The OMake build system

Jason Hickey

4th January 2005
Version 0.9.4

Abstract

omake is a build system designed for building large projects, using accurate dependency analysis based on MD5 digests. *omake* uses a syntax similar to *make*(1), with many additional features.

1 Synopsis

omake [**-k**] [**-j***count*] [**-n**] [**-s**] [**-S**] [**-p**] [–**progress**] [–**no-progress**] [–**print-status**] [–**no-print-status**] [**-w**] [–**no-print-exit**] [–**print-exit**] [**-t**] [**-u**] [**-R**] [–**print-dependencies**] [–**show-dependencies** *target*] [–**install**] [–**install-all**] [–**install-force**] [*filename...*] [**var-definition...**]

2 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.

2.1 Automatic dependency analysis

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

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

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

2.2 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.

3 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. 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.

-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-status** Print status lines for remote execution servers.

–**no-print-status** Do not print status lines for remote execution servers.

–**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 subdirectry 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 forcably 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"`.

4 Structure of this document

This manual is divided into several sections.

Introduction Gives an overview of the *omake* language.

Expressions and Values Describes the the *omake* language in more detail, including a description of objects, expressions, and values.

The standard library Describes the standard library that comes with *omake*.

The OMakeroot file Describes the system `OMakeroot` file, which contains functions to build C, OCaml, and LaTeX programs.

Examples This section gives several examples in tutorial form.

5 Introduction

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.

5.1 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`.

5.2 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
```

5.3 Arrays

Arrays can be defined by appending the `[]` sequence to the variable name and defining initial values for the elements. The following code sequence prints `c d e`.

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

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

5.4 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"""
```

5.5 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 is used to return a value from the function. 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 "I says: Hello world".

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

Printer(I)
```

5.6 Rules

Rules specify commands to solve a file dependency. 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 is newer, 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.

5.7 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. The variables `$@` and `$*` are special. The following special variables may be used in rules.

- `$*`: the target name, without a suffix.
- `$@`: the target name.
- `$^`: a list of the sources, with duplicates removed.
- `$+`: a list of the sources, duplicates are not removed.
- `$<`: the first source

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.

5.8 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
```

5.9 section eval

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 followng rule uses the *omake* IO functions to produce the target `hello.c`.

```
hello.c:
    section eval
        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.

5.10 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.

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

5.11 :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 [ -e .flag ]; then $(CP) a.c $@; fi
```

5.12 :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 simulataneously running programs that have overlapping side-effects.

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

5.13 Special targets

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

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
```

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
```

5.14 Comments

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

5.15 File inclusion

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

```
include files.omake
```

5.16 Scoping

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
   .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.

5.17 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.

```
if $(equal $(OSTYPE), Win32)
   # Win32 uses the copy and del programs
   CP = copy
```

```
   RM = del
   export
elseif $(equal $(OSTYPE), Unix)
   # Other architectures use cp and rm
   CP = cp
   RM = rm
   export
else
   # Abort on other architectures
   eprintln($(OSTYPE) is not recognized)
   exit(1)
```

5.18 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)
```

6 Examples

6.1 Quick start

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 before they are used.
- Subdirectories are incorporated into a project using the `.SUBDIRS:` target.

6.2 Getting started

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

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

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

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

```
#include <stdio.h>

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

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

6.3 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)
```

6.4 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 subsdirectories.

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

```
INCLUDES += -I.. -I../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 subsdirectory 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, 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 inner blocks to outer blocks.

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

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

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

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

# libfoo uses the default configuration
.SUBDIRS: foo

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

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

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

   .SUBDIRS: win32
   export

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

.DEFAULT: hello$(EXE)
```

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

```
INCLUDES += -I.. -I../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 = -I$(ROOT) -I$(dir foo) -I$(dir bar) -I$(dir win32)
```

In the `foo` directory, the `INCLUDES` variable will evaluate to the string `-I.. -I. -I../bar -I../win32`. In the `bar` directory, it would be `-I.. -I../foo -I. -I../win32`. In the root directory it would be `-I. -Ifoo -Ibar -Iwin32`.

6.5 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`.

6.6 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 += -I$(dir foo) -I$(dir bar)
OCAMLINCLUDES += -I $(dir foo) -I $(dir bar)
.SUBDIRS: foo bar

# C libraries
LIBS = foo/libfoo

# OCaml libraries
OCAML_LIBS = bar/libbar

# Also use the Unix module
OCAML_OTHER_LIBS = unix

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

