Make conditionals

David A. Wheeler

2013-11-29

Below is a draft POSIX description, draft POSIX desired action, and background information, on adding if-then-else conditionals to the POSIX specification of “make”.

Draft Description

Almost all “make” implementations include some kind of if-then-else conditional. Most implementations support at least two types of conditionals: an “is-defined” conditional (e.g., “ifdef” or “.ifdef” or “#ifdef”) and a more general conditional (e.g, “if” or “ifeq” or “#if”). Examples of make implementations that provide conditionals include GNU make, makepp, FreeBSD/OpenBSD/NetBSD make (pmake), Microsoft nmake, AT&T/Alcatel-Lucent nmake, imake, and fastmake. Makefile generators that use make syntax (with extensions), like automake, also support make conditionals.

Conditionals are important for creating exchangeable makefiles, because they enable creating makefiles that automatically build on a variety of different systems (by automatically varying various settings). Unsurprisingly, conditionals are widely used in makefiles.

Yet the POSIX make specification fails to provide a standard mechanism for if-then-else conditionals, so conditionals cannot be used portably in makefiles. Tools like automake work around this by generating portable makefiles through workarounds, but workarounds should not be necessary, especially since most make implementations already support if-then-else conditionals.

The general functionality of if-then-else conditionals is similar between different makes, but their syntax and semantics vary widely. Thus, we must pick an approach that is not currently implemented by all. That’s okay, as long as it’s a syntax that could be added by the others (so it’s best to avoid notations that would interfere with others’ notation). I think it comes down to a decision between GNU make, the *BSD makes, AT&T nmake, or a new notation based on them:

1. GNU make and makepp have compatible syntax (makepp copies GNU make). They use ifdef|ifndef for is-defined, and ifeq|ifneq as a more general comparator. GNU make is very widely used (it’s the usual “make” in Linux and MacOS, and is widely available elsewhere), so this notation is widely used. The ifdef|ifndef notation is quite clean. A disadvantage is that the ifeq|ifneq notation is clunky, limiting, and non-intuitive. In particular, it doesn’t support clean complex expressions (with && and ||) in a simple way,

and it's not visually obvious that "equal-to" is intended (there's no "=" or "==").
2. The *BSDs use .ifdef|.ifndef for is-defined, and .if for a more general comparitor. (Fastmake uses a similar (but not identical) ".IF"/".ELSE"/".ENDIF" syntax for its conditional.) The ".ifdef" syntax allow || and &&, which is nice, but since ANY setting defines a value, it's hard to undefine a value on the command line, which I believe is a serious weakness in its semantics. The *BSDs ".if" syntax is cleaner, but semantically it "automagically" determines if == is a numeric or string comparison (e.g., and interpret "0x" specially), which I think is dangerous.
3. AT&T/Alcatel-Lucent nmake use "if". The conditional syntax has slightly unusual semantics (initial single-quote means interpret as string, initial double-quote interpret as pattern, no quote interpret as number). There is no separate "ifdef" syntax, but that could be added. Paul Smith (GNU make) has sent me email that he really does not like the "if" syntax where the same operator syntax has different semantic meanings depending on the quote (or not) attached to the first operator.
4. POSIX could make its own syntax, building on existing approaches, just like printf(1), building on existing experience.

Here are some other alternatives:

1. Microsoft nmake begins with "!IF". I don't see evidence that they're striving for POSIX compliance, and "!" isn't especially reserved, so I'll ignore it.
2. Opus software uses "%if…%elif…%else…%endif", but it's unmaintained as far as I can tell, and "%" interferes with pattern matching.
3. Imake is built on cpp, so it uses cpp syntax (e.g., #ifdef).

Given the concerns listed above, I think the best approach is to add new syntax (#4 in the first list), building on the capabilities of GNU make, BSD make (pmake), and nmake. The following approach uses GNU make's "ifdef" for variable definitions (which are easier to disable on the command line or via include files), but I've added BSD make's .ifdef capabilities which support ||, &&, and !. Both GNU make "ifeq" and BSD make's ".if" have issues, so I propose a new syntax "iftrue" that is strongly influenced by the BSD make syntax. I did not use "if" because that would conflict with the syntax of other tools (e.g., automake and nmake). Existing simple GNU make files that only use "ifdef" would be portable, and existing makefiles that use nonstandard conditionals can continue to use them (though they'd continue to be nonstandard). This proposal does not add macro functions, but is completely consistent with them; see bug report 512. The description below does not create a new syntax for escaping values, but existing substitutions could be used (e.g., x$(space)y could be used for "x y").

