The OMake build system

Jason Hickey *et. al.*

April 11, 2006
Version 0.9.6.9

Abstract

omake is a build system designed to scale from small projects to very large projects spanning many directories. *omake* uses a syntax similar to *make*(1), with many additional features, including accurate automated dependency analysis based on MD5 digests.

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 `OMakefile`s specify the build parameters specific to each of the subdirectories. When *omake* runs, it walks the configuration tree, evaluating rules from all of the `OMakefile`s. The project is then built from the entire collection of build rules.

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.

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.

See the following manual pages for more information.

omake-quickstart A quickstart guide to using *omake*.

omake-options Command-line options for *omake*.

omake-root The system `OMakeroot` contains the default specification of how to build C, OCaml, and LaTeX programs.

omake-language The *omake* language, including a description of objects, expressions, and values.

omake-shell Using the *omake* shell for command-line interpretation.

omake-rules Using *omake* rules to build program.

omake-base Functions and variables in the core standard library.

omake-system Functions on files, input/output, and system commands.

omake-pervasives Pervasives defines the built-in objects.

osh The *osh* command-line interpreter.

omake-doc All the OMake documentation in a single page.

2 OMake quickstart guide

2.1 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 function (like `StaticCLibrary` and `CProgram`) 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 before they are used.
- Subdirectories are incorporated into a project using the `.SUBDIRS:` target.

2.2 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 `OMakefiles`.

```
$ 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)
```

2.3 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)
```

2.4 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 `OMakefiles` 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`.

2.5 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`.

2.6 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))
```

3 Notes

3.1 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.

4 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.

4.1 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.

5 Synopsis

omake [**-k**] [**-j***count*] [**-n**] [**-s**] [**-S**] [**-p**] [**-P**] [**-w**] [**-t**] [**-u**] [**-U**] [**-R**] [–**project**] [–**progress**] [–**no-progress**] [–**print-status**] [–**no-print-status**] [–**print-exit**] [–**no-print-exit**] [–**print-dependencies**] [–**show-dependencies** *target*] [–**force-dotomake**] [–**dotomake** *dir*] [–**flush-includes**] [–**configure**] [–**install**] [–**install-all**] [–**install-force**] [–**version**] [*filename...*] [**var-definition...**]

6 Command-line options

-k Do not abort when a build command fails; continue to build as much of the project as possible.

-n Print the commands that would be executed, but do no execute them. This can be used to see what would happen if the project were to be built.

-s Do not print commands as they are executed (be “silent”).

-S Do not print commands as they are executed *unless* they produce output.

–progress Print a progress indicator. This is normally used with the **-s** or **-S** options.

–no-progress Do not print a progress indicator (default).

–print-exit Print termination codes when commands complete.

–no-print-exit Do not print termination codes when commands complete (default).

-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.

-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.

-P Watch the filesystem for changes forever. If this option is specified, *omake* will restart the build whenever source files are modified.

-R Ignore the current directory and build the project from its root directory. When *omake* is run in a subdirectory of a project, it normally builds files within 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.

-t Update the *omake* database to force the project to be considered up-to-date.

-U Do not trust cached build information. This will force the entire project to be rebuilt.

–depend Do not trust cached dependency information. This will force files to be rescanned for dependency information.

–configure Re-run `static.\` sections of the included omake files, instead of trusting the cached results.

[–**force-dotomake**] Always use the `$HOME/.omake` for the `.omc` cache files.

[–**dotomake *dir***] Use the specified directory instead of the `$HOME/.omake` for the placement of the `.omc` cache files.

-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.

–print-dependencies Print dependency information for the targets on the command line.

–show-dependencies *target* Print dependency information *if* the `target` is built.

–install Install default files `OMakefile` and `OMakeroot` into the current directory. You would typically do this to start a project in the current directory.

–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.

–install-force Normally, *omake* will prompt before it overwrites any existing `OMakefile`. If this option is given, all files are forcibly overwritten without prompting.

var-definition *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"`.