```
.DEFAULT: hello.run
```

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 libarary.

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

7 Expressions and values

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

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

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

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

7.1 Dynamic scoping

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.2 Functional evaluation

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

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

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

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

f = $(incby 5)

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

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

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

The `export` directive can be used to propagate all or part of an inner scope back to its parent. The `export` directive should be the last statement in a block.

If used without arguments, the entire scope is propagated back to the parent; otherwise the arguments should be the names of variables to propagate. The most common usage is to export the definitions in a conditional block. In the following example, the variable `B` is bound to 2 after the conditional. The `A` variable is not redefined.

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

7.3 Eager evaluation

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

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

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

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

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

7.4 Strings, Sequences, and Arrays

omake represents most input values as sequences, where a sequence is a list of values separated by whitespace. Sequences are often nested, and may contain arbitrary values. The elements of a sequence are the values separated by whitespace.

```
osh> A = 1 2
- : "1 2" : Sequence
osh> B = 3 $(A) 5
- : <sequence "3 " : Sequence "1 2" : Sequence " 5" : Sequence> : Sequence
osh> return $(length $(B))
- : 4 : Int
osh> return $(nth 3, $(B))
- : "5" : String
osh> f(x) =
        return $(x)
osh> C = x $(f) y
- : <sequence "x " : Sequence <fun 1> " y" : Sequence> : Sequence
osh> println($(C))
x  y
```

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

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

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

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

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

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

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

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

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

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

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

```
        c:\Program Files\omake\second file

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

7.5 Objects

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

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

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

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

   # Fields
   X = value
   Y = value

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

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

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

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

7.6 Field and method calls

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

```
Point. =
   class Point

   # Default value
   x = $(int 0)

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

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

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

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

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

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

7.7 Method override

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

```
Point2. =
   extends $(Point)

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

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

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

7.8 Super calls

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

```
Point2. =
   extends $(Point)

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

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

8 Special targets

8.1 .PHONY

The `.PHONY` target specifies that a target name does not actually correspond to a file. Phony targets are often used to specify extra dependencies. For example, the `all` target is frequently used to specify a set of files that should be constructed by default, but the name `all` does not correspond to a file.

```
.PHONY: all
all: hello.o
```

`.PHONY` targets have the usual scoping properties. A `.PHONY` target *must* be defined before it is used. The following sequence is incorrect.

```
# This is the WRONG way to use .PHONY.
# .PHONY targets must be declared before being used.
all: hello.o
.PHONY: all
```

Scoping extends to subdirectories. In the following, the target `all` is `.PHONY` in the `foo` subdirectory. Once again, the `.PHONY` declaration must occur *before* the `.SUBDIR` declaration. It is an error to place it afterwards.

```
.PHONY: all
all: hello.o
.SUBDIR: foo
```

8.2 .DEFAULT

omake does not build the target specified by the first rule in an `OMakefile` (since there are several `OMakefile`s). The `.DEFAULT` phony target specifies what targets should be built by default.

```
.DEFAULT: hello.o
```

8.3 .SCANNER

The `.SCANNER` target specifies a command to produce dependency information for a file. The command should produce output that is compatible with *omake*. For example, the following rule uses *gcc*(1) to generate dependencies for `.c` files.

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

8.4 .SUBDIRS

The `.SUBDIRS` target is used to specify subdirectories containing *omake* projects. Each of the subdirectories should have an `OMakefile`. For example, the following line specifies a project with three subdirectories: `foo`, `bar`, and `baz`.

```
.SUBDIRS: foo bar baz
```

The current environment is inherited by the subdirectories. This can be useful if different options are to be used in different subdirectories. For example, the following lines specify that `foo` should be built in debug mode, while `bar` and `baz` should be optimized.