Earlier drafts of this proposal required commands to begin on the left edge, since indented make conditionals could be visually confused with an "if" in a rule command. However, this could easily lead to confusion when make conditionals are nested.

NOTE: It's also helpful to support some related capabilities, but these are out-of-scope of this proposal (though they would work well together):

- Macro functions such as $(if CONDITION, THEN [,ELSE]). See: http://austingroupbugs.net/view.php?id=512
- Recursive variable indirection. See: http://austingroupbugs.net/view.php?id=336

I've collected more information here: https://docs.google.com/document/d/1oUR7iMnaNzkeT3TTOS-Gwul6_V3TE8caIDAd1FwPyNc/edit?usp=sharing

Draft Desired Action - Alternate "iftrue" syntax version

In line 97115, change "include lines," to "include lines, conditional statements,".

Before line 97166 (right after the "include lines") section, add a new section, "Conditional statements" with the following content:

======

If the word "ifdef", "ifndef", "iftrue", "else", or "endif" appears at the beginning of a line after 0 or more <blank> characters, and if followed by or one or more <blank> characters or the end of line, the line is part of a conditional statement. A conditional statement is of the form:

```
<conditional_statement> ::= <begin_condition> <content>
                                    (<continued_condition> <content>)*
                                     (<else_condition> <content>)? <endif_line>
<begin_condition> ::= ( "ifdef" <space>+ <ifdef_condition>
                              | "ifndef" <space>+ <ifdef_condition>
                              | "iftrue" <space>+ <if_condition> ) <eol>
<continued_condition> ::= "else" <space>+ <begin_condition>
<else_condition> ::= "else" <eol>
<content> ::= <line>*
<endif_line> ::= "endif" <eol>
<eol> ::= <return>? <newline>
<ifdef_condition> ::= "(" <ifdef_condition> ")" |
                              "!"? <value> { "&&" <ifdef_condition> | "||" <ifdef_condition> }
<if_condition> ::=  <expression> { "&&" <if_condition> | "||" <if_condition> }
<expression> ::= "(" <expression> ")" | "!" <space>+ <expression> ||
                         <value> [ <op> <value> ]
<value> ::= ( <macro_expansion> | <other_character> )+
<op> ::= "==" || "!=" | "-lt" | "-le" | "-gt" | "-ge" | "-eq" | "-ne"
```

The precedence from highest to lowest is parentheses, "!" (not), "&&" (logical and) and" (logical or). When executed both "&&" and "||" short-circuit, left-to-right. An "other_character" is a

character other than “$”, whitespace, or a control character (note that newline is a control character). A <macro_expansion> begins with “$” and is described in the “Macros” section below, e.g., “${HOME}” is a macro expansion. Outside of macro expansions, <space> characters in <ifdef_condition> and <if_condition> are ignored except that they separate tokens.

An “op” is a string or numeric operator. The string operators are “==” (case-sensitive string equality) and “!=” (case-sensitive string inequality). The numeric operators are “-lt” (numeric less-than), “-le” (numeric less-than or equal-to), “-gt” (numeric greater-than), “-ge” (numeric greater-than or equal-to), “-eq” (numeric equality), and “-ne” (numeric inequality). An operand to a numeric operators that begins with “0x” is considered a number in base 16.

If an expression consists of just a value without the optional “op” it is considered true if it evaluates to at least one character, and false otherwise. In ifdef_condition, a <value> is evaluated and considered the name of a macro to determine if it is defined. A macro is considered defined if ifdef_condition if it has a nonempty value before evaluation (use “::=” to precalculate a macro).

Conditional statements are evaluated as the makefile is read in, in the order specified. If the relevant condition evaluates as true, then the content is read and used; otherwise its content is ignored.

Background

There are many make and make-like implementations; here is some relevant information, first on general information about make, followed by summaries about various implementations.

General make information

Pages that discuss makes in general include:
http://en.wikipedia.org/wiki/Make_%28software%29
http://freecode.com/articles/make-alternatives.

GNU make

GNU make is an extremely popular implementation of make. The GNU make if-then-else conditionals are described at:
http://www.gnu.org/software/make/manual/html_node/Conditionals.html#Conditionals.