In addition, *omake* supports a number of debugging flags on the command line. Run `omake --help` to get a summary of these flags.

7 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.

7.1 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`.

7.2 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
```

7.3 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)))
```

7.4 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"""
```

7.5 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)
```

7.6 Comments

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

7.7 File inclusion

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

```
include files.omake
```

7.8 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 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.

7.9 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 the Logic section), 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.

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)
```

7.10 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)
```

8 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.

8.1 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.

8.2 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)
```

9 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.

9.1 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
```

9.2 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.

9.3 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()
```

9.4 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)')
```

As a matter of style, a `static.` section that is used for configuration should print what it is doing, using `---` as a print prefix.

9.5 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))
```

9.6 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):
```

10 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).

10.1 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).

10.2 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
```

10.3 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 `$""...""` 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"); }"")
```

10.4 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
```

11 Special dependencies

11.1 :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 $@
```

11.2 :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
```

11.3 :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.

12 .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 scanner produces dependencies for the specified target (which may be a pattern) by running the commands, which *must* produce output that is compatible with omake. 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)) $<
```

12.1 Named scanners, and the :scanner: target

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.

12.2 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. See the documentation for the `SCANNER_MODE` variable in *omake-root*(1) for detail.

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.

13 Other special targets

There are several other special targets that define special actions to be take by *omake*.

13.1 .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
```

13.2 .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
```

13.3 .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)
```

13.4 .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
```

14 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`).

14.1 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.

14.2 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`.

14.3 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 sbdirectories, and 2) you can build them from the project root.

```
.PHONY: all install clean

.SUBDIRS: src lib clib
```

15 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.

15.1 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>
```

ignoreeof If the `ignoreeof` is `true`, then `osh` will not exit on a terminal end-of-file (usually `^D` on Unix systems).

15.2 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.

15.3 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.

15.4 See also

See Section omake-shell for more information on the shell language, and Section omake-system for more information on job control.

16 Builtin variables

16.1 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).

16.2 SYSNAME

The name of the operating system for the current machine.

16.3 NODENAME

The hostname of the current machine.

16.4 OS_VERSION

The operating system release.

16.5 MACHINE

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

16.6 HOST

Same as `NODENAME`.

16.7 OMAKE_VERSION

Version of OMake.

16.8 USER

The login name of the user executing the process.

16.9 HOME

The home directory of the user executing the process.

16.10 PID

The OMake process id.

17 Boolean functions and control flow

17.1 not

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

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. The `not` function negates a Boolean value.

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

17.2 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`.

17.3 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)))
```

17.4 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))
```

17.5 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
```

17.6 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)
```

17.7 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.

17.8 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.

17.9 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.

17.10 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
```

17.11 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
```

17.12 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
```

17.13 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.

17.14 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
```

17.15 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)
```

17.16 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)
```

18 Arrays and sequences

18.1 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.

18.2 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.

18.3 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))
```

18.4 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.

18.5 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"`.

18.6 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"`.

18.7 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`.

18.8 sub

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

The `sub` 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 `$(sub 1, 2, a "b c" d e)` evaluates to `"b c" d`.

18.9 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`.

18.10 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.

18.11 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"`.

18.12 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.

18.13 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.

18.14 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`.

18.15 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`.

18.16 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`.

18.17 removeprefix

```
$(removeprefix prefix, sequence) : Array
   prefix : String
   sequence : Array
```

The `removeprefix` function removes a prefix from each component of a sequence.

18.18 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"`.

18.19 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`.

18.20 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"`.

18.21 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"`.

18.22 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.

18.23 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`.

18.24 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`.

18.25 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`.

18.26 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`.

18.27 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`.

18.28 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`.

18.29 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`.

18.30 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`.

18.31 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`.

18.32 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`.

18.33 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`.

18.34 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
```

18.35 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).

19 Arithmetic

19.1 int

The `int` function can be used to create integers. It returns an `Int` object.
`$(int 17)`.

19.2 float

The `float` function can be used to create floating-point numbers. It returns a `Float` object.
`$(float 3.1415926)`.

19.3 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.

19.4 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.

20 First-class functions

20.1 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))
```

20.2 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)
```

20.3 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))
```

21 Iteration and mapping

21.1 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
```

22 File operations

22.1 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)
```

22.2 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.

