn/format
        ...
        /home/jyh/.../omake/osx_resources/installer_files/.svn/format>
```

Another example, listing only those files that are normal files or symbolic links.

```
osh> find(. -name fo* -a \( -f {} -o -L {} \))
- : <array
        /home/jyh/.../omake/mk/.svn/format
        /home/jyh/.../omake/RPM/.svn/format
        ...
        /home/jyh/.../omake/osx_resources/installer_files/.svn/format>
```

9.8 IO functions

9.8.1 Standard channels

The following variables define the standard channels.

stdin

`stdin : InChannel`

The standard input channel, open for reading.

stdout

`stdout : OutChannel`

The standard output channel, open for writing.

stderr

`stderr : OutChannel`

The standard error channel, open for writing.

9.8.2 open-in-string

The `open-in-string` treats a string as if it were a file and returns a channel for reading.

```
$(open-in-string s) : Channel
    s : String
```

9.8.3 open-out-string, out-string

The `open-out-string` creates a channel that writes to a string instead of a file. The string may be retrieved with the `out-string` function.

```
$(open-out-string) : Channel
$(out-string chan) : String
    chan : OutChannel
```

9.8.4 fopen

The `fopen` function opens a file for reading or writing.

```
$(fopen file, mode) : Channel
   file : File
   mode : String
```

The `file` is the name of the file to be opened. The `mode` is a combination of the following characters.

r Open the file for reading; it is an error if the file does not exist.

w Open the file for writing; the file is created if it does not exist.

a Open the file in append mode; the file is created if it does not exist.

+ Open the file for both reading an writing.

t Open the file in text mode (default).

b Open the file in binary mode.

n Open the file in nonblocking mode.

x Fail if the file already exists.

Binary mode is not significant on Unix systems, where text and binary modes are equivalent.

9.8.5 close

```
 $(close channel...)
    channel : Channel
```

The `close` function closes a file that was previously opened with `fopen`.

9.8.6 read

```
$(read channel, amount) : String
   channel : InChannel
   amount  : Int
raises RuntimeException
```

The `read` function reads up to `amount` bytes from an input channel, and returns the data that was read. If an end-of-file condition is reached, the function raises a `RuntimeException` exception.

9.8.7 write

```
$(write channel, buffer, offset, amount) : String
   channel : OutChannel
   buffer  : String
   offset  : Int
   amount  : Int
$(write channel, buffer) : String
   channel : OutChannel
   buffer  : String
raises RuntimeException
```

In the 4-argument form, the `write` function writes bytes to the output channel `channel` from the `buffer`, starting at position `offset`. Up to `amount` bytes are written. The function returns the number of bytes that were written.

The 3-argument form is similar, but the `offset` is 0.

In the 2-argument form, the `offset` is 0, and the `amount` if the length of the `buffer`.

If an end-of-file condition is reached, the function raises a `RuntimeException` exception.

9.8.8 lseek

```
$(lseek channel, offset, whence) : Int
   channel : Channel
   offset  : Int
   whence  : String
raises RuntimeException
```

The `lseek` function repositions the offset of the channel `channel` according to the `whence` directive, as follows:

SEEK_SET The offset is set to `offset`.

SEEK_CUR The offset is set to its current position plus `offset` bytes.

SEEK_END The offset is set to the size of the file plus `offset` bytes.

The `lseek` function returns the new position in the file.

9.8.9 rewind

```
rewind(channel...)
   channel : Channel
```

The `rewind` function set the current file position to the beginning of the file.

9.8.10 tell

```
$(tell channel...) : Int...
   channel : Channel
raises RuntimeException
```

The `tell` function returns the current position of the `channel`.

9.8.11 flush

```
$(flush channel...)
   channel : OutChannel
```

The `flush` function can be used only on files that are open for writing. It flushes all pending data to the file.

9.8.12 dup

```
$(dup channel) : Channel
   channel : Channel
raises RuntimeException
```

The `dup` function returns a new channel referencing the same file as the argument.

9.8.13 dup2

```
dup2(channel1, channel2)
   channel1 : Channel
   channel2 : Channel
raises RuntimeException
```

The `dup2` function causes `channel2` to refer to the same file as `channel1`.

9.8.14 set-nonblock

```
set-nonblock-mode(mode, channel...)
   channel : Channel
   mode : String
```

The `set-nonblock-mode` function sets the nonblocking flag on the given channel. When IO is performed on the channel, and the operation cannot be completed immediately, the operations raises a `RuntimeException`.

9.8.15 set-close-on-exec-mode

```
set-close-on-exec-mode(mode, channel...)
   channel : Channel
   mode : String
raises RuntimeException
```

The `set-close-on-exec-mode` function sets the close-on-exec flags for the given channels. If the close-on-exec flag is set, the channel is not inherited by child processes. Otherwise it is.

9.8.16 pipe

```
$(pipe) : Pipe
raises RuntimeException
```

The `pipe` function creates a `Pipe` object, which has two fields. The `read` field is a channel that is opened for reading, and the `write` field is a channel that is opened for writing.

9.8.17 mkfifo

```
mkfifo(mode, node...)
   mode : Int
   node : Node
```

The `mkfifo` function creates a named pipe.

9.8.18 select

```
$(select rfd..., wfd..., wfd..., timeout) : Select
   rfd : InChannel
   wfd : OutChannel
   efd : Channel
   timeout : float
raises RuntimeException
```

The `select` function polls for possible IO on a set of channels. The `rfd` are a sequence of channels for reading, `wfd` are a sequence of channels for writing, and `efd` are a sequence of channels to poll for error conditions. The `timeout` specifies the maximum amount of time to wait for events.

On successful return, `select` returns a `Select` object, which has the following fields:

read An array of channels available for reading.

write An array of channels available for writing.

error An array of channels on which an error has occurred.

9.8.19 lockf

```
lockf(channel, command, len)
   channel : Channel
   command : String
   len : Int
raises RuntimeException
```

The `lockf` function places a lock on a region of the channel. The region starts at the current position and extends for `len` bytes.

The possible values for `command` are the following.

F_ULOCK Unlock a region.

F_LOCK Lock a region for writing; block if already locked.

F_TLOCK Lock a region for writing; fail if already locked.

F_TEST Test a region for other locks.

F_RLOCK Lock a region for reading; block if already locked.

F_TRLOCK Lock a region for reading; fail is already locked.

9.8.20 InetAddr

The `InetAddr` object describes an Internet address. It contains the following fields.

addr `String`: the Internet address.

port `Int`: the port number.

9.8.21 Host

A `Host` object contains the following fields.

name `String`: the name of the host.

aliases `String Array`: other names by which the host is known.

addrtype `String`: the preferred socket domain.

addrs `InetAddr Array`: an array of Internet addresses belonging to the host.

9.8.22 gethostbyname

```
$(gethostbyname host...) : Host...
   host : String
raises RuntimeException
```

The `gethostbyname` function returns a `Host` object for the specified host. The `host` may specify a domain name or an Internet address.

9.8.23 Protocol

The `Protocol` object represents a protocol entry. It has the following fields.

name `String`: the canonical name of the protocol.

aliases `String Array`: aliases for the protocol.

proto `Int`: the protocol number.

9.8.24 getprotobyname

```
$(getprotobyname name...) : Protocol...
   name : Int or String
raises RuntimeException
```

The `getprotobyname` function returns a `Protocol` object for the specified protocol. The `name` may be a protocol name, or a protocol number.

9.8.25 Service

The `Service` object represents a network service. It has the following fields.

name `String`: the name of the service.

aliases `String Array`: aliases for the service.

port `Int`: the port number of the service.

proto `Protocol`: the protocol for the service.

9.8.26 getservbyname

```
$(getservbyname service...) : Service...
   service : String or Int
raises RuntimeException
```

The `getservbyname` function gets the information for a network service. The `service` may be specified as a service name or number.

9.8.27 socket

```
$(socket domain, type, protocol) : Channel
   domain : String
   type : String
   protocol : String
raises RuntimeException
```

The `socket` function creates an unbound socket.
The possible values for the arguments are as follows.
The `domain` may have the following values.

PF_UNIX or unix Unix domain, available only on Unix systems.

PF_INET or inet Internet domain, IPv4.

PF_INET6 or inet6 Internet domain, IPv6.

The `type` may have the following values.

SOCK_STREAM or stream Stream socket.

SOCK_DGRAM or dgram Datagram socket.

SOCK_RAW or raw Raw socket.

SOCK_SEQPACKET or seqpacket Sequenced packets socket

The `protocol` is an `Int` or `String` that specifies a protocol in the protocols database.

9.8.28 bind

```
bind(socket, host, port)
   socket : InOutChannel
   host : String
   port : Int
bind(socket, file)
   socket : InOutChannel
   file : File
raise RuntimeException
```

The `bind` function binds a socket to an address.

The 3-argument form specifies an Internet connection, the `host` specifies a host name or IP address, and the `port` is a port number.

The 2-argument form is for `Unix` sockets. The `file` specifies the filename for the address.

9.8.29 listen

```
listen(socket, requests)
   socket : InOutChannel
   requests : Int
raises RuntimeException
```

The `listen` function sets up the socket for receiving up to `requests` number of pending connection requests.

9.8.30 accept

```
$(accept socket) : InOutChannel
   socket : InOutChannel
raises RuntimeException
```

The `accept` function accepts a connection on a socket.

9.8.31 connect

```
connect(socket, addr, port)
   socket : InOutChannel
   addr : String
   port : int
connect(socket, name)
   socket : InOutChannel
   name : File
raise RuntimeException
```

The `connect` function connects a socket to a remote address.

The 3-argument form specifies an Internet connection. The `addr` argument is the Internet address of the remote host, specified as a domain name or IP address. The `port` argument is the port number.

The 2-argument form is for Unix sockets. The `name` argument is the filename of the socket.

9.8.32 getchar

```
$(getc) : String
$(getc file) : String
   file : InChannel or File
raises RuntimeException
```

The `getc` function returns the next character of a file. If the argument is not specified, `stdin` is used as input. If the end of file has been reached, the function returns `false`.

9.8.33 gets

```
$(gets) : String
$(gets channel) : String
   channel : InChannel or File
raises RuntimeException
```

The `gets` function returns the next line from a file. The function returns the empty string if the end of file has been reached. The line terminator is removed.

9.8.34 fgets

```
$(fgets) : String
$(fgets channel) : String
   channel : InChannel or File
raises RuntimeException
```

The `fgets` function returns the next line from a file that has been opened for reading with `fopen`. The function returns the empty string if the end of file has been reached. The returned string is returned as literal data. The line terminator is not removed.

9.9 Printing functions

Output is printed with the `print` and `println` functions. The `println` function adds a terminating newline to the value being printed, the `print` function does not.

```
fprint(<file>, <string>)
print(<string>)
eprint(<string>)
fprintln(<file>, <string>)
println(<string>)
eprintln(<string>)
```

The `fprint` functions print to a file that has been previously opened with `fopen`. The `print` functions print to the standard output channel, and the `eprint` functions print to the standard error channel.

9.10 Value printing functions

Values can be printed with the `printv` and `printvln` functions. The `printvln` function adds a terminating newline to the value being printed, the `printv` function does not.

```
fprintv(<file>, <string>)
printv(<string>)
eprintv(<string>)
fprintvln(<file>, <string>)
printvln(<string>)
eprintvln(<string>)
```

The `fprintv` functions print to a file that has been previously opened with `fopen`. The `printv` functions print to the standard output channel, and the `eprintv` functions print to the standard error channel.

9.11 Higher-level IO functions

9.11.1 Regular expressions

Many of the higher-level functions use regular expressions. Regular expressions are defined by strings with syntax nearly identical to `awk`(1).

Strings may contain the following character constants.

- `\\` : a literal backslash.
- `\a` : the alert character `^G`.
- `\b` : the backspace character `^H`.
- `\f` : the formfeed character `^L`.
- `\n` : the newline character `^J`.
- `\r` : the carriage return character `^M`.

- `\t` : the tab character `^I`.
- `\v` : the vertical tab character.
- `\xhh...` : the character represented by the string of hexadecimal digits `h`. All valid hexadecimal digits following the sequence are considered to be part of the sequence.
- `\ddd` : the character represented by 1, 2, or 3 octal digits.

Regular expressions are defined using the special characters `.\^$[(){}*?+`.

- `c` : matches the literal character `c` if `c` is not a special character.
- `\c` : matches the literal character `c`, even if `c` is a special character.
- `.` : matches any character, including newline.
- `^` : matches the beginning of a line.
- `$` : matches the end of line.
- `[abc...]` : matches any of the characters `abc...`
- `[^abc...]` : matches any character except `abc...`
- `r1|r2` : matches either `r1` or `r2`.
- `r1r2` : matches `r1` and then `r2`.
- `r+` : matches one or more occurrences of `r`.
- `r*` : matches zero or more occurrences of `r`.
- `r?` : matches zero or one occurrence of `r`.
- `(r)` : parentheses are used for grouping; matches `r`.
- `\(r\)` : also defines grouping, but the expression matched within the parentheses is available to the output processor through one of the variables `$1`, `$2`, ...
- `r{n}` : matches exactly `n` occurrences of `r`.
- `r{n,}` : matches `n` or more occurrences of `r`.
- `r{n,m}` : matches at least `n` occurrences of `r`, and no more than `m` occurrences.
- `\y`: matches the empty string at either the beginning or end of a word.
- `\B`: matches the empty string within a word.
- `\<`: matches the empty string at the beginning of a word.

- `\>`: matches the empty string at the end of a word.
- `\w`: matches any character in a word.
- `\W`: matches any character that does not occur within a word.
- ``\` ``: matches the empty string at the beginning of a file.
- `\'`: matches the empty string at the end of a file.

Character classes can be used to specify character sequences abstractly. Some of these sequences can change depending on your LOCALE.

- `[:alnum:]` Alphanumeric characters.
- `[:alpha:]` Alphabetic characters.
- `[:lower:]` Lowercase alphabetic characters.
- `[:upper:]` Uppercase alphabetic characters.
- `[:cntrl:]` Control characters.
- `[:digit:]` Numeric characters.
- `[:xdigit:]` Numeric and hexadecimal characters.
- `[:graph:]` Characters that are printable and visible.
- `[:print:]` Characters that are printable, whether they are visible or not.
- `[:punct:]` Punctuation characters.
- `[:blank:]` Space or tab characters.
- `[:space:]` Whitespace characters.

9.11.2 cat

```
cat(files) : Sequence
   files : File or InChannel Sequence
```

The `cat` function concatenates the output from multiple files and returns it as a string.

9.11.3 grep

```
grep(pattern) : String  # input from stdin, default options
   pattern : String
grep(pattern, files) : String  # default options
   pattern : String
   files   : File Sequence
grep(options, pattern, files) : String
  options : String
  pattern : String
  files   : File Sequence
```

The `grep` function searches for occurrences of a regular expression `pattern` in a set of files, and prints lines that match. This is like a highly-simplified version of `grep`(1).

The options are:

q If specified, the output from `grep` is not displayed.

h If specified, output lines will not include the filename (default, when only one input file is given).

n If specified, output lines include the filename (default, when more than one input file is given).

v If specified, search for lines without a match instead of lines with a match,

The `pattern` is a regular expression.

If successful (`grep` found a match), the function returns `true`. Otherwise, it returns `false`.

9.11.4 scan

```
scan(input-files)
case string1
   body1
case string2
   body2
...
default
   bodyd
```

The `scan` function provides input processing in command-line form. The function takes file/filename arguments. If called with no arguments, the input is taken from `stdin`. If arguments are provided, each specifies an `InChannel`, or the name of a file for input. Output is always to `stdout`.

The `scan` function operates by reading the input one line at a time, and processing it according to the following algorithm.

For each line, the record is first split into fields, and the fields are bound to the variables `$1, $2, ...`. The variable `$0` is defined to be the entire line, and `$*` is an array of all the field values. The `$(NF)` variable is defined to be the number of fields.

Next, a case expression is selected. If `string_i` matches the token `$1`, then `body_i` is evaluated. If the body ends in an `export`, the state is passed to the next clause. Otherwise the value is discarded.

For example, here is an `scan` function that acts as a simple command processor.