For example:

```
ifeq ($(CC),gcc)
$(CC) -o foo $(objects) $(libs_for_gcc)
else
$(CC) -o foo $(objects) $(normal_libs)
endif
```

In general they have the form:
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
ifneq (arg1, arg2)
ifneq 'arg1' 'arg2'
ifneq "arg1" "arg2"
ifneq "arg1" 'arg2'
ifneq 'arg1' "arg2"
ifdef variable-name
ifndef variable-name

For purposes of ifdef and ifndef, a macro is defined if it has a non-empty value BEFORE expanding the macro. Thus after:

```
bar =
foo = $(bar)
```

an "ifdef bar" is false, but "ifdef foo" is true.

“make evaluates conditionals when it reads a makefile. Consequently, you cannot use automatic variables in the tests of conditionals because they are not defined until recipes are run...
To prevent intolerable confusion, it is not permitted to start a conditional in one makefile and end it in another. However, you may write an include directive within a conditional, provided you do not attempt to terminate the conditional inside the included file.”

In MacOS, “make” is part of the Xcode tools, and is in fact a version of GNU make:
https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/make.1.html

GNU make also supports conditional functions $(if condition, then-part [,else-part]), $(or ...), and ($and ...), as described here:
http://www.gnu.org/software/make/manual/html_node/Conditional-Functions.html#Conditional-Functions

The "and" and "or" functions short-circuit.

Makepp

http://makepp.sourceforge.net/
http://makepp.sourceforge.net/2.0/makepp_statements.html#ifeq_string1_string2

This is intended to be a "drop-in replacement for GNU make". Thus it supports the same "if ..." and "ifdef VARIABLE.." syntax.

It also supports
ifxxx...
or ifxxx...
else
endif

With "or" and "and" instead of xxx. "and" has a higher precedence than "or".

http://makepp.sourceforge.net/2.0/makepp_functions.html#if_string_result_if_string_not_blank_result_if_string_blank

It supports $(if string, result-if-string-not-blank[, result-if-string-blank]) and $(iftrue string, result-if-string-true[, result-if-string-false])
$(and condition1[,condition2[,condition3...]])
$(or condition1[,condition2[,condition3...]])
Note that "and" and "or" are again short-circuiting.

*BSD make

BSD make is also called "pmake". The FreeBSD make, NetBSD make, and OpenBSD are cousins.

FreeBSD info: http://www.freebsd.org/cgi/man.cgi?query=make&sektion=1
OpenBSD:
http://www.openbsd.org/cgi-bin/man.cgi?query=make&apropos=0&sektion=0&manpath=OpenBSD+Current&arch=i386&format=html
NetBSD: http://netbsd.gw.com/cgi-bin/man-cgi?make++NetBSD-current

A longer description (though slightly historial) is in “Pmake - A tutorial” by Adam de Boor, available at:
http://www.freebsd.org/doc/en/books/pmake/

http://docs.freebsd.org/44doc/psd/12.make/paper.pdf
However, “Pmake - A tutorial” is much older. For example, it documents conditionals as “#if …”, not “.if …”, so there have been changes since then. Nevertheless, it can provide more detail about the original intent.

I’ve been told that, as of FreeBSD 10, FreeBSD and NetBSD make are the same thing.

They look as follows (this is from FreeBSD):

.if [!]expression [operator expression ...]
Test the value of an expression.

.ifdef [!]variable [operator variable ...]
Test the value of a variable.

.ifndef [!]variable [operator variable ...]
Test the value of a variable.

...
where expression can use parentheses, &&, ||, or ! to negate, includes functions defined(var), make(var), empty(var), exists(filename), target(targetname). Also supports == and !=, and >, >, <, <=.

One nastiness: It seems to "automatically figure out" if it's dealing with numbers or strings for == and !=, that's asking for trouble. Basically, if they both “look like” numbers (and “0x” prefix performs a hex conversion), they are numbers; otherwise they are strings.

Good news: uses ".if" so no namespace pollution problems.

Microsoft nmake (part of Visual Studio 2012)

Note that Microsoft nmake is not related to AT&T nmake or Alcatel-Lucent nmake.

General:
http://msdn.microsoft.com/en-us/library/dd9y37ha.aspx
Conditions:
http://msdn.microsoft.com/en-us/library/7y32zxwh.aspx

Its syntax uses !IF, !IFDEF, etc. As far as I can tell they don’t try to comply with POSIX, so I’m not focusing on it.

AT&T nmake

Note that AT&T nmake is not related to Microsoft nmake. Alcatel-Lucent nmake is a descendent of AT&T nmake, so see the Alcatel-Lucent info.