```
CFLAGS = -g
.SUBDIRS: foo

CFLAGS = -O2
.SUBDIRS: bar baz
```

8.5 .INCLUDE

The `.INCLUDE` target is like the `include` form, but it provides for commands to build the included file if it doesn't exist.

```
.INCLUDE: <target>: <dependencies>
    <commands>
```

If the `<target>` does not exist, or it is older than any of the files in the `<dependencies>`, then the `<commands>` are executed to build the `<target>`. Once the `<target>` is up-to-date, it is included as a source file. The `.INCLUDE`d file must use the `OMakefile` syntax.

9 Builtin variables

OSTYPE Set to the machine architecture *omake* is running on. Possible values are `Win32` and `Unix`.

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

NODENAME The hostname of the current machine.

VERSION The operating system release.

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

HOST Same as `NODENAME`.

USER The login name of the user executing the process.

HOME The home directory of the user executing the process.

10 Boolean functions and control flow

10.1 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`.

10.2 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`.

10.3 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)))
```

10.4 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))
```

10.5 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
```

10.6 switch, match

The `switch` and `match` functions performs a 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.

```
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)
```

10.7 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.

10.8 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.

10.9 return

```
return value
```

The `return` function is normally used to return a result from a function.

The return statement does *not* abort function execution; it should be the last statement in a function body.

```
F(a, b, c) =
   return $(a)/$(b)/$(c).a
```

10.10 exit

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

The `exit` function terminates *omake* abnormally.

`$(exit <code>)`

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

11 Operations on variables

11.1 export

```
export
export names
   names : Sequence
```

The `export` function is used to export the environment defined in an inner scope to an outer scope.

Normally, variables and functions defined in inner scopes (greater indentation levels) are not visible in parent scopes (smaller indentation levels). The export function is used to pass the inner definitions to the outer scope. If the

`export` function is used with an argument, only the variables mentioned in the argument are exported.

For example, the following expression defines the `CC` and `CFLAGS` variables depending on the system type.

```
switch $(OSTYPE)
case Win32
   CC = cl
   CFLAGS += /DWIN32
   export
case Unix
   X = abc  # this variable is not exported
   CC = gcc
   CFLAGS += -g -O2
   export CC CFLAGS
default
   raise Unknown system architecture $(OSTYPE)
```

11.2 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
```

11.3 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
```

11.4 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))
```

11.5 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.

12 Arrays and sequences

12.1 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.

12.2 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))
```

12.3 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))
```

12.4 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.

12.5 nth

```
$(nth sequence) : value
   sequence : Sequence
raises RuntimeException
```

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

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

12.6 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`.

12.7 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 trated as a unit; whitespace is significant.

12.8 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"`.

12.9 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`.

12.10 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`.

12.11 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`.

12.12 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"`.

12.13 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`.

12.14 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"`.

12.15 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"`.

12.16 add-wrapper

```
$(add-wrapper prefix, suffix, sequence) : Array
   prefix : String
   suffix : String
   sequence : 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.

12.17 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`.

12.18 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`.

12.19 intersection

```
$(intersection sequence1, sequence2) : Array
   sequence1 : Sequence
   sequence2 : Sequence
```

The `intersection` function takes two arguments, treats them as sets, 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`.

12.20 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`.

12.21 filter-out

```
$(filter-out pattern, sequence) : Array
   pattern : String
   sequence : Sequence
```

The `filter-out` function removes elements from a sequence. The pattern may contain one occurrence of the wildcard `%` character.

For example `$(filter-out %.c, a.c x.o y.o "hello world".c)` evaluates to `x.o y.o`.

12.22 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`.

12.23 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`.

12.24 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`.

12.25 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`.

13 Build functions

13.1 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
```

13.2 OMakeVersion

```
OMakeVersion(version1)
OMakeVersion(version1, version2)
   version1 : String
   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`.

13.3 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.