```
calc() =
   i = 0
   scan(script.in)
   case print
      println($i)
   case inc
      i = $(add $i, 1)
      export
   case dec
      i = $(sub $i, 1)
      export
   case addconst
      i = $(add $i, $2)
      export
   default
      eprintln($"Unknown command: $1")
```

The `scan` function also supports several options.

```
scan(options, files)
...
```

A Parse each line as an argument list, where arguments may be quoted. For example, the following line has three words, "`ls`", "`-l`", "`Program Files`".

```
ls -l "Program Files"
```

O Parse each line using white space as the separator, using the usual OMake algorithm for string parsing. This is the default.

x Once each line is split, reduce each word using the hex representation. This is the usual hex representation used in URL specifiers, so the string "Program Files" may be alternately represented in the form Program-Program+Files.

Note, if you want to redirect the output to a file, the easiest way is to redefine the `stdout` variable. The `stdout` variable is scoped the same way as other variables, so this definition does not affect the meaning of `stdout` outside the `calc` function.

```
calc() =
    stdout = $(fopen script.out, w)
    scan(script.in)
       ...
    close(stdout)
```

9.11.5 awk

```
awk(input-files)
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

or

```
awk(options, input-files)
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

The `awk` function provides input processing similar to `awk`(1), but more limited. The `input-files` argument is a sequence of values, each specifies an `InChannel`, or the name of a file for input. If called with no options and no file arguments, the input is taken from `stdin`. Output is always to `stdout`.

The variables `RS` and `FS` define record and field separators as regular expressions. The default value of `RS` is the regular expression `\r|\n|\r\n`. The default value of `FS` is the regular expression `[ \t]`+.

The `awk` function operates by reading the input one record at a time, and processing it according to the following algorithm.

For each line, the record is first split into fields using the field separator `FS`, and the fields are bound to the variables `$1, $2, ...`. The variable `$0` is defined to be the entire line, and `$*` is an array of all the field values. The `$(NF)` variable is defined to be the number of fields.

Next, the cases are evaluated in order. For each case, if the regular expression `pattern_i` matches the record `$0`, then `body_i` is evaluated. If the body ends in an `export`, the state is passed to the next clause. Otherwise the value is discarded. If the regular expression contains `\(r\)` expression, those expression override the fields `$1, $2, ...`.

For example, here is an `awk` function to print the text between two delimiters `\begin{<name>}` and `\end{<name>}`, where the `<name>` must belong to a set passed as an argument to the `filter` function.

```
filter(names) =
   print = false

   awk(Awk.in)
   case $"^\\end\{\([:alpha:]+\)\}"
      if $(mem $1, $(names))
         print = false
         export
      export
   default
      if $(print)
         println($0)
   case $"^\\begin\{\([:alpha:]+\)\}"
      print = $(mem $1, $(names))
      export
```

Note, if you want to redirect the output to a file, the easiest way is to redefine the `stdout` variable. The `stdout` variable is scoped the same way as other variables, so this definition does not affect the meaning of `stdout` outside the `filter` function.

```
filter(names) =
    stdout = $(fopen file.out, w)
    awk(Awk.in)
       ...
    close(stdout)
```

Options.

b "Break" when evaluating cases. Only the first case that matches will be selected.

The `break` function can be used to abort the loop, exiting the `awk` function immediately.

9.11.6 fsubst

```
fsubst(files)
case pattern1 [options]
```

```
   body1
case pattern2 [options]
   body2
...
default
   bodyd
```

The `fsubst` function provides a `sed`(1)-like substitution function. Similar to `awk`, if `fsubst` is called with no arguments, the input is taken from `stdin`. If arguments are provided, each specifies an `InChannel`, or the name of a file for input.

The `RS` variable defines a regular expression that determines a record separator, The default value of `RS` is the regular expression `\r|\n|\r\n`.

The `fsubst` function reads the file one record at a time.

For each record, the cases are evaluated in order. Each case defines a substitution from a substring matching the `pattern` to replacement text defined by the body.

Currently, there is only one option: `g`. If specified, each clause specifies a global replacement, and all instances of the pattern define a substitution. Otherwise, the substitution is applied only once.

Output can be redirected by redefining the `stdout` variable.

For example, the following program replaces all occurrences of an expression `word.` with its capitalized form.

```
section
   stdout = $(fopen Subst.out, w)
   fsubst(Subst.in)
   case $"\<\([[:alnum:]]+\)\." g
      value $(capitalize $1).
   close(stdout)
```

9.11.7 lex

```
lex(files)
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

The `lex` function provides a simple lexical-style scanner function. The input is a sequence of files or channels. The cases specify regular expressions. Each time the input is read, the regular expression that matches the *longest prefix* of the input is selected, and the body is evaluated.

If two clauses both match the same input, the *last* one is selected for execution. The `default` case matches the regular expression `.`; you probably want to place it first in the pattern list.

If the body end with an `export` directive, the state is passed to the next clause.

For example, the following program collects all occurrences of alphanumeric words in an input file.

```
collect-words($(files)) =
   words[] =
   lex($(files))
   default
      # empty
   case $"[[:alnum:]]+" g
      words[] += $0
      export
```

The `default` case, if one exists, matches single characters. Since

It is an error if the input does not match any of the regular expressions.

The `break` function can be used to abort the loop.

9.11.8 lex-search

```
lex-search(files)
case pattern1
   body1
case pattern2
   body2
...
default
   bodyd
```

The `lex-search` function is like the `lex` function, but input that does not match any of the regular expressions is skipped. If the clauses include a `default` case, then the `default` matches any skipped text.

For example, the following program collects all occurrences of alphanumeric words in an input file, skipping any other text.

```
collect-words($(files)) =
   words[] =
   lex-search($(files))
   default
      eprintln(Skipped $0)
   case $"[[:alnum:]]+" g
      words[] += $0
      export
```

The `default` case, if one exists, matches single characters. Since
It is an error if the input does not match any of the regular expressions.
The `break` function can be used to abort the loop.

9.11.9 Lexer

The `Lexer` object defines a facility for lexical analysis, similar to the `lex`(1) and `flex`(1) programs.

In `omake`, lexical analyzers can be constructed dynamically by extending the `Lexer` class. A lexer definition consists of a set of directives specified with method calls, and set of clauses specified as rules.

For example, consider the following lexer definition, which is intended for lexical analysis of simple arithmetic expressions for a desktop calculator.

```
lexer1. =
   extends $(Lexer)

   other: .
      eprintln(Illegal character: $* )
      lex()

   white: $"[[:space:]]+"
      lex()

   op: $"[-+*/()]"
      switch $*
      case +
         Token.unit($(loc), plus)
      case -
         Token.unit($(loc), minus)
      case *
         Token.unit($(loc), mul)
      case /
         Token.unit($(loc), div)
      case $"("
         Token.unit($(loc), lparen)
      case $")"
         Token.unit($(loc), rparen)

   number: $"[[:digit:]]+"
      Token.pair($(loc), exp, $(int $* ))

   eof: $"\'"
      Token.unit($(loc), eof)
```

This program defines an object `lexer1` the extends the `Lexer` object, which defines lexing environment.

The remainder of the definition consists of a set of clauses, each with a method name before the colon; a regular expression after the colon; and in this case, a body. The body is optional, if it is not specified, the method with the given name should already exist in the lexer definition.

NB The clause that matches the *longest* prefix of the input is selected. If two clauses match the same input prefix, then the *last* one is selected. This is unlike most standard lexers, but makes more sense for extensible grammars.

The first clause matches any input that is not matched by the other clauses. In this case, an error message is printed for any unknown character, and the input is skipped. Note that this clause is selected only if no other clause matches.

The second clause is responsible for ignoring white space. If whitespace is found, it is ignored, and the lexer is called recursively.

The third clause is responsible for the arithmetic operators. It makes use of the `Token` object, which defines three fields: a `loc` field that represents the source location; a `name`; and a `value`.

The lexer defines the `loc` variable to be the location of the current lexeme in each of the method bodies, so we can use that value to create the tokens.

The `Token.unit($(loc), name)` method constructs a new `Token` object with the given name, and a default value.

The `number` clause matches nonnegative integer constants. The `Token.pair($(loc), name, value)` constructs a token with the given name and value.

Lexer object operate on `InChannel` objects. The method `lexer1.lex-channel(channel)` reads the next token from the channel argument.

9.11.10 Lexer matching

During lexical analysis, clauses are selected by longest match. That is, the clause that matches the longest sequence of input characters is chosen for evaluation. If no clause matches, the lexer raises a `RuntimeException`. If more than one clause matches the same amount of input, the first one is chosen for evaluation.

9.11.11 Extending lexer definitions

Suppose we wish to augment the lexer example so that it ignores comments. We will define comments as any text that begins with the string `(*`, ends with `*)`, and comments may be nested.

One convenient way to do this is to define a separate lexer just to skip comments.

```
lex-comment. =
   extends $(Lexer)

   level = 0

   other: .
      lex()
```

```
   term: $"[*][)]"
      if $(not $(eq $(level), 0))
         level = $(sub $(level), 1)
         lex()

   next: $"[(][*]"
      level = $(add $(level), 1)
      lex()

   eof: $"\'"
      eprintln(Unterminated comment)
```

This lexer contains a field `level` that keeps track of the nesting level. On encountering a `(*` string, it increments the level, and for `*)`, it decrements the level if nonzero, and continues.

Next, we need to modify our previous lexer to skip comments. We can do this by extending the lexer object `lexer1` that we just created.

```
lexer1. +=
   comment: $"[(][*]"
      lex-comment.lex-channel($(channel))
      lex()
```

The body for the comment clause calls the `lex-comment` lexer when a comment is encountered, and continues lexing when that lexer returns.

9.11.12 Threading the lexer object

Clause bodies may also end with an `export` directive. In this case the lexer object itself is used as the returned token. If used with the `Parser` object below, the lexer should define the `loc`, `name` and `value` fields in each `export` clause. Each time the `Parser` calls the lexer, it calls it with the lexer returned from the previous lex invocation.

9.11.13 Parser

The `Parser` object provides a facility for syntactic analysis based on context-free grammars.

`Parser` objects are specified as a sequence of directives, specified with method calls; and productions, specified as rules.

For example, let's finish building the desktop calculator started in the `Lexer` example.

```
parser1. =
   extends $(Parser)
```

```
#
# Use the main lexer
#
lexer = $(lexer1)

#
# Precedences, in ascending order
#
left(plus minus)
left(mul div)
right(uminus)

#
# A program
#
start(prog)

prog: exp eof
   return $1

#
# Simple arithmetic expressions
#
exp: minus exp :prec: uminus
   neg($2)

exp: exp plus exp
   add($1, $3)

exp: exp minus exp
   sub($1, $3)

exp: exp mul exp
   mul($1, $3)

exp: exp div exp
   div($1, $3)

exp: lparen exp rparen
   return $2
```

Parsers are defined as extensions of the `Parser` class. A `Parser` object must have a `lexer` field. The `lexer` is not required to be a `Lexer` object, but it must provide a `lexer.lex()` method that returns a token object with `name` and `value` fields. For this example, we use the `lexer1` object that we defined previously.

The next step is to define precedences for the terminal symbols. The precedences are defined with the `left`, `right`, and `nonassoc` methods in order of increasing precedence.

The grammar must have at least one start symbol, declared with the `start` method.

Next, the productions in the grammar are listed as rules. The name of the production is listed before the colon, and a sequence of variables is listed to the right of the colon. The body is a semantic action to be evaluated when the production is recognized as part of the input.

In this example, these are the productions for the arithmetic expressions recognized by the desktop calculator. The semantic action performs the calculation. The variables `$1, $2, ...` correspond to the values associated with each of the variables on the right-hand-side of the production.

9.11.14 Calling the parser

The parser is called with the `$(parser1.parse-channel start, channel)` or `$(parser1.parse-file start, file)` functions. The `start` argument is the start symbol, and the `channel` or `file` is the input to the parser.

9.11.15 Parsing control

The parser generator generates a pushdown automation based on LALR(1) tables. As usual, if the grammar is ambiguous, this may generate shift/reduce or reduce/reduce conflicts. These conflicts are printed to standard output when the automaton is generated.

By default, the automaton is not constructed until the parser is first used.

The `build(debug)` method forces the construction of the automaton. While not required, it is wise to finish each complete parser with a call to the `build(debug)` method. If the `debug` variable is set, this also prints with parser table together with any conflicts.

The `loc` variable is defined within action bodies, and represents the input range for all tokens on the right-hand-side of the production.

9.11.16 Extending parsers

Parsers may also be extended by inheritance. For example, let's extend the grammar so that it also recognizes the `<<` and `>>` shift operations.

First, we extend the lexer so that it recognizes these tokens. This time, we choose to leave `lexer1` intact, instead of using the += operator.

```
lexer2. =
   extends $(lexer1)

   lsl: $"<<"
      Token.unit($(loc), lsl)
```

```
    asr: $">>"
       Token.unit($(loc), asr)
```

Next, we extend the parser to handle these new operators. We intend that the bitwise operators have lower precedence than the other arithmetic operators. The two-argument form of the `left` method accomplishes this.

```
parser2. =
   extends $(parser1)

   left(plus, lsl lsr asr)

   lexer = $(lexer2)

   exp: exp lsl exp
      lsl($1, $3)

   exp: exp asr exp
      asr($1, $3)
```

In this case, we use the new lexer `lexer2`, and we add productions for the new shift operations.

9.11.17 Passwd

The `Passwd` object represents an entry in the system's user database. It contains the following fields.

`pw_name`: the login name.

`pw_passwd`: the encrypted password.

`pw_uid`: user id of the user.

`pw_gid`: group id of the user.

`pw_gecos`: the user name or comment field.

`pw_dir`: the user's home directory.

`pw_shell`: the user's default shell.

Not all the fields will have meaning on all operating systems.

9.11.18 getpwnam, getpwuid

```
$(getpwnam name...) : Passwd
   name : String
$(getpwuid uid...) : Passwd
   uid : Int
raises RuntimeException
```

The `getpwnam` function looks up an entry by the user's login and the `getpwuid` function looks up an entry by user's numerical id (uid). If no entry is found, an exception will be raised.

9.11.19 getpwents

```
$(getpwents) : Array
```

The `getpwents` function returns an array of `Passwd` objects, one for every user fund in the system user database. Note that depending on the operating system and on the setup of the user database, the returned array may be incomplete or even empty.

9.11.20 Group

The `Group` object represents an entry in the system's user group database. It contains the following fields.

`gr_name`: the group name.

`gr_group`: the encrypted password.

`gr_gid`: group id of the group.

`gr_mem`: the group member's user names.

Not all the fields will have meaning on all operating systems.

9.11.21 getgrnam, getgrgid

```
$(getgrnam name...) : Group
   name : String
$(getgrgid gid...) : Group
   gid : Int
raises RuntimeException
```

The `getgrnam` function looks up a group entry by the group's name and the `getgrgid` function looks up an entry by groups's numerical id (gid). If no entry is found, an exception will be raised.