More info here:
http://web.archive.org/web/20130605191027/http://www2.research.att.com/~gsf/nmake/nmake.html

Note: I'm using archive.org URLs in this case, because of:
http://www.unix.com/whats-your-mind/237119-david-korn-glenn-fowler-laid-off.html

The nmake documentation says, "Although **nmake** makefiles are not compatible with UNIX **make** or Microsoft **nmake**, **nmake** can generate UNIX **make** or Microsoft **nmake** style makefiles for porting and bootstrapping." However, it also says that nmake will consult the following files (in order): "Nmakefile, nmakefile, Makefile, and then makefile". Thus, it appears that nmake is designed to be able to handle at least some makefiles, so it seems relevant to consider it.

Fowler's "A Case for make" (Software - Practice and Experience, Vol. 20 No. S1, pp. 30-46, June 1990) shows "if … elif … else … end" as the syntax, where "x" == "y" is the expression syntax. Sadly, it doesn't explain the conditional syntax in any detail. It does note that conditional statements are big advantage: "A few makefile language extensions allow for more concise abstractions. Notice that the last example is just a cumbersome emulation of a makefile procedure call and conditional test. An if-else construct would allow a single project makefile to test for target types…".

I expect the URLs to switch to *astopen.org* prefixes soon.

I have only found limited documentation on AT&T nmake. I've found more about Alcatel-Lucent nmake; see that below. I expect that much of the Alcatel-Lucent nmake documentation applies to both.

Alcatel-Lucent nmake

This is based on AT&T nmake.

Basic info here: http://nmake.alcatel-lucent.com/
http://nmake.alcatel-lucent.com/manual/15/nmake.html

Those don't provide a lot of detail. Thankfully, there is a user's guide and reference manual (in

addition to the man page) available at: http://nmake.alcatel-lucent.com/manual/ and these provide MUCH more detail.

This nmake supports #if, etc., as cpp directives, but its User Guide specifically says, “Makefiles should not contain # directives. State variables and programming constructs should be used instead.” Thus, we’ll ignore the cpp directives and focus on the nmake built-in capabilities.

In nmake, there are normal variables and integer variables. Integer variables are assigned using “let var = …”. This difference in types is important in expressions, including those for “if”.

The nmake user’s guide explains its conditionals as follows:

"Each expression may be a combination of arithmetic and string expressions where the string expression components must be quoted. "..." strings use shell pattern matching (e.g., "string" == "pattern") whereas ’...’ strings use string equality. Empty strings evaluate to 0 in arithmetic expressions and non-empty strings evaluate to non-zero. Expression components may also contain optional variable assignments. All computations are done using signed long integers."

```
if expression
...
elif expression
...
else
...
end
```

The nmake 15 user’s guide gives this example (page 131):

```
CHAP1 = sect1a.tr
CHAP2 = sect2a.tr sect2b.tr
if "$(DFLAGS)" == "post"
        POST = dpost
elif "$(DFLAGS)" == "i300"
        POST = dimpress
elif "$(DFLAGS)" == "aps"
        POST = daps
end
```

The nmake 15 user’s guide page 127+ explains nmake expressions:

===

“Each expression can be a combination of arithmetic and string expressions where the string expression components must be quoted, i.e., "string".
Strings enclosed with double quotes use shell pattern matching scheme (for example, "$(VAR)" == "pattern"). Empty strings evaluate to 0 in arithmetic

expressions and non-empty strings evaluate to nonzero. Expression components can also contain optional variable assignments. All computations are done using signed long integers. For example:

```
if "$(VAR)" == "string"
        action
end
if v=2 < 4
        action
end
```

The string comparison requires both items on either side of the relational operator (e.g., ==) to be enclosed in double quotations. However, when one of the items is a string and the other is a variable in its unexpanded form (say, VAR == "string") nmake applies an implicit string conversion rule on VAR to enable string comparison. So, for the expression, VAR == "string" nmake essentially converts to "$(VAR:V)" == "string" for the comparison. The integer variable v is both assigned and compared on a single line (see Integer Variables -- The let Statement, below).

NOTE: The general rule for arithmetic and string expressions is;

```
if var
        true /* where var is non-null and !=0*/
else
        false /* where var=0 or var is null */
end
if "$(var)"
        true /* where var is non-null */
else
        false /* where var is null */
end
```

===

Just before this the user's guide explains expressions, focusing on integer expressions. It defines the operators available on page 130 as: "Arithmetic, Logic, and Bitwise Operators"

The arithmetic operators supported by nmake are listed below.