13.4 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
```

13.5 shell

```
$(shell command) : Array
   command : Sequence
$(shella command) : Array
   command : Sequence
```

The `shell` function evaluates a command using the command shell, and returns the whitespace-spearated words of the standard output as the result.

The `shella` function acts similarly, but it returns the lines as separate items in the array.

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).

13.6 rule

```
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])
```

14 Arithmetic

14.1 int

The `int` function can be used to create integers. It returns `Number` objects.
`$(int <s>)`.

14.2 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.

14.3 Basic arithmetic

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.

14.4 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 = $(fun X, Y, return($(addsuffix $(Y), $(X))))

F =
   fun(X, Y)
       return($(addsuffix $(Y), $(X)))
```

14.5 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)
```

14.6 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))
```

15 File operations

15.1 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 provdes 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)
```

15.2 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
```

15.3 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.

15.4 exists-in-path

```
$(exists-in-path files) : String
   files : String Sequence
```

The exists-in-path function tests whether all executables are present in the current search path.

15.5 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 without 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`.

15.6 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 /`.

15.7 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`.

15.8 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.

15.9 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`.

15.10 file-exists

```
$(file-exists files) : String
   files : File Sequence
```

The `file-exists` function checks whether the files listed exist.

15.11 target-exists

```
$(target-exists files) : String
   files : File Sequence
```

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.

15.12 filter-exists, filter-targets

```
$(filter-exists files) : File Sequence
   files : File Sequence
$(target-exists files) : File Sequence
   files : File Sequence
```

The `filter-exists` and `filter-targets` functions remove files from a list of files if they do not exist (`filter-exists`) or they do not exist and can't be built according to the current project (`filter-targets`).

15.13 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 *emph* 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.

15.14 sort rule

It is possible to firther 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`.

15.15 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>)
```

15.16 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.

c If a glob expansion does not match, return the pattern instead of aborting.

. 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.

In addition, the following variables may be defined that affect the behavior of `glob`.

GLOB_IGNORE A list of shell patterns for filenames that `glob` should ignore.

GLOB_ALLOW A list of shell pattern. If a file does not match a pattern in `GLOB_ALLOW`, it is ignored.

The returned files are sorted by name.

15.17 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.

15.18 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.

15.19 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.

15.20 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.

15.21 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.

15.22 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.

15.23 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.

– Interprete the remaining arguments literally.

15.24 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.

15.25 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.

15.26 readlink

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

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

15.27 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.

15.28 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.

15.29 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.

16 IO functions

16.1 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.

16.2 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.

16.3 close

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

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

16.4 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.

16.5 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.

16.6 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.

16.7 rewind

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

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

16.8 tell

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

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

16.9 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.

16.10 dup

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

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

16.11 dup2

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

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

16.12 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`.

16.13 set-nonblock

```
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.

16.14 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.

16.15 mkfifo

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

The `mkfifo` function creates a named pipe.

16.16 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.

16.17 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.

16.18 InetAddr

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

addr `String`: the Internet address.

port `Int`: the port number.

16.19 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 prefered socket domain.

addrs `InetAddr Array`: an array of Internet addresses belonging to the host.

16.20 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.

16.21 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.

16.22 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.

16.23 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.

16.24 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.

16.25 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.

16.26 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.

16.27 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.

16.28 accept

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

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

16.29 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.

16.30 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`.

16.31 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.

16.32 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.

16.33 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.

16.34 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 `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.

17 Higher-level IO functions

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 beginnning 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.

17.1 cat

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

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

17.2 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`.

17.3 awk

```
awk([input [, output]])
case pattern1:
   body1
case pattern2:
   body2
...
default:
   bodyd
```

The `awk` function provides input processing similar to *awk*(1), but more limited. The function takes up to two arguments. If called with no arguments, the input is from `stdin`, and output is to `stdout`. If one argument is specified, it specifies a `InChannel`, or the name of a file for input. If two arguments are specified, the second argument should be an `OutChannel` or the name of a file for output.