9.11.22 tgetstr

```
$(tgetstr id) : String
   id : String
```

The `tgetstr` function looks up the terminal capability with the indicated `id`. This assumes the terminfo to lookup is given in the `TERM` environment variable. This function returns an empty value if the given terminal capability is not defined.

Note: if you intend to use the value returned by `tgetstr` inside the shell `prompt`, you need to wrap it using the `prompt-invisible` function.

9.11.23 xterm-escape-begin, xterm-escape-end

```
$(xterm-escape-begin) : String
$(xterm-escape-end) : String
```

The `xterm-escape-begin` and `xterm-escape-end` functions return the escape sequences that can be used to set the XTerm window title. Will return empty values if this capability is not available.

Note: if you intend to use these strings inside the shell `prompt`, you need to use `$(prompt_invisible_begin)$(xterm-escape-begin)` and `$(xterm-escape-end)$(prompt_invisible_end)`.

9.11.24 xterm-escape

```
$(xterm-escape s) : Sequence
```

When the `TERM` environment variable indicates that the XTerm title setting capability is available, `$(xterm-escape s)` is equivalent to `$(xterm-escape-begin)s$(xterm-escape-end)`. Otherwise, it returns an empty value.

Note: if you intend to use the value returned by `xterm-escape` inside the shell `prompt`, you need to wrap it using the `prompt-invisible` function.

9.11.25 prompt-invisible-begin, prompt-invisible-end

```
$(prompt-invisible-begin) : String
$(prompt-invisible-end) : String
```

The `prompt-invisible-begin` and `prompt-invisible-end` functions return the escape sequences that must used to mark the "invisible" sections of the shell `prompt` (such as various escape sequences).

9.11.26 prompt-invisible

```
$(prompt-invisible s) : Sequence
```

The `prompt-invisible` will wrap its argument with `$(prompt-invisible-begin)` and `$(prompt-invisible-end)`. All the 'invisible" sections of the shell `prompt` (such as various escape sequences) must be wrapped this way.

9.11.27 gettimeofday

```
$(gettimeofday) : Float
```

The `gettimeofday` function returns the time of day in seconds since January 1, 1970.

Chapter 10

Shell commands

Shell commands (commands to be executed by the operating system) can be freely mixed with other code.

NOTE: the syntax and shell usage is identical on all platforms, including Win32. To avoid portability problems on Win32, it is recommended that you avoid the use of the native shell interpreter `cmd`.

```
LIB = $(dir lib)
println(The contents of the $(LIB) directory is:)
ls $(LIB)
```

10.1 Simple commands

The syntax of shell commands is similar to the syntax used by the Unix shell `bash`. In general, a command is a *pipeline*. A basic command is part of a pipeline. It is specified with the name of an executable and some arguments. Here are some examples.

```
ls
ls -AF .
echo Hello world
```

The command is found using the current search path in the variable `PATH[]`, which should define an array of directories containing executables.

A command may also be prefixed by environment variable definitions.

```
# Prints "Hello world"
env X="Hello world" Y=2 printenv X
# Pass the include path to the Visual C++
env include="c:\Program Files\Microsoft SDK\include" cl foo.cpp
```

10.2 Globbing

Commands may contain wildcard patterns. A pattern specifies a set of files through a limited kind of regular expression. Patterns are expanded before the function is executed.

```
# List all files with a .c suffix
ls *.c

# List all files with a single character prefix, and .c suffix
ls ?.c

# Rename the file hello.ml to foo.ml
mv {hello,foo}.ml
```

A comprehensive description of OMake glob patterns is given in Section 9.4.

10.3 Background jobs

The command may also be placed in the background by placing an ampersand after the command. Control returns to the shell without waiting for the job to complete. The job continues to run in the background.

```
gcc -o hugeprogram *.c &
```

10.4 File redirection

Input and output can be redirected to files by using the `<`, `>`, and `>&` directives after the command.

```
# Write to the "foo" file
echo Hello world > foo

# Redirect input from the foo file
cat < foo

# Redirect standard output and errors to the foo file
gcc -o boo *.c >& foo
```

10.5 Pipelines

Pipelines are sequences of commands, where the output from each command is sent to the next. Pipelines are defined with the `|` and `|&` syntax. With `|` the output is redirected, but errors are not. With `|&` both output and errors are redirected.

```
# Send the output of the ls command to the printer
ls *.c | lpr

# Send output and errors to jyh as email
gcc -o hugefile *.c |& mail jyh
```

10.6 Conditional execution

Commands may also be composed though conditional evaluation using the `||` and `&&` syntax. Every command has an integer exit code, which may be zero or some other integer. A command is said to *succeed* if its exit code is zero. The expression `command1 && command2` executes `command2` only if `command1` succeeds. The expression `command1 || command2` executes `command2` only if `command1` fails.

```
# Display the x/y file if possible
cd x && cat y

# Run foo.exe, or print an error message
(test -x foo.exe && foo.exe) || echo "foo.exe is not executable"
```

10.7 Grouping

Parenthesis are used for grouping in a pipeline or conditional command. In the following expression, the `test` function is used to test whether the `foo.exe` file is executable. If it is, the `foo.exe` file is executed. If the file is not executable (or if the `foo.exe` command fails), the message `"foo.exe is not executable"` is printed.

```
# Run foo.exe, or print an error message
(test -x foo.exe && foo.exe) || echo "foo.exe is not executable"
```

10.8 What is a shell command?

Syntactially, shell commands are any line that is not one of the following:

- A variable definition of the form `VAR=string`
- A function call `f(...)` or method call `o.f(...)`
- A rule definition containing a colon `string: ...`
- A special command, including the following:
  - `if ...`
  - `switch ...`

- match ...
- section ...
- return ...

Commands may also be builtin (aliases). See the documentation for the `Shell` object for more information.

10.9 Basic builtin functions

10.9.1 echo

The `echo` function prints a string.

```
$(echo <args>)
echo <args>
```

10.9.2 cd

The `cd` function changes the current directory.

```
cd(dir)
   dir : Dir
```

The `cd` function also supports a 2-argument form:

```
$(cd dir, e)
   dir : Dir
   e : expression
```

In the two-argument form, expression `e` is evaluated in the directory `dir`. The current directory is not changed otherwise.

The behavior of the `cd` function can be changed with the `CDPATH` variable, which specifies a search path for directories. This is normally useful only in the `osh` command interpreter.

```
CDPATH : Dir Sequence
```

For example, the following will change directory to the first directory `./foo`, `~/dir1/foo`, `~/dir2/foo`.

```
CDPATH[] =
   .
   $(HOME)/dir1
   $(HOME)/dir2
cd foo
```

10.10 Job control builtin functions

10.10.1 jobs

The `jobs` function prints a list of jobs.

```
jobs
```

10.10.2 bg

The `bg` function places a job in the background.

```
bg <pid...>
```

10.10.3 fg

The `fg` function brings a job to the foreground.

```
fg <pid...>
```

10.10.4 stop

The `stop` function suspends a job.

```
stop <pid...>
```

10.10.5 wait

The `wait` function waits for a job to finish. If no process identifiers are given, the shell waits for all jobs to complete.

```
wait <pid...>
```

10.10.6 kill

The `kill` function signals a job.

```
kill [signal] <pid...>
```

10.11 Command history

10.11.1 history

```
$(history-index) : Int
$(history) : String Sequence
history-file : File
history-length : Int
```

The history variables manage the command-line history in `osh`. They have no effect in `omake`.

The `history-index` variable is the current index into the command-line history. The `history` variable is the current command-line history.

The history-file variable can be redefined if you want the command-line history to be saved. The default value is ~/.omake/osh_history.

The history-length variable can be redefined to specify the maximum number of lines in the history that you want saved. The default value is 100.

Chapter 11

The standard objects

`Pervasives` defines the objects that are defined in all programs. The following objects are defined.

11.1 Pervasives objects

11.1.1 Object

Parent objects: none.

The `Object` object is the root object. Every class is a subclass of `Object`.

It provides the following fields:

- `$(o.object-length)`: the number of fields and methods in the object.
- `$(o.object-mem <var>)`: returns `true` iff the `<var>` is a field or method of the object.
- `$(o.object-add <var>, <value>)`: adds the field to the object, returning a new object.
- `$(o.object-find <var>)`: fetches the field or method from the object; it is equivalent to `$(o.<var>)`, but the variable can be non-constant.
- `$(o.object-map <fun>)`: maps a function over the object. The function should take two arguments; the first is a field name, the second is the value of that field. The result is a new object constructed from the values returned by the function.
- `o.object-foreach`: the `object-foreach` form is equivalent to `object-map`, but with altered syntax.

```
o.object-foreach(<var1>, <var2>)
   <body>
```

For example, the following function prints all the fields of an object `o`.

```
PrintObject(o) =
   o.object-foreach(v, x)
      println($(v) = $(x))
```

The `export` form is valid in a `object-foreach` body. The following function collects just the field names of an object.

```
FieldNames(o) =
   names[] =
   o.object-foreach(v, x)
      names[] += $(v)
      export
   return $(names)
```

11.1.2 Map

Parent objects: `Object`.

A `Map` object is a dictionary from values to values. The `<key>` values are restricted to simple values: integers, floating-point numbers, strings, files, directories, and arrays of simple values.

The Map object provides the following methods.

- `$(o.length)`: the number of items in the map.
- `$(o.mem <key>)`: returns `true` iff the `<key>` is defined in the map.
- `$(o.add <key>, <value>)`: adds the field to the map, returning a new map.
- `$(o.find <key>)`: fetches the field from the map.
- `$(o.keys)`: fetches an array of all the keys in the map, in alphabetical order.
- `$(o.values)`: fetches an array of all the values in the map, in the alphabetical order of the corresponding keys.
- `$(o.map <fun>)`: maps a function over the map. The function should take two arguments; the first is a field name, the second is the value of that field. The result is a new object constructed from the values returned by the function.
- `o.foreach`: the `foreach` form is equivalent to `map`, but with altered syntax.

```
o.foreach(<var1>, <var2>)
   <body>
```

For example, the following function prints all the fields of an object `o`.

```
PrintObject(o) =
   o.foreach(v, x)
      println($(v) = $(x))
```

The `export` form is valid in a `foreach` body. The following function collects just the field names of the map.

```
FieldNames(o) =
   names =
   o.foreach(v, x)
      names += $(v)
      export
   return $(names)
```

There is also simpler syntax when the key is a string. The table can be defined using definitions with the form `$|key|` (the number of pipe symbols `|` is allowed to vary).

```
$|key 1| = value1
$||key1|key2|| = value2    # The key is key1|key2
X = $|key 1|               # Define X to be the value of field $|key 1|
```

The usual modifiers are also allowed. The expression `` $`|key| `` represents lazy evaluation of the key, and `$,|key|` is normal evaluation.

11.1.3 Number

Parent objects: `Object`.

The `Number` object is the parent object for integers and floating-point numbers.

11.1.4 Int

Parent objects: `Number`.

The `Int` object represents integer values.

11.1.5 Float

Parent objects: `Number`.

The `Float` object represents floating-point numbers.

11.1.6 Sequence

Parent objects: `Object`.

The `Sequence` object represents a generic object containing sequential elements. It provides the following methods.

- `$(s.length)`: the number of elements in the sequence.
- `$(s.map <fun>)`: maps a function over the fields in the sequence. The function should take one argument. The result is a new sequence constructed from the values returned by the function.
- `s.foreach`: the `foreach` form is equivalent to `map`, but with altered syntax.

  ```
  s.foreach(<var>)
     <body>
  ```

  For example, the following function prints all the elements of the sequence.

  ```
  PrintSequence(s) =
     s.foreach(x)
        println(Elem = $(x))
  ```

  The `export` form is valid in a `foreach` body. The following function counts the number of zeros in the sequence.

  ```
  Zeros(s) =
     count = $(int 0)
     s.foreach(v)
        if $(equal $(v), 0)
           count = $(add $(count), 1)
           export
        export
     return $(count)
  ```

11.1.7 Array

Parent objects: `Sequence`.

The `Array` is a random-access sequence. It provides the following additional methods.

- `$(s.nth <i>)`: returns element `i` of the sequence.
- `$(s.rev <i>)`: returns the reversed sequence.

11.1.8 String

Parent objects: `Array`.

11.1.9 Fun

Parent objects: `Object`.

The `Fun` object provides the following methods.

- `$(f.arity)`: the arity if the function.

11.1.10 Rule

Parent objects: `Object`.

The `Rule` object represents a build rule. It does not currently have any methods.

11.1.11 Target

Parent object: `Object`.

The `Target` object contains information collected for a specific target file.

- `target`: the target file.
- `effects`: the files that may be modified by a side-effect when this target is built.
- `scanner_deps`: static dependencies that must be built before this target can be scanned.
- `static-deps`: statically-defined build dependencies of this target.
- `build-deps`: all the build dependencies for the target, including static and scanned dependencies.
- `build-values`: all the value dependencies associated with the build.
- `build-commands`: the commands to build the target.
- `output-file`: if output was diverted to a file, with one of the `--output-*` options A, this field names that file. Otherwise it is `false`.

The object supports the following methods.

- `find(file)`: returns a Target object for the given file. Raises a `RuntimeException` if the specified target is not part of the project.
- `find-optional(file)`: returns a `Target` object for the given file, or `false` if the file is not part of the project.

NOTE: the information for a target is constructed dynamically, so it is possible that the `Target` object for a node will contain different values in different contexts. The easiest way to make sure that the `Target` information is complete is to compute it within a rule body, where the rule depends on the target file, or the dependencies of the target file.

11.1.12 Node

Parent objects: `Object`.

The `Node` object is the parent object for files and directories. It supports the following operations.

- `$(node.stat)`: returns a `stat` object for the file. If the file is a symbolic link, the `stat` information is for the destination of the link, not the link itself.
- `$(node.lstat)`: returns a `stat` object for the file or symbolic link.
- `$(node.unlink)`: removes the file.
- `$(node.rename <file>)`: renames the file.
- `$(node.link <file>)`: creates a hard link `<dst>` to this file.
- `$(node.symlink <file>)`: create a symbolic link `<dst>` to this file.
- `$(node.chmod <perm>)`: change the permission of this file.
- `$(node.chown <uid>, <gid>)`: change the owner and group id of this file.

11.1.13 File

Parent objects: `Node`.

The file object represents the name of a file.

11.1.14 Dir

Parent objects: `Node`.

The `Dir` object represents the name of a directory.

11.1.15 Channel

Parent objects: `Object`.

A `Channel` is a generic IO channel. It provides the following methods.

- `$(o.close)`: close the channel.

11.1.16 InChannel

Parent objects: `Channel`.

A `InChannel` is an input channel. The variable `stdin` is the standard input channel.

It provides the following methods.

- `$(InChannel.fopen <file>)`: open a new input channel.
- `$(InChannel.of-string <string>)`: open a new input channel, using a string as input.

11.1.17 OutChannel

Parent object: `Channel`.

A `OutChannel` is an output channel. The variables `stdout` and `stderr` are the standard output and error channels.

It provides the following methods.

- `$(OutChannel.fopen <file>)`: open a new output channel.
- `$(OutChannel.string)`: open a new output channel, writing to a string.
- `$(OutChannel.to-string)`: get the current string of output, for an output channel created as `OutChannel.open-string`.
- `$(OutChannel.append <file>)`: opens a new output channel, appending to the file.
- `$(c.flush)`: flush the output channel.
- `$(c.print <string>)`: print a string to the channel.
- `$(c.println <string>)`: print a string to the channel, followed by a line terminator.

11.1.18 Location

Parent objects: `Location`.