Operator Description
! Logical not
!= Not equal
% Modulo
& Bitwise AND
&& Logical AND
* Multiplication
+ Addition
- Subtraction
/ Division
< Less than

<< Bit shift left
<= Less than or equal to
== Logical equal
> Greater than
>= Greater than or equal to
>> Bit shift right
?: C-style operator (see below)
^ Bitwise exclusive OR
| Bitwise inclusive OR
|| Logical OR
~ Bitwise one's compliment

It also supports C-style "?:" operator of the form:
expression ? true_expression : false_expression

The nmake 15 reference document page 531 says:
"Each expression may be a combination of arithmetic and string expressions where the string expression components must be quoted. "..." strings use shell pattern matching (e.g., "string" == "pattern") whereas '...' strings use string equality. Empty strings evaluate to 0 in arithmetic expressions and non-empty strings evaluate to non-zero. Expression components may also contain optional variable assignments. All computations are done using signed long integers."

I think the nmake variable evaluation operations, e.g., $(VAR:op) where "op" is a long list of mysterious one-letter names is a terrible syntax; it seems designed to be impossible to guess what's going on. I think the GNU make function name syntax is far more readable. However, the "E - Evaluation" description of nmake gives more insight into expression evaluation.. if, in fact, they are the same thing. It's not clear that they are, but since the nmake documentation is poor on this point, I'll use it in hopes that it is. The reference document page 236 says:
===
"Logical expressions evaluate to 0 on false, and 1 on true. The available logical operators include:" where "==" is "equivalence comparison", and the other operators are "logical comparisons". These other ones are <, <=, >, >=, !=.
"Integer expressions can be given as logical comparisons or as integer evaluations. The logical operators listed above may be used in integer expressions as well as string comparisons. In addition, standard integer evaluation is available and includes the complete list of arithmetic operators….
Suppose a makefile contains an integer variable counter and a variable LIST, such that, if counter is even, the first token of LIST is needed, and if counter is odd, the second token is needed. The following statement produces the desired result:
$(LIST:O=$("(counter & 0x01)+1":E))
counter is an integer variable, and nmake expands its value whenever the name is encountered. Therefore, it is not necessary to specify the expansion using the syntax $(counter). The entire integer expression must be given within delimiting

double quotes. Also, parentheses are used for proper mathematical evaluation.
===

NOTE: This use of “if var” in nmake is at odds with how “if var” works in automake. One solution is to simply NOT define “if var” (where “var” is solo) NOT defined, or at least, not defined if “var” has the single value “0”.
NOTE: It’s really weird that the quote marks used (single vs. double) defines the OPERATOR does; I would expect that the OPERATOR would be different when you want a different operator (!). I’m not so excited about that aspect of the syntax, though it is not necessarily a killer.
NOTE: A challenge with this syntax is that if text is given WITHOUT a prefix, it is considered the name of an integer variable and NOT as unprocessed text. This disambiguates integer vs. string processing, but it is significantly different from makefile syntax elsewhere.
(It’s also worth noting that the nmakes also support “for…” which is useful.)

Opus Make

Opus Make (http://www.opussoftware.com/) was highly praised in http://freecode.com/articles/make-alternatives. However, doesn't seem to be actively maintained (it talks about getting the software for MS-DOS, OS/2, Win NT, and Win 95), so I’m more examining it for ideas.

Its syntax is “%if… %elif… %else .. %endif”. Comparing two string values uses “==”. Here’s an example, from its documentation:

```
%if $(CC) == bcc                # if compiler is bcc
  CFLAGS = –m$(MODEL)                 # $(CFLAGS) is –ms
  LDSTART = c0$(MODEL)                # the start-up object file
  LDLIBS = c$(MODEL)                  # the library
  LDFLAGS = /Lf:\bc\lib               # f:\bc\lib is library directory
%elif $(CC) == cl               # else if compiler is cl
  CFLAGS = –A$(MODEL,UC)              # $(CFLAGS) is –AS
  LDSTART =                     # no special start-up
  LDLIBS =                      # no special library
  LDFLAGS = /Lf:\c6\lib;        # f:\c6\lib is library directory
%else                           # else
% abort Unsupported CC==$(CC)    # compiler is not supported
```

%endif # endif

smake

(joerg@schily.isdn.cs.tu-berlin.de)
http://cdrecord.berlios.de/private/man/smake.html

No evidence from documentation that it supports if-then-else conditionals.