22.3 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.

22.4 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.

22.5 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
```

22.6 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.

22.7 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`.

22.8 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`.

22.9 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 /`.

22.10 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.

22.11 absname

```
$(absname files) : String Sequence
   files : File Sequence
```

The `absname` function returns the absolute pathname for each of the files or directories.

22.12 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.

22.13 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`.

22.14 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.

22.15 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`.

22.16 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.

22.17 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`.

22.18 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>)`

22.19 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 .

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.

22.20 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.

22.21 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.

22.22 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.

22.23 Stat

The `Stat` object represents the result returned by the `stat` and `lstat` functions. It contains the following fields.

A `stat` object has the following fields. Not all of the fields will have meaning on all architectures.

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.

22.24 stat

```
$(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.

22.25 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.

22.26 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.

22.27 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 are not supported in Win32.
Normally, only the superuser can create hard links to directories.

22.28 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.

22.29 readlink

```
$(readlink node...) : Node
   node : Node
```

The `readlink` function reads the value of a symbolic link.

22.30 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. The `chmod` function does nothing on Win32 platforms.
Options:

-v Explain what is happening.

-r Change files and directories recursively.

-f Continue on errors.

– Interpret the remaining argument literally.

22.31 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.

22.32 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.

22.33 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.

22.34 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.

22.35 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.

22.36 rehash

```
rehash()
```

The `rehash` function resets all search paths.

22.37 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.

22.38 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.

22.39 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.

22.40 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.

22.41 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 regular expression.

23 IO functions

23.1 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.

23.2 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.

23.3 close

```
$(close channel...)
   channel : Channel
```

The `close` function closes a file that was previously opened with `fopen`.

23.4 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.

23.5 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.

23.6 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.

23.7 rewind

```
rewind(channel...)
   channel : Channel
```

The `rewind` function set the current file position to the beginning of the file.

23.8 tell

```
$(tell channel...) : Int...
   channel : Channel
raises RuntimeException
```

The `tell` function returns the current position of the `channel`.

23.9 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.

23.10 dup

```
$(dup channel) : Channel
   channel : Channel
raises RuntimeException
```

The `dup` function returns a new channel referencing the same file as the argument.

23.11 dup2

```
dup2(channel1, channel2)
   channel1 : Channel
   channel2 : Channel
raises RuntimeException
```

The `dup2` function causes `channel2` to refer to the same file as `channel1`.

23.12 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`.

23.13 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.

23.14 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.

23.15 mkfifo

```
mkfifo(mode, node...)
   mode : Int
   node : Node
```

The `mkfifo` function creates a named pipe.

23.16 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.

23.17 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.

23.18 InetAddr

The `InetAddr` object describes an Internet address. It contains the following fields.

addr `String`: the Internet address.

port `Int`: the port number.

23.19 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.

23.20 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.

23.21 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.

23.22 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.

23.23 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.

23.24 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.

23.25 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.

23.26 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.

23.27 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.

23.28 accept

```
$(accept socket) : InOutChannel
   socket : InOutChannel
raises RuntimeException
```

The `accept` function accepts a connection on a socket.

23.29 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.

23.30 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`.

23.31 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.

23.32 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.

23.33 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.

23.34 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.

24 Higher-level IO functions

24.1 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.

24.2 cat

```
cat(files) : Sequence
   files : File or InChannel Sequence
```

The `cat` function concatenates the output from multiple files and returns it as a string.

24.3 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.

n If specified, output lines include the filename.

The `pattern` is a regular expression.

If successful (`grep` found a match), the function returns `true`. Otherwise, it returns `false`.