The `Location` object represents a location in a file.

11.1.19 Position

Parent objects: `Position`.

The `Position` object represents a stack trace.

11.1.20 Exception

Parent objects: `Object`.

The `Exception` object is used as the base object for exceptions. It has no fields.

11.1.21 RuntimeException

Parent objects: `Exception`.

The `RuntimeException` object represents an exception from the runtime system. It has the following fields.

- `position`: a string representing the location where the exception was raised.
- `message`: a string containing the exception message.

11.1.22 UnbuildableException

Parent objects: `Exception`.

The `UnbuildableException` object should be used to signal that a target is not buildable. It will be caught by functions such as `target-exists`. This exception has the following fields:

- `target`: indicates which target is not buildable.
- `message`: a string containing the exception message.

11.1.23 Shell

Parent objects: `Object`.

The `Shell` object contains the collection of builtin functions available as shell commands.

You can define aliases by extending this object with additional methods. All methods in this class are called with one argument: a single array containing an argument list.

- `echo`

  The `echo` function prints its arguments to the standard output channel.

- `jobs`

  The `jobs` method prints the status of currently running commands.

- `cd`

  The `cd` function changes the current directory. Note that the current directory follows the usual scoping rules. For example, the following program lists the files in the `foo` directory, but the current directory is not changed.

  ```
  section
     echo Listing files in the foo directory...
     cd foo
     ls
  ```

```
echo Listing files in the current directory...
ls
```

- `bg`

  The `bg` method places a job in the background. The job is resumed if it has been suspended.

- `fg`

  The `fg` method brings a job to the foreground. The job is resumed if it has been suspended.

- `stop`

  The `stop` method suspends a running job.

- `wait`

  The `wait` function waits for a running job to terminate. It is not possible to wait for a suspended job.

  The job is not brought to the foreground. If the `wait` is interrupted, the job continues to run in the background.

- `kill`

  The `kill` function signal a job.

  `kill [signal] <pid...>`.

  The signals are either numeric, or symbolic. The symbolic signals are named as follows.

  ABRT, ALRM, HUP, ILL, KILL, QUIT, SEGV, TERM, USR1, USR2, CHLD, STOP, TSTP, TTIN, TTOU, VTALRM, PROF.

- `exit`

  The `exit` function terminates the current session.

- `which`, `where`

  See the documentation for the corresponding functions.

- `rehash`

  Reset the search path.

- `ln-or-cp` *src* dst

  Links or copies *src to* dst, overwriting *dst. Namely,* `ln-or-cp` *would first delete the* dst file (unless it is a directory), if it exists. Next it would try to create a symbolic link *dst poiting to* src (it will make all the necessary adjustmnents of relative paths). If symbolic link can not be created (*e.g.* the OS or the filesystem does not support symbolic links), it will try to create a hard link. If that fails too, it will try to forcibly copy *src to* dst.

- `history`

  Print the current command-line history.

- `digest`

  Print the digests of the given files.

- Win32 functions.

  Win32 doesn't provide very many programs for scripting, except for the functions that are builtin to the DOS `cmd.exe`. The following functions are defined on Win32 and only on Win32. On other systems, it is expected that these programs already exist.

  - `grep`

    ```
    grep [-q] [-n] pattern files...
    ```

    The `grep` function calls the `omake grep` function.

- Internal versions of standard system commands.

  By default, `omake` uses internal versions of the following commands: `cp`, `mv`, `cat`, `rm`, `mkdir`, `chmod`, `test`, `find`. If you really want to use the standard system versions of these commands, set the `USE_SYSTEM_COMMANDS` as one of the first definitions in your `OMakeroot` file.

  - `mkdir`

    ```
    mkdir [-m <mode>] [-p] files
    ```

    The `mkdir` function is used to create directories. The -verb+-m+ option can be used to specify the permission mode of the created directory. If the `-p` option is specified, the full path is created.

  - `cp`
  - `mv`

    ```
    cp [-f] [-i] [-v] src dst
    cp [-f] [-i] [-v] files dst
    mv [-f] [-i] [-v] src dst
    mv [-f] [-i] [-v] files dst
    ```

    The `cp` function copies a `src` file to a `dst` file, overwriting it if it already exists. If more than one source file is specified, the final file must be a directory, and the source files are copied into the directory.

    -f Copy files forcibly, do not prompt.

    -i Prompt before removing destination files.

    -v Explain what is happening.

  - `rm`

```
rm [-f] [-i] [-v] [-r] files
rmdir [-f] [-i] [-v] [-r] dirs
```

The `rm` function removes a set of files. No warnings are issued if the files do not exist, or if they cannot be removed.

Options:

-f Forcibly remove files, do not prompt.

-i Prompt before removal.

-v Explain what is happening.

-r Remove contents of directories recursively.

– `chmod`

```
chmod [-r] [-v] [-f] mode files
```

The `chmod` function changes the permissions on a set of files or directories. This function does nothing on Win32. The `mode` may be specified as an octal number, or in symbolic form `[ugoa]*[-`=][rwxXstugo]+. See the man page for `chmod` for details.

Options:

-r Change permissions of all files in a directory recursively.

-v Explain what is happening.

-f Continue on errors.

– `cat`

```
cat files...
```

The `cat` function prints the contents of the files to stdout

– `test`

`test` *expression*
`[` *expression* +]+
`[ --help`
`[ --version`

See the documentation for the `test` function.

– `find`

```
find \emph{expression}
```

See the documentation for the `find` function.

Chapter 12

Build functions and utilities

12.1 Builtin .PHONY targets

The complete set of builtin `.PHONY` targets include the following.

.PHONY Declares new phony targets (Section 7.10).

.DEFAULT Declare the default build targets (Section 7.7).

.SUBDIRS Include a directory as part of the project (Section 7.8).

.SCANNER Define a dependency scanner (Section 7.8).

.INCLUDE Include a file (Section 7.9).

.ORDER Define a file-dependency ordering rule (Section 9.3.5).

.BUILD_BEGIN Commands to be executed at the beginning of a build.

.BUILD_SUCCESS Commands to be executed if the build is successful.

.BUILD_FAILURE Commands to be executed if the build fails.

The `.BUILD` targets can be used to specify commands to be executed at the beginning and end of the build. The `.BUILD_BEGIN` target is built at the beginning of a project build, and one of `.BUILD_FAILURE` or `.BUILD_SUCCESS` is executed when the build terminates.

For example, the following set of rules simply print additional messages about the status of the build.

```
.BUILD_BEGIN:
    echo Build starting

.BUILD_SUCCESS:
    echo The build was successful
```

```
.BUILD_FAILURE:
    println($"The build failed: $(length $(find-build-targets Failed)) targets could not be built")
```

Another common use is to define notifications to be performed when the build completes. For example, the following rule will create a new X terminal displaying the summary of the build (using the `BUILD_SUMMARY` variable).

```
.BUILD_FAILURE:
    xterm -e vi $(BUILD_SUMMARY)
```

If you do not wish to add these rules directly to your project (which is probably a good idea if you work with others), you can define them in your `.omakerc` (see Section A.8).

The `find-build-targets` function is useful for obtaining a firther summary of the build. Note that when output diversions are in effect (with the `--output-*` options — see Chapter A), any output produced by the commands is copied to a file. The name of the file is specified by the `output-file` field of the `Target` object. You may find this useful in defining custom build summaries.

12.2 Options and versioning

12.2.1 OMakeFlags

```
OMakeFlags(options)
   options : String
```

The `OMakeFlags` function is used to set `omake` options from within `OMakefile`s. The options have exactly the same format as options on the command line.

For example, the following code displays the progress bar unless the `VERBOSE` environment variable is defined.

```
if $(not $(defined-env VERBOSE))
    OMakeFlags(-S --progress)
    export
```

12.2.2 OMakeVersion

```
OMakeVersion(version1)
OMakeVersion(version1, version2)
   version1, version2 : String
```

The `OMakeVersion` function is used for version checking in `OMakefile`s. It takes one or two arguments.

In the one argument form, if the `omake` version number is less than `<version1>`, then an exception is raised. In the two argument form, the version must lie between `version1` and `version2`.

12.2.3 cmp-versions

```
$(cmp-versions version1, version2)
   version1, version2 : String
```

The `cmp-versions\` functions can be used to compare arbitrary version strings. It returns 0 when the two version strings are equal, a negative number when the first string represents an earlier version, and a positive number otherwise.

12.2.4 DefineCommandVars

```
DefineCommandVars()
```

The `DefineCommandVars` function redefines the variables passed on the commandline. Variables definitions are passed on the command line in the form `name=value`. This function is primarily for internal use by `omake` to define these variables for the first time.

12.3 Examining the dependency graph

12.3.1 dependencies, dependencies-all, dependencies-proper

```
$(dependencies targets) : File Array
$(dependencies-all targets) : File Array
$(dependencies-proper targets) : File Array
   targets : File Array
raises RuntimeException
```

The `dependencies` function returns the set of immediate dependencies of the given targets. This function can only be used within a rule body and all the arguments to the `dependency` function must also be dependencies of this rule. This restriction ensures that all the dependencies are known when this function is executed.

The `dependencies-all` function is similar, but it expands the dependencies recursively, returning all of the dependencies of a target, not just the immediate ones.

The `dependencies-proper` function returns all recursive dependencies, except the dependencies that are leaf targets. A leaf target is a target that has no dependencies and no build commands; a leaf target corresponds to a source file in the current project.

In all three functions, files that are not part of the current project are silently discarded.

One purpose of the `dependencies-proper` function is for "clean" targets. For example, one way to delete all intermediate files in a build is with a rule that uses the `dependencies-proper`. Note however, that the rule requires building the project before it can be deleted. For a shorter form, see the `filter-proper-targets` function.

```
.PHONY: clean

APP = ...     # the name of the target application
clean: $(APP)
   rm $(dependencies-proper $(APP))
```

12.3.2 target

```
$(target targets) : Target Array
   targets : File Sequence
raises RuntimeException
```

The `target` function returns the Target object associated with each of the targets. See the `Target` object for more information.

12.3.3 find-build-targets

```
$(find-build-targets tag) : Target Array
   tag : Succeeded | Failed
```

The `find-build-targets` allow the results of the build to be examined. The `tag` must specifies which targets are to be returned; the comparison is case-insensitive.

Succeeded The list of targets that were built successfully.

Failed The list of targets that could not be built.

These are used mainly in conjuction with the `.BUILD_SUCCESS` (Section 12.1) and `.BUILD_FAILURE` (Section 12.1) phony targets. For example, adding the following to your project `OMakefile` will print the number of targets that failed (if the build failed).

```
.BUILD_FAILURE:
    echo "Failed target count: $(length $(find-build-targets Failed))"
```

12.3.4 project-directories

```
$(project-directories) : Dir Array
```

The `project-directories` function returns the list of all directories that are considered to be part of the project.

To get the complete directory list, this function should be called from within a rule body.

12.3.5 rule

The `rule` function is called whenever a build rule is defined. It is unlikely that you will need to redefine this function, except in very exceptional cases.

```
rule(multiple, target, pattern, sources, options, body) : Rule
   multiple : String
   target   : Sequence
   pattern  : Sequence
   sources  : Sequence
   options  : Array
   body     : Body
```

The `rule` function is called when a rule is evaluated.

multiple A Boolean value indicating whether the rule was defined with a double colon `::`.

target The sequence of target names.

pattern The sequence of patterns. This sequence will be empty for two-part rules.

sources The sequence of dependencies.

options An array of options. Each option is represented as a `Map` object associating each specified option with a value.

body The body expression of the rule.

Consider the following rule.

```
target: pattern: sources :name1: option1 :name2: option2
   expr1
   expr2
```

This expression represents the following function call, where square brackets are used to indicate arrays, and the curly brackets represent a `Map` object.

```
rule(false, target, pattern, sources,
     { $|:name1:| = option1; $|:name2:| = option2 }
     [expr1; expr2])
```

12.4 The OMakeroot file

The standard `OMakeroot` file defines the functions are rules for building standard projects.

12.4.1 Variables

ROOT The root directory of the current project.

CWD The current working directory (the directory is set for each `OMakefile` in the project).

EMPTY The empty string.

STDROOT The name of the standard installed `OMakeroot` file.

ABORT_ON_COMMAND_ERROR If set to true, the construction of a target should be aborted whenever one of the commands to build it fail. This defaults to true, and should normally be left that way.

SCANNER_MODE This variable should be defined as one of four values (defaults to `enabled`).

enabled Allow the use of default `.SCANNER` rules. Whenever a rule does not specify a `:scanner:` dependency explicitly, try to find a `.SCANNER` with the same target name.

disabled Never use default `.SCANNER` rules.

warning Allow the use of default `.SCANNER` rules, but print a warning whenever one is selected.

error Do not allow the use of default `.SCANNER` rules. If a rule does not specify a `:scanner:` dependency, and there is a default `.SCANNER` rule, the build will terminate abnormally.

12.4.2 System variables

INSTALL The command to install a program (`install` on `Unix`, `cp` on `Win32`).

PATHSEP The normal path separator (`:` on `Unix`, `;` on `Win32`).

DIRSEP The normal directory separator (`/` on `Unix`, `\` on `Win32`).

EXT_LIB File suffix for a static library (default is `.a` on `Unix`, and `.lib` on `Win32`).

EXT_OBJ File suffix for an object file (default is `.o` on `Unix`, and `.obj` on `Win32`).

EXT_ASM File suffix for an assembly file (default is `.s` on `Unix`, and `.asm` on `Win32`).

EXE File suffix for executables (default is empty for `Unix`, and `.exe` on `Win32` and `Cygwin`).

12.5 Building C and C++ code

OMake provides extensive support for building C and C++ programs. In order to use the functions defined in this section, you need to make sure the line

```
open build/C
```

is present in your `OMakeroot` file.

12.5.1 Autoconfiguration variables

These variables will get defined based on the "autoconf-style" `static.` tests executed when you run OMake for the first time. You can use them to configure your project accordingly, and you should not redefine them.

You can use the `--configure` command line option (Section A.3.9) to force re-execution of all the tests.

A different set of autoconfiguration tests is performed depending on the build environment involved — one set of tests would be performed in a `Win32` environment, and another — in a Unix-like environment (including Linux, OS X and Cygwin).

12.5.1.1 Unix-like systems

GCC_FOUND A boolean flag specifying whether the `gcc` binary was found in your path.

GXX_FOUND A boolean flag specifying whether the `g++` binary was found in your path.

12.5.1.2 Win32

CL_FOUND A boolean flag specifying whether the `cl` binary was found in your path.

LIB_FOUND A boolean flag specifying whether the `lib` binary was found in your path.

12.5.2 C and C++ configuration variables

The following variables can be redefined in your project.

CC The name of the C compiler (on `Unix` it defaults to `gcc` when `gcc` is present and to `cc` otherwise; on `Win32` defaults to `cl /nologo`).

CXX The name of the C++ compiler (on `Unix` it defaults to `gcc` when `gcc` is present and to `c++` otherwise; on `Win32` defaults to `cl /nologo`).

CPP The name of the C preprocessor (defaults to `cpp` on `Unix`, and `cl /E` on `Win32`).

CFLAGS Compilation flags to pass to the C compiler (default empty on `Unix`, and `/DWIN32` on `Win32`).

CXXFLAGS Compilation flags to pass to the C++ compiler (default empty on `Unix`, and `/DWIN32` on `Win32`).

INCLUDES Additional directories that specify the search path to the C and C++ compilers (default is `.`). The directories are passed to the C and C++ compilers with the `-I` option. The include path with `-I` prefixes is defined in the `PREFIXED_INCLUDES` variable.