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, Awk.out)
   case $"^\\end\{\([:alpha:]+\)\}"
      if $(mem $1, $(names))
         print = false
         export
      export
   default
      if $(print)
         println($0)
   case $"^\\begin\{\([:alpha:]+\)\}"
      print = $(mem $1, $(names))
      export
```

17.4 fsubst

```
fsubst([input [, output]])
case pattern1 [options]
   body1
case pattern2 [options]
   body2
...
default
   bodyd
```

The `fsubst` function provides a *sed*(1)-like substitution function. The function takes up to two arguments. If called with no arguments, the input is from `stdin`, and output is to `stdout`. If one argument is specified, it specifies a `InChannel`, or the name of a file for input. If two arguments are specified, the second argument should be an `OutChannel` or the name of a file for output.

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.

For example, the following program replaces all occurrences of an expression `word.` with its capitalized form.

```
fsubst(Subst.in, Subst.out)
case $"\<\([[:alnum:]]+\)\." g
   return $(capitalize $1).
```

17.5 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)

   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 $* ))

   other: .
      eprintln(Illegal character: $* )
      lex()

   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.

The first clause is responsible for ignoring white space. If whitespace is found, it is ignored, and the lexer is called recursively.

The second 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.

17.6 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.

17.7 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

term: $"[*][)]"
   if $(not $(eq $(level), 0))
      level = $(sub $(level), 1)
      lex()

next: $"[(][*]"
   level = $(add $(level), 1)
   lex()

other: .
   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.

17.8 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.

17.9 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.

17.10 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.

17.11 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.

17.12 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.

17.13 gettimeofday

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

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

17.14 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 `a.c b.c c.c`.

```
X =
   foreach(x, a b c)
      $(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
```

18 Shell functions

18.1 echo

The `echo` function prints a string.

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

18.2 jobs

The `jobs` function prints a list of jobs.

`jobs`

18.3 cd

The `cd` function changes the current directory.

`cd <dir>`.

18.4 bg

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

`bg <pid...>`

18.5 fg

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

`fg <pid...>`

18.6 stop

The `stop` function suspends a job.

`stop <pid...>`

18.7 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...>
```

18.8 kill

The `kill` function signals a job.

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

19 The Pervasives Object

The `Pervasives` object is the toplevel object that is opened to all programs. In addition to the builtin functions that have been mentioned, it includes the following objects.

19.1 Object

Parent objects: none.

The `Object` object is a generic 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)
```

19.2 Number

Parent objects: `Object`.

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

19.3 Int

Parent objects: `Number`.

The `Int` object represents integer values.

19.4 Int

Parent objects: `Number`.

The `Float` object represents floating-point numbers.

19.5 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 two arguments; the first is a field name, the second is the value of that field. 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(<var1>, <var2>)
   <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)
```

19.6 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.

19.7 String

Parent objects: `Array`.

The `String` object provides the following methods.

- `$(s.explode)`: returns an array of the characters in `s`.

19.8 Map

Parent objects: `Sequence`.

A `Map` is a map from strings to values. It provides the following methods.

- `$(o.mem <var>)`: returns `true` iff the `<var>` is defined in the map.

- `$(o.add <var>, <value>)`: adds the field to the map, returning a new map.
- `$(o.find <var>)`: 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)
```

19.9 Fun

Parent objects: `Object`.

The `Fun` object provides the following methods.

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

19.10 Rule

Parent objects: `Object`.

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

19.11 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.

19.12 File

Parent objects: `Node`.

The file object represents the name of a file.

19.13 Dir

Parent objects: `Node`.

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

19.14 Channel

Parent objects: `Object`.

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

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

19.15 InChannel

Parent objects: `Channel`.

A `InChannel` is an input channel. The variable `stdin` is the standard input channel.

It provides the following methods.

- `$(InChannel.open <file>)`: open a new input channel.

19.16 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.

- `$(OutChannel.open <file>)`: open a new output channel.
- `$(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.

19.17 Location

Parent objects: `Location`.

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

19.18 Position

Parent objects: `Position`.

The `Position` object represents a stack trace.

19.19 Exception

Parent objects: `Object`.

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

19.20 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.

19.21 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.

- 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`, `rm`, `mkdir`, `chmod`, `ln`. 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 forcably, 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 Forcably 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.

20 The OMakeroot file

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

20.1 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.

20.2 System variables

CP The copy command (`cp` on `Unix`, `copy` on `Win32`).

RM The removal command (`rm` on `Unix`, `del /f` on `Win32`).

MKDIR The command to create a directory (`mkdir`).

CHMOD The command to change permissions on a file (`chmod` on Unix, disabled on `Win32`).

INSTALL The command to install a program (`install` on `Unix`, `copy` on `Win32`).

PATHSEP The normal path separator (`:` on `Unix`, `;` on `Win32`).

DIRSEP The normal directory separator (`/` on `Unix`, `\` on `Win32`).

21 Building C programs

omake provides extensive support for building C programs.

21.1 C configuration variables

The following variables can be redefined in your project.

CC The name of the C compiler (defaults to `cc` on `Unix`, and `cl` on `Win32`).

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`).

INCLUDES Additional directories that specify the search path to the C compiler (default is `.`). The directories are passed to the C compiler 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 assember (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`.

RANLIB The name of the command to postprocess the output of `AR` (defaults to `ranlib` on `Unix`, empty on `Win32`).

LD The name of the linker (defaults to `ld` on `Unix`, and `cl` on `Win32`).

LDFLAGS Options to pass to the linker (default is empty).

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`).

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`).

21.2 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.

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)
```