24.4 awk

```
awk(input-files)
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

The `awk` function provides input processing similar to *awk*(1), but more limited. The function takes 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 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)
```

24.5 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)
```

24.6 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.

24.7 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.

24.8 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.

24.9 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.

24.10 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.

24.11 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.

24.12 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.

24.13 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.

24.14 gettimeofday

```
$(gettimeofday) : Float
```

The `gettimeofday` function returns the time of day in seconds since January 1, 1970.

25 Shell functions

25.1 echo

The `echo` function prints a string.

```
$(echo <args>) echo <args>
```

25.2 jobs

The `jobs` function prints a list of jobs.

```
jobs
```

25.3 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
```

25.4 bg

The `bg` function places a job in the background.

```
bg <pid...>
```

25.5 fg

The `fg` function brings a job to the foreground.

```
fg <pid...>
```

25.6 stop

The `stop` function suspends a job.

```
stop <pid...>
```

25.7 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...>
```

25.8 kill

The `kill` function signals a job.

```
kill [signal] <pid...>
```

25.9 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`.

26 Pervasives

`Pervasives` defines the objects that are defined in all programs. The following objects are defined.

26.1 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 `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 an object.

```
FieldNames(o) =
   names =
   o.foreach(v, x)
      names += $(v)
      export
   return $(names)
```

26.2 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.

- `$(o.mem <key>)`: returns `true` iff the `<key>` is defined in the map.
- `$(o.add <key>, <value>)`: adds the field to the map, returning a new map.
- `$(o.find <key>)`: fetches the field from the map.
- `$(o.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.

26.3 Number

Parent objects: `Object`.

The `Number` object is the parent object for integers and floating-point numbers.

26.4 Int

Parent objects: `Number`.

The `Int` object represents integer values.

26.5 Float

Parent objects: `Number`.

The `Float` object represents floating-point numbers.

26.6 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)
  ```

26.7 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.

26.8 String

Parent objects: `Array`.

26.9 Fun

Parent objects: `Object`.

The `Fun` object provides the following methods.

- `$(f.arity)`: the arity if the function.

26.10 Rule

Parent objects: `Object`.

The `Rule` object represents a build rule. It does not currently have any methods.

26.11 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.

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.

26.12 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.

26.13 File

Parent objects: `Node`.

The file object represents the name of a file.

26.14 Dir

Parent objects: `Node`.

The `Dir` object represents the name of a directory.

26.15 Channel

Parent objects: `Object`.

A `Channel` is a generic IO channel. It provides the following methods.

- `$(o.close)`: close the channel.

26.16 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.

26.17 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.

- `$(OutChannel.append <file>)`: opens a new output channel, appending to the file.
- `$(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.

26.18 Location

Parent objects: `Location`.

The `Location` object represents a location in a file.

26.19 Position

Parent objects: `Position`.

The `Position` object represents a stack trace.

26.20 Exception

Parent objects: `Object`.

The `Exception` object is used as the base object for exceptions. It has no fields.

26.21 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.

26.22 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.

- `history`

  Print the current command-line history.

- 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.

  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 \emph{expression}
\verb+[+ \emph{expression} +]+
\verb+[ --help+
\verb+[ --version+
```

See the documentation for the `test` function.

– `find`

```
find \emph{expression}
```

See the documentation for the `find` function.

27 Build functions

27.1 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
```

27.2 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`.

27.3 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.

27.4 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.

28 The OMakeroot file

The standard `OMakeroot` file defines the functions are rules for building standard projects.

28.1 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.

VERBOSE Whether certain commands should be verbose (`false` by default).

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.

28.2 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`).

29 Building C programs

omake provides extensive support for building C programs.

29.1 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).

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`).

AR The name of the program to create static libraries (defaults to `ar cq` on `Unix`, and `lib` on `Win32`).

AROUT The option string that specifies the output file for `AR`.

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).

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`).

29.2 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.