LIBS Additional libraries needed when building a program (default is empty).

CCOUT The option to use for specifying the output file in C and C++ compilers (defaults to `-o` on `Unix` and `/Fo` on `Win32`).

AS The name of the assembler (defaults to `as` on `Unix`, and `ml` on `Win32`).

ASFLAGS Flags to pass to the assembler (default is empty on `Unix`, and `/c /coff` on `Win32`).

ASOUT The option string that specifies the output file for `AS` (defaults to `-o` on `Unix` and `/Fo` on `Win32`).

AR The name of the program to create static libraries (defaults to `ar cq` on `Unix`, and `lib` on `Win32`).

LD The name of the linker (defaults to `ld` on `Unix`, and `cl` on `Win32`).

LDFLAGS Options to pass to the linker (default is empty).

LDOUT The option to use for specifying the output file in C and C++ linkers (defaults to `-o` on `Unix` and `/Fe` on `Win32`).

YACC The name of the `yacc` parser generator (default is `yacc` on `Unix`, empty on `Win32`).

LEX The name of the `lex` lexer generator (default is `lex` on `Unix`, empty on `Win32`).

12.5.3 Generated C files

Because the C scanners do not normally know anything about generated source files (such as generated header files), these files may need to be created before running the scanner.

12.5.3.1 CGeneratedFiles, LocalCGeneratedFiles

```
CGeneratedFiles(files)
LocalCGeneratedFiles(files)
```

The `CGeneratedFiles` and `LocalCGeneratedFiles` functions specify files that need to be generated before any C files are scanned for dependencies. For example, if `config.h` and `inputs.h` are both generated files, specify:

```
CGeneratedFiles(config.h inputs.h)
```

The `CGeneratedFiles` function is *global* — its arguments will be generated before any C files anywhere in the project are scanned for dependencies. The `LocalCGeneratedFiles` function follows the normal scoping rules of OMake.

12.5.4 Building C programs and Libraries

12.5.4.1 StaticCLibrary

The `StaticCLibrary` builds a static library.

`StaticCLibrary(<target>, <files>)`

The `<target>` does *not* include the library suffix, and The `<files>` list does not include the object suffix. These are obtained from the `EXT_LIB` and `EXT_OBJ` variables.

This function returns the library filename.

The following command builds the library `libfoo.a` from the files `a.o b.o c.o` on `Unix`, or the library `libfoo.lib` from the files `a.obj b.obj c.obj` on `Win32`.

```
StaticCLibrary(libfoo, a b c)
.DEFAULT: $(StaticCLibrary libbar, a b c d)
```

12.5.4.2 StaticCLibraryCopy

The `StaticCLibraryCopy` function copies the static library to an install location.

`StaticCLibraryCopy(<tag>, <dir>, <lib>)`

The `<tag>` is the name of a target (typically a `.PHONY` target); the `<dir>` is the installation directory, and `<lib>` is the library to be copied (without the library suffix).

This function returns the filename of the library in the target directory.

For example, the following code copies the library `libfoo.a` to the `/usr/lib` directory.

```
.PHONY: install

StaticCLibraryCopy(install, /usr/lib, libfoo)
```

12.5.4.3 StaticCLibraryInstall

The `StaticCLibraryInstall` function builds a library, and sets the install location in one step. It returns the filename of the library in the target directory.

`StaticCLibraryInstall(<tag>, <dir>, <libname>, <files>)`

```
StaticCLibraryInstall(install, /usr/lib, libfoo, a b c)
```

12.5.4.4 StaticCObject, StaticCObjectCopy, StaticCObjectInstall

These functions mirror the `StaticCLibrary`, `StaticCLibraryCopy`, and `StaticCLibraryInstall` functions, but they build an *object* file (a `.o` file on `Unix`, and a `.obj` file on `Win32`).

12.5.4.5 CProgram

The `CProgram` function builds a C program from a set of object files and libraries.

`CProgram(<name>, <files>)`

The `<name>` argument specifies the name of the program to be built; the `<files>` argument specifies the files to be linked. The function returns the filename of the executable.

Additional options can be passed through the following variables.

CFLAGS Flags used by the C compiler during the link step.

LDFLAGS Flags to pass to the loader.

LIBS Additional libraries to be linked.

For example, the following code specifies that the program `foo` is to be produced by linking the files `bar.o` and `baz.o` and libraries `libfoo.a`.

```
section
   LIBS = libfoo
   LDFLAGS += -lbar
   CProgram(foo, bar baz)
```

12.5.4.6 CProgramCopy

The `CProgramCopy` function copies a file to an install location.

```
   CProgramCopy(<tag>, <dir>, <program>)

CProgramCopy(install, /usr/bin, foo)
```

12.5.4.7 CProgramInstall

The `CProgramInstall` function specifies a program to build, and a location to install, simultaneously.

```
   CProgramInstall(<tag>, <dir>, <name>, <files>)

section
   LIBS = libfoo
   LDFLAGS += -lbar
   CProgramInstall(install, /usr/bin, foo, bar baz)
```

12.5.4.8 CXXProgram, CXXProgramInstall

The `CXXProgram` and `CXXProgramInstall` functions are equivalent to their C counterparts, except that would use `$(CXX)` and `$(CXXFLAGS)` for linking instead of `$(CC)` and `$(CFLAGS)`.

12.5.4.9 StaticCXXLibrary, StaticCXXLibraryCopy, StaticCXXLibraryInstall

Similarly, `StaticCXXLibrary`, `StaticCXXLibraryCopy` and `StaticCXXLibraryInstall` are the C++ equivalents of `StaticCLibrary`, `StaticCLibraryCopy` and `StaticCLibraryInstall` functions.

12.6 Building OCaml code

OMake provides extensive support for building OCaml code, including support for tools like `ocamlfind`, `ocamlyacc` and `menhir`. In order to use the functions defined in this section, you need to make sure the line

```
open build/OCaml
```

is present in your `OMakeroot` file.

12.6.1 Autoconfiguration variables for OCaml compilation

These variables will get defined based on the "autoconf-style" tests executed when you run OMake for the first time. You can use them to configure your project accordingly, and you should not redefine them.

You can use the `--configure` command line option (Section A.3.9) to force re-execution of all the tests.

OCAMLOPT_EXISTS True when `ocamlopt` (or `ocamlopt.opt`) is available on your machine.

OCAMLFIND_EXISTS True when the ocamlfind is available on your machines.

OCAMLDEP_MODULES_AVAILABLE True when a version of `ocamldep` that understands the `-modules` option is available on your machine.

MENHIR_AVAILABLE True when the Menhir parser-generator is available on your machine.

12.6.2 Configuration variables for OCaml compilation

The following variables can be redefined in your project.

USE_OCAMLFIND Whether to use the `ocamlfind` utility (default `false`)

OCAMLC The OCaml bytecode compiler (default `ocamlc.opt` if it exists and `USE_OCAMLFIND` is not set, otherwise `ocamlc`).

OCAMLOPT The OCaml native-code compiler (default `ocamlopt.opt` if it exists and `USE_OCAMLFIND` is not set, otherwise `ocamlopt`).

CAMLP4 The `camlp4` preprocessor (default `camlp4`).

OCAMLLEX The OCaml lexer generator (default `ocamllex`).

OCAMLLEXFLAGS The flags to pass to `ocamllex` (default `-q`).

OCAMLYACC The OCaml parser generator (default `ocamlyacc`).

OCAMLYACCFLAGS Additional options to pass to `$(OCAMLYACC)`.

OCAMLDEP The OCaml dependency analyzer (default `ocamldep`).

OCAMLDEP_MODULES The OCaml dependency analyzer that understands the `-module` option (default `ocamldep`, if `ocamldep -modules` works, or `ocamlrun ocamldep-omake`, if `ocamlrun ocamldep-omake -modules` works, and empty when neither works).

OCAMLDEP_MODULES_ENABLED Instead of using `OCAMLDEP` in a traditional `make`-style fashion, run `$(OCAMLDEP_MODULES) -modules` and then postprocess the output internally to discover all the relevant generated `.ml` and `.mli` files. See Section 12.6.5 for more information on interactions between OMake, `OCAMLDEP` and generated files. This feature is currently considered highly experimental and is disabled by default.

OCAMLMKTOP The OCaml toploop compiler (default `ocamlmktop`).

OCAMLLINK The OCaml bytecode linker (default `$(OCAMLC)`).

OCAMLOPTLINK The OCaml native-code linker (default `$(OCAMLOPT)`).

OCAMLINCLUDES Search path to pass to the OCaml compilers (default `.`). The search path with the `-I` prefix is defined by the `PREFIXED_OCAMLINCLUDES` variable.

OCAMLFIND The `ocamlfind` utility (default `ocamlfind` if `USE_OCAMLFIND` is set, otherwise empty).

OCAMLFINDFLAGS The flags to pass to `ocamlfind` (default empty, `USE_OCAMLFIND` must be set).

OCAMLPACKS Package names to pass to `ocamlfind` (`USE_OCAMLFIND` must be set).

BYTE_ENABLED Flag indicating whether to use the bytecode compiler (default `true`, when no `ocamlopt` found, `false` otherwise).

NATIVE_ENABLED Flag indicating whether to use the native-code compiler (default `true`, when ocamlopt is found, `false` otherwise). Both `BYTE_ENABLED` and `NATIVE_ENABLED` can be set to true; at least one should be set to true.

MENHIR_ENABLED Define this as `true` if you wish to use `menhir` instead of `ocamlyacc` (default `false`).

12.6.3 OCaml command flags

The following variables specify *additional* options to be passed to the OCaml tools.

OCAMLDEPFLAGS Flags to pass to `OCAMLDEP` (but not to `OCAMLDEP_MODULES`).

OCAMLPPFLAGS Flags to pass to `CAMLP4`.

OCAMLCFLAGS Flags to pass to the byte-code compiler (default `-g`).

OCAMLOPTFLAGS Flags to pass to the native-code compiler (default empty).

OCAMLFLAGS Flags to pass to either compiler (default `-warn-error A`).

OCAML_BYTE_LINK_FLAGS Flags to pass to the byte-code linker (default empty).

OCAML_NATIVE_LINK_FLAGS Flags to pass to the native-code linker (default empty).

OCAML_LINK_FLAGS Flags to pass to either linker.

MENHIR_FLAGS Additional flags to pass to `menhir`.

12.6.4 Library variables

The following variables are used during linking.

OCAML_LIBS Libraries to pass to the linker. These libraries become dependencies of the link step.

OCAML_OTHER_LIBS Additional libraries to pass to the linker. These libraries are *not* included as dependencies to the link step. Typical use is for the OCaml standard libraries like `unix` or `str`.

OCAML_CLIBS C libraries to pass to the linker.

OCAML_LIB_FLAGS Extra flags for the library linker.

ABORT_ON_DEPENDENCY_ERRORS OCaml linker requires the OCaml files to be listed in dependency order. Normally, all the functions presented in this section will automatically sort the list of OCaml modules passed in as the `<files>` argument. However, this variable is set to `true`, the order of the files passed into these function will be left as is, but OMake will abort with an error message if the order is illegal.

12.6.5 Generated OCaml Files

As of OCaml version 3.09.2, the standard `ocamldep` scanner is "broken". The main issue is that it finds only those dependencies that already exist. If `foo.ml` contains a dependency on `Bar`,

```
foo.ml:
   open Bar
```

then the default `ocamldep` will only find the dependency if a file `bar.ml` or `bar.ml` exists in the include path. It will not find (or print) the dependency if, for example, only `bar.mly` exists at the time `ocamldep` is run, even though `bar.ml` and `bar.mli` can be generated from `bar.mly`.

OMake currently provides two methods for addressing this problem — one that requires manually specifying the generated files, and an experimental method for discovering such "hidden" dependencies automatically. The `OCAMLDEP_MODULES_ENABLED` variable controls which method is going to be used. When this variable is false, the manual specifications are expected and when it is true, the automated discovery will be attempted.

12.6.5.1 OCamlGeneratedFiles, LocalOCamlGeneratedFiles

```
OCamlGeneratedFiles(files)
LocalOCamlGeneratedFiles(files)
```

When the `OCAMLDEP_MODULES_ENABLED` variable variable is set to `false`, the `OCamlGeneratedFiles` and `LocalOCamlGeneratedFiles` functions specify files that need to be generated before any OCaml files are scanned for dependencies. For example, if `parser.ml` and `lexer.ml` are both generated files, specify:

```
OCamlGeneratedFiles(parser.ml lexer.ml)
```

The `OCamlGeneratedFiles` function is *global* — its arguments will be generated before any OCaml files anywhere in the project are scanned for dependencies. The `LocalOCamlGeneratedFiles` function follows the normal scoping rules of OMake.

These functions have no effect when the `OCAMLDEP_MODULES_ENABLED` variable is true.

12.6.5.2 Automatic discovery of generated files during dependency analysis

Having to specify the generated files manualy when OMake could discover them automatically is obviously suboptimal. To address this, we try to use a custom `ocamldep` that *only* finds the free module names in a file.

This functionality is experimental and is disabled by default for now. Set the `OCAMLDEP_MODULES_ENABLED` variable to `true` (or to `$(OCAMLDEP_MODULES_AVAILABLE)`) in your project to enable it.

Note that the experimental `ocamldep` functionality this relies upon is not yet included in the standard OCaml (it is expected to be a part of the upcoming OCaml 3.10 — see `http://caml.inria.fr/mantis/view.php?id=4047`). Temporarily, we distribute a bytecode version `ocamldep-omake` of the appropriately modified `ocamldep`. The appropriate `ocamldep` will be discovered automatically — see and the `OCAMLDEP_MODULES_AVAILABLE` and `OCAMLDEP_MODULES` variables will be set accordingly.

12.6.6 Using the Menhir parser generator

Menhir is a parser generator that is mostly compatible with `ocamlyacc`, but with many improvements. A few of these are listed here (excerpted from the Menhir home page `http://cristal.inria.fr/~fpottier/menhir/`).

- Menhir's explanations are believed to be understandable by mere humans.
- Menhir allows grammar specifications to be split over multiple files. It also allows several grammars to share a single set of tokens.
- Menhir is able to produce parsers that are parameterized by Objective Caml modules.

Added by jyh With the `--infer` option, Menhir can typecheck the semantic actions in your grammar at *generation* time.

What do you need to do to use Menhir instead of `ocamlyacc`?

1. Place the following definition before the relevant section of your project (or at the top of your project `OMakefile` if you want to use Menhir everywhere).

   ```
   MENHIR_ENABLED = true
   ```

2. Optionally, add any desired Menhir options to the `MENHIR_FLAGS` variable.

   ```
   MENHIR_FLAGS += --infer
   ```

With this setup, any file with a `.mly` suffix will be compiled with Menhir.

If your grammar is split across several files, you need to specify it explicitly, using the `MenhirMulti` function.

```
MenhirMulti(target, sources)
    target : filename, without suffix
    sources : the files that define the grammar, without suffixes
```

For example, if you want to generate the parser files `parse.ml` and `parse.mli`, from the grammar specified in files `a.mly` and `b.mly`, you would use the following.

```
MenhirMulti(parse, a b)
```

12.6.6.1 OCamlLibrary

The OCamlLibrary function builds an OCaml library.

```
OCamlLibrary(<libname>, <files>)
```

The <libname> and <files> are listed *without* suffixes.

This function returns the list of all the targets that it defines the rules for (including the $(name)$(EXT_LIB) file when NATIVE_ENABLED is set).