21.3 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).

For example, the following code copies the library `libfoo.a` to the `/usr/lib` directory.

```
.PHONY: install

StaticCLibraryCopy(install, /usr/lib, libfoo)
```

21.4 StaticCLibraryInstall

The StaticCLibraryInstall function builds a library, and sets the install location in one step.

```
StaticCLibraryInstall(<tag>, <dir>, <libname>, <files>)
```

```
StaticCLibraryInstall(install, /usr/lib, libfoo, a b c)
```

21.5 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).

21.6 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.

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)
```

21.7 CProgramCopy

The CProgramCopy function copies a file to an install location.

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

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

21.8 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)
```

22 Building OCaml programs

22.1 Variables for OCaml programs

The following variables can be redefined in your project.

USE_OCAMLFIND Whether to use the ocamlfind utility (default `true\` if `ocamlfind` exists, `false\` otherwise).

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`).

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).

OCAMLPACKS Package names to pass to `ocamlfind\verb` (`OCAMLFIND` must be set).

BYTE_ENABLED Flag indicating whether to use the bytecode compiler (default `false`).

NATIVE_ENABLED Flag indicating whether to use the native-code compiler (default `true`). Both `BYTE_ENABLED` and `NATIVE_ENABLED` can be set to true; at least one should be set to true.

22.2 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.

22.3 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.

22.4 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 dependecy 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.

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)
```

22.5 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.

22.6 OCamlLibraryInstall

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

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

22.7 OCamlProgram

The `OCamlProgram` function builds an OCaml program.

`OCamlProgram(<name>, <files>)`

Additonal 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.

22.8 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.

22.9 OCamlProgramInstall

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

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

23 Building LaTeX programs

23.1 Configuration variables

The following variables can be modifed in your project.

LATEX The LaTeX command (default `latex -interaction=errorstopmode`).

BIBTEX The BibTeX command (default `bibtex`).

MAKEINDEX The command to build an index (default `makeindex`).

DVIPS The `.dvi` to PostScript converter (default `dvips`).

DVIPDFM The `.dvi` to `.pdf` converter (default `dvipdfm`).

23.2 LaTeXDocument

The `LaTeXDocument` produces a LaTeX document.

`LaTeXDocument(<name>, <texfiles>)`

The document `<name>` and `<texfiles>` are listed without suffixes.

Additional variables used:

TEXINPUTS The LaTeX search path.

TEXDEPS Additional files this document depends on.

23.3 LaTeXDocumentCopy

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

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

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

23.4 LaTeXDocumentInstall

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

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

24 See Also

make(1), *osh*(1)

25 Version

Version: 0.9.4 of 4th January 2005.

26 License and Copyright

27 Author

Jason Hickey
Caltech 256-80
Pasadena, CA 91125, USA
Email: `jyh@cs.caltech.edu`
WWW: `http://www.cs.caltech.edu/~jyh`