29.3 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)
```

29.4 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)
```

29.5 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)
```

29.6 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`).

29.7 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$(EXT_LIB)
   CProgram(foo, bar baz)
```

29.8 CProgramCopy

The `CProgramCopy` function copies a file to an install location.

```
CProgramCopy(<tag>, <dir>, <program>)
```

```
CProgramCopy(install, /usr/bin, foo)
```

29.9 CProgramInstall

The `CProgramInstall` function specifies a program to build, and a location to install, simultaneously.

```
CProgramInstall(<tag>, <dir>, <name>, <files>)
```

```
section
   LIBS = libfoo$(EXT_LIB)
   CProgramInstall(install, /usr/bin, foo, bar baz)
```

29.10 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)`.

30 Building OCaml programs

30.1 Variables for OCaml programs

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`).

OCAMLDEP The OCaml dependency analyzer (default `ocamldep`).

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.

30.2 OCaml command flags

The following variables specify additional options to be passed to the OCaml tools.

OCAMLDEPFLAGS Flags to pass to `OCAMLDEP`.

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`).

OCAMLINCLUDES Include path (default `.`).

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.

30.3 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.

30.4 OCamlGeneratedFiles, LocalOCamlGeneratedFiles

```
OCamlGeneratedFiles(files)
LocalOCamlGeneratedFiles(files)
```

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.

30.5 OCamlLibrary

The OCamlLibrary function builds an OCaml library.

OCamlLibrary(<libname>, <files>)

The <libname> and <files> are listed *without* suffixes.

Additional variables used by the function:

ABORT_ON_DEPENDENCY_ERRORS The linker requires that the files to be listed in dependency order. If this variable is true, the order of the files is determined by the command line, but *omake* will abort with an error message if the order is illegal. Otherwise, the files are sorted automatically.

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)
```

30.6 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.

30.7 OCamlLibraryInstall

The `OCamlLibraryInstall` function builds a library and copies it to an install location in one step.

`OCamlLibraryInstall(<tag>, <libdir>, <libname>, <files>)`

30.8 OCamlProgram

The `OCamlProgram` function builds an OCaml program. It returns the array with all the targets for which it have 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.

30.9 OCamlProgramCopy

The `OCamlProgramCopy` function copies an OCaml program to an install location.

`OCamlProgramCopy(<tag>, <bindir>, <name>)`

Additional variables used:

NATIVE_ENABLED If `NATIVE_ENABLED` is set, the native-code executable is copied; otherwise the byte-code executable is copied.

30.10 OCamlProgramInstall

The `OCamlProgramInstall` function builds a programs and copies it to an install location in one step.

`OCamlProgramInstall(<tag>, <bindir>, <name>, <files>)`

31 Building LaTeX programs

31.1 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 empty).

USEPDFLATEX Flag indicating whether to use pdflatex instead of dvipdfm to generate the `.pdf` document (default `false`).

31.2 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.

31.3 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.

31.4 LaTeXDocumentCopy

The `LaTeXDocumentCopy` copies the document to an install location.

```
LaTeXDocumentCopy(<tag>, <libdir>, <installname>, <docname>)
```

This function copies just the `.pdf` and `.ps` files.

31.5 LaTeXDocumentInstall

The `LaTeXDocumentInstall` builds a document and copies it to an install location in one step.

```
LaTeXDocumentInstall(<tag>, <libdir>, <installname>, <docname>, <files>)
```

32 Examining the dependency graph

32.1 dependencies, dependencies-all

```
$(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))
```

32.2 target

```
$(target targets) : Rule 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.

32.3 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 two-element array with an option name, and the option 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.

```
rule(false, target, pattern, sources,
     [[:name1:, option1], [:name2:, option2]]
     [expr1; expr2])
```

33 References

33.1 See Also

omake(1), omake-quickstart(1), omake-options(1), omake-root(1), omake-language(1), omake-shell(1), omake-rules(1), omake-base(1), omake-system(1), omake-pervasives(1), osh(1), *make*(1)

33.2 Version

Version: 0.9.6.9 of April 11, 2006.

33.3 License and Copyright

33.4 Author

Jason Hickey *et. al.*
Caltech 256-80
Pasadena, CA 91125, USA
Email: `omake-devel@metaprl.org`
WWW: `http://www.cs.caltech.edu/~jyh`