The following code builds the libfoo.cmxa library from the files foo.cmx and bar.cmx (if NATIVE_ENABLED is set), and libfoo.cma from foo.cmo and bar.cmo (if BYTE_ENABLED is set).

```
OCamlLibrary(libfoo, foo bar)
```

12.6.6.2 OCamlPackage

The OCamlPackage function builds an OCaml package.

```
OCamlPackage(<name>, <files>)
```

The <name> and <files> are listed *without* suffixes. The <files> must have been compiled with the -for-pack <ident> flag to the OCaml compiler.

This function returns the list of all the targets that it defines the rules for (including the $(name)$(EXT_LIB) file when NATIVE_ENABLED is set).

The following code builds the libfoo.cmx package from the files package.cmx and bar.cmx (if NATIVE_ENABLED is set), and package.cmo from foo.cmo and bar.cmo (if BYTE_ENABLED is set).

```
OCamlPackage(package, foo bar)
```

12.6.6.3 OCamlLibraryCopy

The OCamlLibraryCopy function copies a library to an install location.

```
OCamlLibraryCopy(<tag>, <libdir>, <libname>, <interface-files>)
```

The <interface-files> specify additional interface files to be copied if the INSTALL_INTERFACES variable is true.

12.6.6.4 OCamlLibraryInstall

The OCamlLibraryInstall function builds a library and copies it to an install location in one step.

```
OCamlLibraryInstall(<tag>, <libdir>, <libname>, <files>)
```

12.6.6.5 OCamlProgram

The OCamlProgram function builds an OCaml program. It returns the array with all the targets for which it has defined the rules ($(name)$(EXE) and $(name).run and/or $(name).opt, depending on the NATIVE_ENABLED and BYTE_ENABLED variables).

```
OCamlProgram(<name>, <files>)
```

Additional variables used:

`OCAML_LIBS` Additional libraries passed to the linker, without suffix. These files become dependencies of the target program.

`OCAML_OTHER_LIBS` Additional libraries passed to the linker, without suffix. These files do *not* become dependencies of the target program.

`OCAML_CLIBS` C libraries to pass to the linker.

`OCAML_BYTE_LINK_FLAGS` Flags to pass to the bytecode linker.

`OCAML_NATIVE_LINK_FLAGS` Flags to pass to the native code linker.

`OCAML_LINK_FLAGS` Flags to pass to both linkers.

12.6.6.6 OCamlProgramCopy

The `OCamlProgramCopy` function copies an OCaml program to an install location.

```
OCamlProgramCopy(<tag>, <bindir>, <name>)
```

Additional variables used:

NATIVE_ENABLED If the `NATIVE_ENABLED` variable is set, the native-code executable is copied; otherwise the byte-code executable is copied.

12.6.6.7 OCamlProgramInstall

The `OCamlProgramInstall` function builds a programs and copies it to an install location in one step.

```
OCamlProgramInstall(<tag>, <bindir>, <name>, <files>)
```

12.7 Building LaTeX files

OMake provides support for building LaTeX documents, including support for automatically running BiBTex and for producing PostScript and PDF files. In order to use the functions defined in this section, you need to make sure the line

```
open build/LaTeX
```

is present in your `OMakeroot` file.

12.7.1 Configuration variables

The following variables can be modified in your project.

LATEX The LaTeX command (default `latex`).

TETEX2_ENABLED Flag indicating whether to use advanced LaTeX options present in TeTeX v.2 (default value is determined the first time omake reads `LaTeX.src` and depends on the version of LaTeX you have installed).

LATEXFLAGS The LaTeX flags (defaults depend on the `TETEX2_ENABLED` variable)

BIBTEX The BibTeX command (default `bibtex`).

MAKEINDEX The command to build an index (default `makeindex`).

DVIPS The `.dvi` to PostScript converter (default `dvips`).

DVIPSFLAGS Flags to pass to `dvips` (default `-t letter`).

DVIPDFM The `.dvi` to `.pdf` converter (default `dvipdfm`).

DVIPDFMFLAGS Flags to pass to `dvipdfm` (default `-p letter`).

PDFLATEX The `.latex` to `.pdf` converter (default `pdflatex`).

PDFLATEXFLAGS Flags to pass to pdflatex (default is `` $`(LATEXFLAGS) ``).

USEPDFLATEX Flag indicating whether to use pdflatex instead of dvipdfm to generate the `.pdf` document (default `false`).

12.7.2 Building LaTeX documents

12.7.2.1 LaTeXDocument

The `LaTeXDocument` produces a LaTeX document.

```
LaTeXDocument(<name>, <texfiles>)
```

The document `<name>` and `<texfiles>` are listed without suffixes. This function returns the filenames for the generated `.ps` and `.pdf` files.

Additional variables used:

TEXINPUTS The LaTeX search path (an array of directories, default is taken from the `TEXINPUTS` environment variable).

TEXDEPS Additional files this document depends on.

TEXVARS An array of names of the environment variables that are to be updated based on the value of OMake's `TEXINPUTS` variable. Defaults to `TEXINPUTS BIBINPUTS BSTINPUTS`.

12.7.2.2 TeXGeneratedFiles, LocalTeXGeneratedFiles

```
TeXGeneratedFiles(files)
LocalTeXGeneratedFiles(files)
```

The `TeXGeneratedFiles` and `LocalTeXGeneratedFiles` functions specify files that need to be generated before any LaTeXfiles are scanned for dependencies. For example, if `config.tex` and `inputs.tex` are both generated files, specify:

```
TeXGeneratedFiles(config.tex inputs.tex)
```

The `TeXGeneratedFiles` function is *global* — its arguments will be generated before any TeX files anywhere in the project are scanned for dependencies. The `LocalTeXGeneratedFiles` function follows the normal scoping rules of OMake.

12.7.2.3 LaTeXDocumentCopy

The `LaTeXDocumentCopy` copies the document to an install location.

```
LaTeXDocumentCopy(<tag>, <libdir>, <installname>, <docname>)
```

This function copies just the `.pdf` and `.ps` files.

12.7.2.4 LaTeXDocumentInstall

The `LaTeXDocumentInstall` builds a document and copies it to an install location in one step.

```
LaTeXDocumentInstall(<tag>, <libdir>, <installname>, <docname>, <files>)
```

Chapter 13

Autoconfiguration functions and variables

OMake standard library provides a number of functions and variables intended to help one write build specifications that need to be capable of autoconfiguring itself to adjust to different build environments.

13.1 General-purpose autoconfiguration functions

The following general-purpose functions can be used to discover the properties of your build environment in a fashion similar to the one used by GNU autoconf tool you may be familiar with. It is recommended that these function be used from an appropriate `static.` block (see Section 4.18 for more information).

In order to use the following general-purpose functions, you need to have the line

```
open configure/Configure
```

included in your `OMakefile` or `OMakeroot`.

13.1.1 ConfMsgChecking, ConfMsgResult

```
ConfMsgChecking(<msg>)
...
ConfMsgResult(<msg>)
```

The `ConfMsgChecking` function output message of the form `--- Checking <msg>...` *without* any trailing newline. After the test advertized by `ConfMsgChecking` is performed, the `ConfMsgResult` function should be used to output the result.

In certain cases users may want to redefine these function — for example, to use a different output formatting and/or to copy the messages to a log file.

Example:

```
static. =
   ConfMsgChecking(which foo to use)
   foo = ...
   ConfMsgResult($(foo))
```

13.1.2 ConfMsgWarn, ConfMsgError

```
ConfMsgWarn(<msg>)
ConfMsgError(<msg>)
```

Print a warning or an error message respectively. `ConfMsgError` would then abort OMake.

13.1.3 ConfMsgYesNo, ConfMsgFound

```
flag = $(ConfMsgYesNo <bool expr>
flag = $(ConfMsgFound <bool expr>
```

The `ConfMsgFound` function expects to receive a boolean flag describing whether a test previously announced using the `ConfMsgChecking` function found what it was looking for. `ConfMsgFound` will output the appropriate result ("found" or "NOT found") using the `ConfMsgResult` function and return its argument back.

The `ConfMsgYesNo` function is similar, outputting a simple ("yes" or "NO").

13.1.4 TryCompileC, TryLinkC, TryRunC

```
success = $(TryCompileC <prog_text>)
success = $(TryLinkC <prog_text>)
success = $(TryRunC <prog_text>)
```

Given the *text* of a C program, the `TryCompileC`, `TryLinkC`, and `TryRunC` functions would try to compile / compile and link / compile, link, and run, the given program and return a boolean flag indicating whether the attempt was successful.

`TryCompileC` will use the `CC`, `CFLAGS` and `INCLUDES` variables to run the C compiler. `TryLinkC` and `TryRunC` will also use the `LDFLAGS` variable to run the C compiler and linker. However, the flags like `/WX`, `-Werror` and `-warn-error` will be not be passed to the compiler, even if they occur in `CFLAGS`.

These functions are silent and should normally be used with an appropriate `ConfMsgChecking` ... `ConfMsgResult`.

13.1.5 RunCProg

```
output = $(RunCProg <prog>)
```

`RunCProg` is similar to the `RunCProg` function, except that it returns the output of the function (will return `false` if the program fails to compile or run).

13.1.6 CheckCHeader, VerboseCheckCHeader

```
success = $(CheckCHeader <files>)
success = $(VerboseCheckCHeader <files>)
```

Use the `TryCompileC` function to check whether your C compiler can locate and process the specified headers files. Will incude `<stdio.h>` before including the header files.

Both functions return a boolean value. The `CheckCHeader` function is silent; the `VerboseCheckCHeader` function will use the `ConfMsgChecking` and `ConfMsgResult` functions to describe the test and the outcome.

Example:

```
static. =
   NCURSES_H_AVAILABLE = $(VerboseCheckCHeader ncurses.h)
```

13.1.7 CheckCLib, VerboseCheckCLib

```
success = $(CheckCLib <libs>, <functions>)
success = $(VerboseCheckCLib <libs>, <functions>)
```

Use the `TryLinkC` function to check whether your C compiler and linker can find the named functions when linking with the named libraries. Will pass the `<libs>` to the compiler using the `-l` flag.

Both functions return a boolean value. The `CheckCLib` function is silent; the `VerboseCheckCHeader` function will use the `ConfMsgChecking` and `ConfMsgResult` functions to describe the test and the outcome.

Example:

```
static. =
    NCURSES_LIB_AVAILABLE = $(VerboseCheckCLib ncurses, initscr setupterm tigetstr)
```

13.1.8 CheckProg

```
success = $(CheckProg <prog>)
```

Checks whether the program `<prog>` exists in your path. Will use the `ConfMsgChecking` and `ConfMsgResult` functions to describe the test and the outcome.

13.2 Translating `autoconf` scripts

Some of the functions described above are very similar to the ones present in `autoconf`. Below is a brief translation table for such functions.

`AC_MSG_CHECKING` is very similar to `ConfMsgChecking` function.

`AC_MSG_RESULT` is very similar to `ConfMsgResult` function.

`AC_MSG_WARN` is very similar to `ConfMsgWarn` function.

`AC_MSG_ERROR` is very similar to `ConfMsgError` function.

`AC_TRY_COMPILE` is somewhat similar to `TryCompileC` function, except the `TryCompileC` function returns a boolean value and only works for `C`. Similarly,

`AC_TRY_LINK` is approximated by `TryLinkC` function, and

`AC_TRY_RUN` is approximated by `TryRunC` function.

13.3 Predefined configuration tests

A number of configuration tests are already included in the standard library. In order to use them in your project, simply `open` (see Section 4.7) the corresponding build file in your `OMakefile` and the tests will run the first time OMake is executed. Note that it is not a problem to `open` these files from more than one place in your project — if you do that, the test will still run only once.

13.3.1 NCurses library configuration

Add `open configure/ncurses` line to your `OMakefile` to get access to the following autoconfiguration variables.

NCURSES_AVAILABLE A boolean flag that would be set when both the `curses.h` header, the `term.h` header, and the `ncurses` library very found.

NCURSES_TERMH_IN_NCURSES A boolean flag that would be set when `term.h` has to be included as `<ncurses/term.h>` instead of `<term.h>`.

NCURSES_CFLAGS The `CFLAGS` to use when compiling ncurses code. Will include `-DNCURSES` and `-DTERMH_IN_NCURSES`, respectively when `NCURSES_AVAILABLE` and `NCURSES_TERMH_IN_NCURSES` are true.

NCURSES_CLIBS The `LDFLAGS` to use when linking ncurses code. Will normally contain `-lncurses` when ncurses is found and remain empty otherwise.

13.3.2 ReadLine library configuration

Add `open configure/readline` line to your `OMakefile` to get access to the following autoconfiguration variables.

READLINE_AVAILABLE A boolean flag that would be set when both the `readline/readline.h` header, the `readline/history.h` header, and the `readline` library very found.

READLINE_GNU A boolean flag that would be set when the GNU version of the readline library is found (as opposed to the BSD one).

READLINE_CFLAGS The `CFLAGS` to use when compiling readline code. Will include `-DREADLINE_ENABLED` and `-DREADLINE_GNU`, respectively when `READLINE_AVAILABLE` and `READLINE_GNU` are true.

READLINE_CLIBS The `LDFLAGS` to use when linking readline code. Will normally contain `-lncurses -lreadline` when readline is found and remain empty otherwise.

13.3.3 Snprintf configuration

Add `open configure/snprintf` line to your `OMakefile` to get access to the following autoconfiguration variables.

SNPRINTF_AVAILABLE A boolean flag telling whether the snprintf function is available in the standard C library.

Chapter 14

The OSH shell

OMake also includes a standalone command-line interpreter `osh` that can be used as an interactive shell. The shell uses the same syntax, and provides the same features on all platforms `omake` supports, including Win32.

14.1 Startup

On startup, `osh` reads the file `~/.oshrc` if it exists. The syntax of this file is the same as an `OMakefile`. The following additional variables are significant.

prompt The `prompt` variable specifies the command-line prompt. It can be a simple string.

```
prompt = osh>
```

Or you may choose to define it as a function of no arguments.

```
prompt() =
    return $"<$(USER):$(HOST) $(homename $(CWD))>"
```

An example of the latter prompt is as follows.

```
<jyh:kenai.yapper.org ~>cd links/omake
<jyh:kenai.yapper.org ~/links/omake>
```

If you include any "invisible" text in the prompt (such as various terminal escape sequences), they must be wrapped using the `prompt-invisible` function. For example, to create a bold prompt on terminals that support it, you can use the following.

```
prompt =
   bold-begin = $(prompt-invisible $(tgetstr bold))
   bold-end = $(prompt-invisible $(tgetstr sgr0))
   value $(bold-begin)$"osh>"$(bold-end)
```

ignoreeof If the `ignoreeof` is `true`, then `osh` will not exit on a terminal end-of-file (usually `^D` on Unix systems).

14.2 Aliases

Command aliases are defined by adding functions to the `Shell.` object. The following alias adds the `-AF` option to the `ls` command.

```
Shell. +=
   ls(argv) =
      "ls" -AF $(argv)
```

Quoted commands do not undergo alias expansion. The quotation `"ls"` prevents the alias from being recursive.

14.3 Interactive syntax

The interactive syntax in `osh` is the same as the syntax of an `OMakefile`, with one exception in regard to indentation. The line before an indented block must have a colon at the end of the line. A block is terminated with a `.` on a line by itself, or `^D`. In the following example, the first line `if true` has no body, because there is no colon.

```
# The following if has no body
osh>if true
# The following if has a body
osh>if true:
if>       if true:
if>          println(Hello world)
if>          .
Hello world
```

Note that `osh` makes some effort to modify the prompt while in an indented body, and it auto-indents the text.

The colon signifier is also allowed in files, although it is not required.

Appendix A

Synopsis

`omake` [`-j <count>`] [`-k`] [`-p`] [`-P`] [`-n`] [`-s`] [`-S`] [`-w`] [`-t`] [`-u`] [`-U`] [`-R`] [`--verbose`] [`--project`] [`--depend`] [`--progress`] [`--print-status`] [`--print-exit`] [`--print-dependencies`] [`--show-dependencies <target>`] [`--all-dependencies`] [`--verbose-dependencies`] [`--force-dotomake`] [`--dotomake <dir>`] [`--flush-includes`] [`--configure`] [`--save-interval <seconds>`] [`--install`] [`--install-all`] [`--install-force`] [`--version`] [`--absname`] [`--output-normal`] [`--output-postpone`] [`--output-only-errors`] [`--output-at-end`] `filename...` [`var-definition...`]

A.1 General usage

For Boolean options (for example, `-s`, `--progress`, etc.) the option can include a prefix `--no`, which inverts the usual sense of the option. For example, the option `--progress` means "print a progress bar," while the option `--no--progress` means "do not print a progress bar."

If multiple instances of an option are specified, the final option determines the behavior of OMake. In the following command line, the final `--no-S` cancels the earlier `-S`.

```
% omake -S --progress --no-S
```

A.2 Output control

A.2.1 -s

`-s`

Never not print commands as they are executed (be "silent").

A.2.2 -S

`-S`

Do not print commands as they are executed *unless* they produce output and/or fail. This is the default.

A.2.3 -w

-w
Print directory information in make format as commands are executed. This is mainly useful for editors that expect make-style directory information for determining the location of errors.

A.2.4 --progress

--progress
Print a progress indicator. This option is enabled by default when the OMake's output (stdout) is on a terminal and disabled by default (except on Windows) when the OMake's output is redirected.

A.2.5 --print-status

--print-status
Print status lines (the + and - lines).

A.2.6 --print-exit

--print-exit
Print termination codes when commands complete.

A.2.7 --verbose

--verbose
Make OMake very verbose. This option is equivalent to --no-S --print-status --print-exit VERBOSE=true

A.2.8 --output-normal

--output-normal
As rule commands are executed, relay their output to the OMake output right away. This is enabled by default, unless --output-postpone or --output-only-errors is enabled.

A.2.9 --output-postpone

--output-postpone
When a rule finishes, print the output as a single block. This is useful in combination -j option (see Section A.3.12), where the output of multiple subprocesses can be garbled. The diversion is printed as a single coherent unit.

Note that enabling --output-postpone will by default disable the --output-normal option. This might be problematic if you have a command that decides to ask for interactive input. If the `--output-postpone` is enabled, but the `--output-normal` is not, the prompt of such a command will not be visible and it may be hard to figure out why the build appears "stuck". You might also consider using the `--progress` flag (see Section A.2.4) so that you can see when the build is active.

A.2.10 `--output-only-errors`

`--output-only-errors`
Similar to `--output-postpone`, except that the postponed output from commands that were successful will be discarded. This can be useful in reducing unwanted output so that you can concentrate on any errors.

A.2.11 `--output-at-end`

`--output-at-end`
If any rules/commands fail, re-print the output of the failed commands when OMake finishes the build. This is especially useful when any of the `-k`, `-p`, or `-P` options are enabled.

This option is off by default. However, when `-k` is enabled — either explicitly or via one of the `-p`/`-P` options — `--output-at-end` will be enabled by default.

A.2.12 `-o`

`-o [01jwWpPxXsS]`
For brevity, the `-o` option is also provided to duplicate the above output options. The `-o` option takes a argument consisting of a sequence of characters. The characters are read from left-to-right; each specifies a set of output options. In general, an uppercase character turns the option *on*; a lowercase character turns the option *off*.

0 Equivalent to `-s --output-only-errors`

This option specifies that `omake` should be as quiet as possible. If any errors occur during the build, the output is delayed until the build terminates. Output from successful commands is discarded.

1 Equivalent to `-S --progress --output-only-errors`

This is a slightly more relaxed version of "quiet" output. The output from successful commands is discarded. The output from failed commands is printed immediately after the command complete. The output from failed commands is displayed twice: once immediately after the command completes, and again when the build completes. A progress bar is displayed so that you know when the build is active. Include the '`p`' option if you want to turn off the progress bar (for example `omake -o 1p`).

2 Equivalent to `--progress --output-postpone`

The is even more relaxed, output from successful commands is printed. This is often useful for deinterleaving the output when using `-j`.

W Equivalent to `-w`

w Equivalent to `--no-w`

P Equivalent to `--progress`

p Equivalent to `--no--progress`

X Equivalent to `--print-exit`

x Equivalent to `--no-print-exit`

S Equivalent to `-S`

s Equivalent to `--no-S`

A.3 Build options

A.3.1 `-k`

`-k`

Do not abort when a build command fails; continue to build as much of the project as possible. This option is implied by both `-p` and `-P` options. In turn, this option would imply the `--output-at-end` option.

A.3.2 `-n`

`-n`

This can be used to see what would happen if the project were to be built.

A.3.3 `-p`

`-p`

Watch the filesystem for changes, and continue the build until it succeeds. If this option is specified, `omake` will restart the build whenever source files are modified. Implies `-k`.

A.3.4 `-P`

`-P`

Watch the filesystem for changes forever. If this option is specified, `omake` will restart the build whenever source files are modified. Implies `-k`.

A.3.5 `-R`

`-R`

Ignore the current directory and build the project from its root directory. When `omake` is run in a subdirectory of a project and no explicit targets are given on the command line, it would normally only build files within the current directory and its subdirectories (more precisely, it builds all the `.DEFAULT` targets in the current directory and its subdirectories). If the `-R` option is specified, the build is performed as if `omake` were run in the project root.

In other words, with the `-R` option, all the relative targets specified on the command line will be taken relative to the project root (instead of relative to the current directory). When no targets are given on the command line, all the `.DEFAULT` targets in the project will be built (regardless of the current directory).

A.3.6 `-t`

`-t`

Update the `omake` database to force the project to be considered up-to-date.

A.3.7 `-U`

`-U`

Do not trust cached build information. This will force the entire project to be rebuilt.

A.3.8 `--depend`

`--depend`

Do not trust cached dependency information. This will force files to be rescanned for dependency information.

A.3.9 `--configure`

`--configure`

Re-run `static.` sections of the included omake files, instead of trusting the cached results.

A.3.10 `--force-dotomake`

`--force-dotomake`

Always use the `$HOME/.omake` for the `.omc` cache files.

A.3.11 `--dotomake`

`--dotomake <dir>`

Use the specified directory instead of the `$HOME/.omake` for the placement of the `.omc` cache files.

A.3.12 `-j`

`-j <count>`

Run multiple build commands in parallel. The `count` specifies a bound on the number of commands to run simultaneously. In addition, the count may specify servers for remote execution of commands in the form `server=count`. For example, the option `-j 2:small.host.org=1:large.host.org=4` would specify that up to 2 jobs can be executed locally, 1 on the server `small.host.org` and 4 on `large.host.org`. Each remote server must use the same filesystem location for the project.

Remote execution is currently an experimental feature. Remote filesystems like NFS do not provide adequate file consistency for this to work.

A.3.13 `--print-dependencies`

`--print-dependencies`

Print dependency information for the targets on the command line.

A.3.14 `--show-dependencies`

`--show-dependencies <target>`

Print dependency information *if* the `target` is built.

A.3.15 `--all-dependencies`

`--all-dependencies`

If either of the options `--print-dependencies` or `--show-dependencies` is in effect, print transitive dependencies. That is, print all dependencies recursively. If neither option `--print-dependencies`, `--show-dependencies` is specified, this option has no effect.

A.3.16 `--verbose-dependencies`

`--verbose-dependencies`

If either of the options `--print-dependencies` or `--show-dependencies` is in effect, also print listings for each dependency. The output is very verbose, consider redirecting to a file. If neither option `--print-dependencies`, `--show-dependencies` is specified, this option has no effect.

A.3.17 `--install`

`--install`

Install default files `OMakefile` and `OMakeroot` into the current directory. You would typically do this to start a project in the current directory.

A.3.18 `--install-all`

`--install-all`

In addition to installing files `OMakefile` and `OMakeroot`, install default `OMakefile`s into each subdirectory of the current directory. `cvs`(1) rules are used for filtering the subdirectory list. For example, `OMakefile`s are not copied into directories called `CVS`, `RCCS`, etc.

A.3.19 `--install-force`

`--install-force`

Normally, `omake` will prompt before it overwrites any existing `OMakefile`. If this option is given, all files are forcibly overwritten without prompting.

A.3.20 `--absname`

`--absname`

Filenames should expand to absolute pathnames.

N.B. This is an experimental option. It may become deprecated.

A.3.21 `variable definition`

`name=[value]`

`omake` variables can also be defined on the command line in the form `name=value`. For example, the `CFLAGS` variable might be defined on the command line with the argument `CFLAGS="-Wall -g"`.

A.4 Additional options

In addition, `omake` supports a number of debugging flags on the command line. Run `omake --help` to get a summary of these flags.

A.5 Environment variables

A.5.1 `OMAKEFLAGS`

If defines, the `OMAKEFLAGS` should specify a set of options exactly as they are specified on the command line.

A.5.2 OMAKELIB

If defined, the `OMAKELIB` environment variable should refer to the installed location of the OMake standard library. This is the directory that contains `Pervasives.om` etc. On a Unix system, this is often `/usr/lib/omake` or `/usr/local/lib/omake`, and on Win32 systems it is often `c:\Program Files\OMake\lib`.

If not defined, `omake` uses the default configured location. You should normally leave this unset.

A.6 Functions

A.6.1 OMakeFlags

The `OMakeFlags` function can be used within an `OMakefile` to modify the set of options. The options should be specified exactly as they are on the command line. For example, if you want some specific project to be silent and display a progress bar, you can add the following line to your `OMakefile`.

```
OMakeFlags(-S --progress)
```

For options where it makes sense, the options are scoped like variables. For example, if you want OMake to be silent for a single rule (instead of for the entire project), you can use scoping the restrict the range of the option.

```
section
    # Do not display command output when foo is constructed
    OMakeFlags(-S)

    foo: fee
       echo "This is a generated file" > foo
       cat fee >> foo
       chmod 555 foo
```

A.7 Option processing

When `omake` is invoked, the options are processed in the following order.

1. All options specified by the `OMAKEFLAGS` environment variable are defined globally.

2. All options from the command line are defined globally.

3. Any individual calls the the `OMakeFlags` function modify the options locally.

A.8 .omakerc

If the `$(HOME)/.omakerc` exists, it is read before any of the `OMakefiles` in your project. The `.omakerc` file is frequently used for user-specific customization. For example, instead of defining the `OMAKEFLAGS` environment variable, you could add a line to your `.omakerc`.

```
$(HOME)/.omakerc:
    # My private options
    OMakeFlags(-S --progress)
```

Appendix B

OMake grammar

B.1 OMake lexical conventions

The OMake language is based on the language for GNU/BSD make, where there are few lexical conventions. Strictly speaking, there are no keywords, and few special symbols.

B.1.1 Comments

Comments begin with the `#` character and continue to the end-of-line. Text within a comment is unrestricted.

Examples.

```
# This is a comment
# This $comment contains a quote " character
```

B.1.2 Special characters

The following characters are special in some contexts.

```
$  (  )  ,  .  =  :  "  '  `  \  #
```

- `$` is used to denote a variable reference, or function application.
- Parentheses `)`, `(` are argument deliminters.
- The command `,` is an argument separator.
- The period symbol `.` is a name separator.
- The equality symbol `=` denotes a definition.
- The colon symbol `:` is used to denote rules, and (optionally) to indicate that an expression is followed by an indented body.

- The quotation symbols `"` and `'` delimit character strings.
- The symbol `#` is the first character of a constant.
- The escape symbol `\` is special *only when* followed by another special character. In this case, the special status of the second character is removed, and the sequence denotes the second character. Otherwise, the `\` is not special.

  Examples:

  - `\$`: the `$` character (as a normal character).
  - `\#`: the `#` character (as a normal character).
  - `\\`: the `\` character (as a normal character).
  - `c\:\Windows\moo\#boo`: the string `c:\Windows\moo#boo`.

B.1.3 Identifiers

Identifiers (variable names) are drawn from the ASCII alphanumeric characters as well as `_`, `-`, `~`, `@`. Case is significant; the following identifiers are distinct: `FOO`, `Foo`, `foo`. The identifier may begin with any of the valid characters, including digits.

Using `egrep` notation, the regular expression for identifiers is defined as follows.

```
identifier ::= [-@~_A-Za-z0-9]+
```

The following are legal identifiers.

```
Xyz    hello_world    seventy@nine
79-32  Gnus~Gnats     CFLAGS
```

The following are not legal identifiers.

```
x+y    hello&world
```

B.1.4 Command identifiers

The following words have special significance when they occur as the *first* word of a program line. They are not otherwise special.

```
case     catch  class    declare    default
do       else   elseif   export     extends
finally  if     import   include    match
open     raise  return   section    switch
try      value  when     while
```

B.1.5 Variable references

A variable reference is denoted with the `$` special character followed by an identifier. If the identifier name has more than one character, it must be enclosed in parentheses. The parenthesized version is most common. The following are legal variable references.

```
$(Xyz)    $(hello_world)   $(seventy@nine)
$(79-32)  $(Gnus~Gnats)    $(CFLAGS)
```

Single-character references also include several additional identifiers, including `&*<^?][`. The following are legal single-character references.

```
$@   $&   $*   $<   $^   $+   $?   $[   $]
$A   $_   $a   $b   $x   $1   $2   $3
```

Note that a non-parenthesized variable reference is limited to a single character, even if it is followed by additional legal identifier charqcters. Suppose the value of the `$x` variable is 17. The following examples illustrate evaluation.

```
$x           evaluates to    17
foo$xbar     evaluates to    foo17bar
foo$(x)bar   evaluates to    foo17bar
```

The special sequence `$$` represents the character literal `$`. That is, the two-character sequences `\$` and `$$` are normally equalivalent.

B.1.6 String constants

Literal strings are defined with matching string delimiters. A left string delimiter begins with the dollar-sign `$`, and a non-zero number of single-quote or double-quote characters. The string is terminated with a matching sequence of quotation symbols. The delimiter quotation may not be mixed; it must contain only single-quote characters, or double-quote characters. The following are legal strings.

```
    $'Hello world'
    $"""printf("Hello world\n")"""
    $''''
Large "block" of
text # spanning ''multiple'' lines''''
```

The string delimiters are *not* included in the string constant. In the single-quote form, the contents of the string are interpreted verbatim–there are no special characters.

The double-quote form permits expression evaluation within the string, denoted with the `$` symbol. The following are some examples.

```
X = Hello
Y = $""$X world""              # Hello world
Z = $'''$X world'''            # $X world
I = 3
W = $"6 > $(add $I, 2)"        # 6 > 5
```

Note that quotation symbols without a leading `$` are not treated specially by OMake. The quotation symbols is included in the sequence.

```
osh>println('Hello world')
'Hello world'
osh>println($'Hello world')
Hello world
osh>X = Hello
- : "Hello" : Sequence
osh>println('$X world')
Hello world
```

B.2 The OMake grammar

OMake programs are constructed from expressions and statements. Generally, an input program consists of a sequence of statements, each of which consists of one or more lines. Indentation is significant–if a statement consists of more than one line, the second and remaining lines (called the *body*) are usually indented relative to the first line.

B.2.1 Expressions

The following table lists the syntax for expressions.

expr	::=	
		(empty)
		– Text (see note)
	—	*text*
	—	*string-literal*
		– Applications
	—	*dollar* <char>
	—	*dollar* (*pathid* *args*)
		– Concatenation
	—	*expr* *expr*
dollar	::=	$ — $‘ — $,
pathid	::=	
		id
	—	*pathid* . *id*
arg	::=	*expr* – excluding special characters)(,)
args	::=	*(empty)* — *arg*, ..., *arg*

An *expression* is a sequence composed of text, string-literals, variables references and function applications. Text is any sequence of non-special characters.

B.2.1.1 Inline applications

An *application* is the application of a function to zero-or-more arguments. Inline applications begin with one of the "dollar" sequences `$`, `` $` ``, or `$,`. The application itself is specified as a single character (in which case it is a variable reference), or it is a parenthesized list including a function identifier *pathid*, and zero-or-more comma-separated arguments *args*. The arguments are themselves a variant of the expressions where the special character `)(,` are not allowed (though any of these may be made non-special with the `\` escape character). The following are some examples of valid expressions.

- `xyz abc`

  The text sequence "`xyz abc`"

- `xyz$wabc`

  A text sequence containing a reference to the variable `w`.

- `$(addsuffix .c, $(FILES))`

  An application of the function `addsuffix`, with first argument `.c`, and second argument `$(FILES)`.

- `$(a.b.c 12)`

  This is a method call. The variable `a` must evaluate to an object with a field `b`, which must be an object with a method `c`. This method is called with argument `12`.

The additional dollar sequences specify evaluation order, `` $` `` (lazy) and `$,` (eager), as discussed in the section on dollar modifiers (Section B.3).

B.2.2 Statements and programs

The following table lists the syntax of statements and programs.

params	::=	*(empty)* — *id*, ..., *id*
target	::=	*expr* – excluding special character :
program	::=	*stmt* <eol> ... <eol> *stmt*
stmt	::=	
		– Special forms
	—	command *expr optcolon-body*
	—	command (*args*) *optcolon-body*
	—	catch *id* (*id*) *optcolon-body*
	—	class *id* ... *id*
		– Variable definitions
	—	*pathid* {+}= *expr*
	—	*pathid* {+}= <eol> *indented-body*
	—	*pathid* [] {+}= *expr*
	—	*pathid* [] {+}= <eol> *indented-exprs*
		– Functions
	—	*pathid*(*args*) *optcolon-body*
	—	*pathid*(*params*) = <eol> *indented-body*
		– Objects
	—	*pathid* . {+}= <eol> *indented-body*
		– Rules
	—	*target* : *target rule-options* <eol> *indented-body*
	—	*target* :: *target rule-options* <eol> *indented-body*
	—	*target* : *target* : *target rule-options* <eol> *indented-body*
	—	*target* :: *target* : *target rule-options* <eol> *indented-body*
		– Shell commands
	—	*expr*
indented-body	::=	*(empty)*
	—	*indented-stmt* <eol> ... <eol> *indented-stmt*
indented-exprs	::=	*(empty)*
	—	*indented-expr* <eol> ... <eol> *indented-expr*
optcolon-body	::=	*(empty)*
	—	<eol> *indented-body*
	—	: <eol> *indented-body*
rule-option	::=	*:id: target*
rule-options	::=	*(empty)*
	—	*rule-options rule-option*

B.2.2.1 Special forms

The special forms include the following.

Conditionals (see the section on conditionals — Section 4.9). The `if` command should be followed by an expression that represents the condition, and an indented body. The conditional may be followed by `elseif` and `else` blocks.

```
if expr
    indented-body
elseif expr
    indented-body
...
else
    indented-body
```

matching (see the section on matching — Section 4.10). The `switch` and `match` commands perform pattern-matching. All cases are optional. Each case may include `when` clauses that specify additional matching conditions.

```
match(expr)
case expr
   indented-body
when expr
   indented-body
...
case expr
   indented-body
default
   indented-body
```

Exceptions (see also the `try` function documentation). The `try` command introduces an exception handler. Each `name` is the name of a class. All cases, including `catch`, `default`, and `finally` are optional. The `catch` and `default` clauses contain optional `when` clauses.

```
try
    indented-body
catch name1(id1)
    indented-body
when expr
    indented-body
...
catch nameN(idN)
    indented-body
default
    indented-body
finally
    indented-body
```

The `raise` command is used to raise an exception.

```
raise expr
```

section (see the `section` description in Section 4.8). The `section` command introduces a new scope.

```
section
    indented-body
```

include, open (see also Section 4.7). The `include` command performs file inclusion. The expression should evaluate to a file name.

The `open` form is like include, but it performs the inclusion only if the inclusion has not already been performed. The `open` form is usually used to include library files. [jyh– this behavior will change in subsequent revisions.]

```
include expr
open expr
```

return (see the description of functions in Section 4.5). The `return` command terminates execution and returns a value from a function.

```
return expr
```

value (see the description of functions in Section 4.5). The `value` command is an identity. Syntactically, it is used to coerce a n expression to a statement.

```
value expr
```

export (see the section on scoping — Section 5.3). The `export` command exports a environment from a nested block. If no arguments are given, the entire environment is exported. Otherwise, the export is limited to the specified identifiers.

```
export expr
```

while (see also the `while` function description). The `while` command introduces a `while` loop.

```
while expr
    indented-body
```

class, extends (see the section on objects — Section 4.11). The `class` command specifies an identifier for an object. The `extends` command specifies a parent object.

```
class id
extends expr
```

B.2.2.2 Variable definitions

See the section on variables (Section 4.1). The simplest variable definition has the following syntax. The `=` form is a new definition. The += form appends the value to an existing definition.

```
id = expr
id += expr

osh> X = 1
- : "1" : Sequence
osh> X += 7
- : "1" " " "7" : Sequence
```

A multi-line form is allowed, where the value is computed by an indented body.

```
id {+}=
    indented-body

osh> X =
         Y = HOME
         println(Y is $Y)
         getenv($Y)
Y is HOME
- : "/home/jyh" : Sequence
```

The name may be qualified qith one of the `public`, `prtected`, or `private` modifiers. Public variables are dynamically scoped. Protected variables are fields in the current object. Private variables are statically scoped.

[jyh: revision 0.9.9 introduces modular namespaces; the meaning of these qualifiers is slightly changed.]

```
public.X = $(addsuffix .c, 1 2 3)
protected.Y = $(getenv HOME)
private.Z = $"Hello world"
```

B.2.2.3 Applications and function definitions

See the section on functions (Section 4.5). A function-application statement is specified as a function name, followed a parenthesized list of comma-separated arguments.

```
osh> println($"Hello world")

osh> FILES = 1 2 3
- : 1 2 3
osh> addsuffix(.c, $(FILES))
```

```
- : 1.c 2.c 3.c

# The following forms are equivalent
osh> value $(println $"Hello world")
osh> value $(addsuffix .c, $(FILES))
- : 1.c 2.c 3.c
```

If the function application has a body, the body is passed (lazily) to the function as its first argument. [jyh: in revision 0.9.8 support is incomplete.] When using `osh`, the application must be followed by a colon `:` to indicate that the application has a body.

```
# In its 3-argument form, the foreach function takes
# a body, a variable, and an array.  The body is evaluated
# for each element of the array, with the variable bound to
# the element value.
#
# The colon is required only for interactive sessions.
osh> foreach(x, 1 2 3):
        add($x, 1)
- : 2 3 4
```

Functions are defined in a similar form, where the parameter list is specified as a comma-separated list of identifiers, and the body of the function is indented.

```
osh> f(i, j) =
        add($i, $j)
- : <fun 2>
osh> f(3, 7)
- : 10 : Int
```

B.2.2.4 Objects

See the section on objects (Section 4.11). Objects are defined as an identifier with a terminal period. The body of the object is indented.

```
Obj. =
    class Obj

    X = 1
    Y = $(sub $X, 12)
    new(i, j) =
       X = $i
       Y = $j
       value $(this)
    F() =
       add($X, $Y)
    println($Y)
```

The body of the object has the usual form of an indented body, but new variable definitions are added to the object, not the global environment. The object definition above defines an object with (at least) the fields `X` and `Y`, and methods `new` and `F`. The name of the object is defined with the `class` command as `Obj`.

The `Obj` itself has fields `X = 1` and `Y = -11`. The `new` method has the typical form of a constructor-style method, where the fields of the object are initialized to new values, and the new object returned (`$(this)` refers to the current object).

The `F` method returns the sum of the two fields `X` and `Y`.

When used in an object definition, the += form adds the new definitions to an existing object.

```
pair. =
   x = 1
   y = 2

pair. +=
   y = $(add $y, 3)
# pair now has fields (x = 1, and y = 5)
```

The `extends` form specifies inheritance. Multiple inheritance is allowed. At evaluation time, the `extends` directive performs inclusion of the entire parent object.

```
pair. =
   x = 1
   y = 2

depth. =
   z = 3
   zoom(dz) =
      z = $(add $z, $(dz))
      return $(this)

triple. =
   extends $(pair)
   extends $(depth)

   crazy() =
      zoom($(mul $x, $y))
```

In this example, the `triple` object has three fields x, y, and z; and two methods `zoom` and `crazy`.

B.2.2.5 Rules

See the chapter on rules (Chapter 7). A rule has the following parts.

1. A sequence of targets;
2. one or two colons;
3. a sequence of *dependencies* and *rule options*;
4. and an indented body.

The targets are the files to be built, and the dependencies are the files it depends on. If two colons are specified, it indicates that there may be multiple rules to build the given targets; otherwise only one rule is allowed.

If the target contains a `%` character, the rule is called *implicit*, and is considered whenever a file matching that pattern is to be built. For example, the following rule specifies a default rule for compiling OCaml files.

```
%.cmo: %.ml %.mli
   $(OCAMLC) -c $<
```

This rule would be consulted as a default way of building any file with a `.cmo` suffix. The dependencies list is also constructed based on the pattern match. For example, if this rule were used to build a file `foo.cmo`, then the dependency list would be `foo.ml foo.mli`.

There is also a three-part version of a rule, where the rule specification has three parts.

```
targets : pattern : dependencies rule-options
   indented-body
```

In this case, the pattern *must* contain a single `%` character. However this is considered to be a sequence of *explicit* rules, where each target is matched against the pattern, and a new rule is computed based on the pattern match. For example, the following rule specifies how to build the explicit targets `a.cmo` and `b.cmo`.

```
a.cmo b.cmo: %.cmo: %.ml %.mli
   $(OCAMLC) -c $<
```

This example is equivalent to the following two-rule sequence.

```
a.cmo: a.ml a.mli
   $(OCAMLC) -c $<
b.cmo: b.ml b.mli
   $(OCAMLC) -c $<
```

There are several special targets, including the following.

- `.PHONY` : declare a "phony" target. That is, the target does not correspond to a file.
- `.ORDER` : declare a rule for dependency ordering.

- `.INCLUDE` : define a rule to generate a file for textual inclusion.
- `.SUBDIRS` : specify subdirectories that are part of the project.
- `.SCANNER` : define a rule for dependency scanning.

There are several rule options.

- `:optional: dependencies` the subsequent dependencies are optional, it is acceptable if they do not exist.
- `:exists: dependencies` the subsequent dependencies must exist, but changes to not affect whether this rule is considered out-of-date.
- `:effects: targets` the subsequent files are side-effects of the rule. That is, they may be created and/or modified while the rule is executing. Rules with overlapping side-effects are never executed in parallel.
- `:scanner: name` the subsequent name is the name of the `.SCANNER` rule for the target to be built.
- `:value: expr` the `expr` is a "value" dependency. The rule is considered out-of-date whenever the value of the `expr` changes.

Several variables are defined during rule evaluation.

- `$*` : the name of the target with the outermost suffix removed.
- `$>` : the name of the target with all suffixes removed.
- `$@` : the name of the target.
- `$^` : the explicit file dependencies, sorted alphabetically, with duplicates removed.
- `$+` : all explicit file dependencies, with order preserved.
- `$<` : the first explicit file dependency.
- `$&` : the free values of the rule (often used in `:value:` dependencies).

B.2.2.6 Shell commands

See the chapter on shell commands (Chapter 10).

While it is possible to give a precise specification of shell commands, the informal description is simpler. Any non-empty statement where each prefix is *not* one of the other statements, is considered to be a shell command. Here are some examples.

```
ls                                  -- shell command
echo Hello world > /dev/null        -- shell command
echo(Hello world)                   -- function application
echo(Hello world) > /dev/null       -- syntax error
echo Hello: world                   -- rule
X=1 getenv X                        -- variable definition
env X=1 getenv X                    -- shell command
if true                             -- special form
\if true                            -- shell command
"if" true                           -- shell command
```

B.3 Dollar modifiers

Inline applications have a function and zero-or-more arguments. Evaluation is normally strict: when an application is evaluated, the function identifier is evaluated to a function, the arguments are then evaluated and the function is called with the evaluated arguments.

The additional "dollar" sequences specify additional control over evaluation. The token `` $` `` defines a "lazy" application, where evaluation is delayed until a value is required. The `$,` sequence performs an "eager" application within a lazy context.

To illustrate, consider the expression `$(addsuffix .c, $(FILES))`. The `addsuffix` function appends its first argument to each value in its second argument. The following `osh` interaction demonstrates the normal bahavior.

```
osh> FILES[] = a b c
- : <array a b c>
osh> X = $(addsuffix .c, $(FILES))
- : <array ...>
osh> FILES[] = 1 2 3 # redefine FILES
- : <array 1 2 3>
osh> println($"$X")  # force the evaluation and print
a.c b.c c.c
```

When the lazy operator `` $` `` is used instead, evaluation is delayed until it is printed. In the following sample, the value for `X` has changed to the `$(apply ..)` form, but otherwise the result is unchanged because it it printed immediately.

```
osh> FILES[] = a b c
- : <array a b c>
osh> SUF = .c
- : ".c"
osh> X = $`(addsuffix $(SUF), $(FILES))
- : $(apply global.addsuffix ...)
osh> println($"$X")  # force the evaluation and print
a.c b.c c.c
```

However, consider what happens if we redefine the `FILES` variable after the definition for `X`. In the following sample, the result changes because evaluation occurs *after* the values for `FILES` has been redefined.

```
osh> FILES[] = a b c
- : <array a b c>
osh> SUF = .c
- : ".c"
osh> X = $`(addsuffix $(SUF), $(FILES))
- : $(apply global.addsuffix ...)
osh> SUF = .x
osh> FILES[] = 1 2 3
osh> println($"$X")  # force the evaluation and print
1.x 2.x 3.x
```

In some cases, more explicit control is desired over evaluation. For example, we may wish to evaluate `SUF` early, but allow for changes to the `FILES` variable. The `$,(SUF)` expression forces early evaluation.

```
osh> FILES[] = a b c
- : <array a b c>
osh> SUF = .c
- : ".c"
osh> X = $`(addsuffix $,(SUF), $(FILES))
- : $(apply global.addsuffix ...)
osh> SUF = .x
osh> FILES[] = 1 2 3
osh> println($"$X")  # force the evaluation and print
1.c 2.c 3.c
```

Index

Appendix C

References

C.1 See Also

omake(1) (Chapter 1), osh(1) (Chapter 14), make(1)

C.2 Version

Version: 0.9.8.1 of 15th March 2007.

C.3 License and Copyright

C.4 Author

Jason Hickey, Aleksey Nogin, *et. al.*
Caltech 256-80
Pasadena, CA 91125, USA
Email: `omake-devel@metaprl.org`
WWW: `http://www.cs.caltech.edu/~jyh` and `http://nogin.org/`