Fastmake

fastmake http://www.fastmake.org/ is a GNU make-like implementation; documentation is at http://www.fastmake.org/doc.html

Fastmake uses a ".IF …" syntax; here is an extract from its documentation:

"Fastmake introduces new condition syntax. This is because GNU make conditions (ifeq, ifneq) are limited to equality checking. The syntax of condition:

```
.IF <logical expression>
   statements
[.ELIF <logical expression>]
   statements
[.ELSE]
   statements
.END
```

where *logical expression* is the result of logical operations:
==, !=, &&, ||

A macro can be used as logical expression. In this case it is TRUE if a variable has been initialized to non-empty value. In the following example *statements* are executed if *PCH* macro has been defined.

```
.IF $(PCH)
   statements
.END
```

To check if a variable has been initialized to eigher empty or non-empty value use *ifdef* directive. Logical expressions are evaluated in-place. For example the output of this code will be *1*, not *2*

```
A = 1
```

```
.IF $(A) == 1
all :; @echo 1
.ELIF $(A) == 2
all :; @echo 2
.END
A = 2
```

To have a condition that evaluates on each usage *$(if)* function can be used."

It also has various performance improvements (e.g., commands run in same process). It has an include extension, "include <filename>", that includes a file starting from the makefile's directory (instead of the current directory)... that might be useful for non-recursive make. It has a built-in $(makedep …) function that determines C++ dependencies, usually called after a compile.

Oracle/Sun SunOS make

Documentation is at: http://download.oracle.com/docs/cd/E23824_01/html/821-1461/make-1s.html. A useful introductory article is "Introduction to the make utility" at http://developers.sun.com/solaris/articles/make_utility.html.

Remarkably, this appears to be one of the very few "make" implementations (besides smake) that lacks if-then-else. It documents using Bourne shell if-then-else, but that's different.

It has an ":=" operator that lets you set variables only when certain targets are being built, which is interesting.

Cmake (makefile generator)

Cmake generates makefiles, and has its own (different) syntax, so its syntax isn't really that relevant here. However, the fact that cmake generates makefiles and has conditionals is relevant, because if there was a standard format for conditionals, cmake could generate it.

Info: http://en.wikipedia.org/wiki/CMake http://www.cmake.org/

Its conditionals have the following slightly-wonky syntax:

```
if (${UNIX})
  set (DESKTOP $ENV{HOME})
else()
```

```
  set (DESKTOP $ENV{USERPROFILE}/Desktop)
endif()
```

Notice the “()” after else and endif.

Automake (makefile generator)

Info: http://www.gnu.org/software/automake/manual/automake.html#Conditionals

Automake preprocesses a makefile, and in certain conditions replaces or generates additional makefile entries. I’d like to create a conditional syntax that doesn’t conflict with automake, since automake is a widely-used tool.

Automake conditionals use "if... else... endif", but only a single macroname is allowed after "if", which is true if it's been set (by configure). Example:

```
if DEBUG
DBG = debug
else
DBG =
endif
noinst_PROGRAMS = $(DBG)
```

imake (X)

“imake” is a makefile generator using the cpp preprocessor. It was originally written for the X Window system. It was used by X from X11R1 (1987) to X11R6.9 (2005). X.org X11R7.0 replaced it with the GNU autotools (including automake), but they still maintain it for other users.

Since it’s based on cpp, it uses “#ifdef” etc.

Qmake (makefile generator)

Qmake is a makefile generator, included with Qt. Its syntax is described here: https://qt-project.org/doc/qt-5.1/qmake/qmake-language.html#test-functions

Qmake has “scopes” which are “similar to if statements in procedural programming languages. If a certain condition is true, the declarations inside the scope are processed…. Scopes consist of a condition followed by an opening brace on the same line, a sequence of commands and definitions, and a closing brace on a new line:”

```
<condition> {
    <command or definition>
```

```
    ...
}
```

Commentary

I think there's a need for a simple ifdef-like mechanism, and a full "if" capability in make. I focused on GNU and BSD make.

ifdef-like options

GNU and BSD make have different syntaxes and different semantics.

GNU make has "ifdef VARIABLE"..."else ANOTHER_CONDITION"..."else"..."endif". GNU make's "ifdef" syntax is clean and straightforward. One weakness in GNU make is that it does not support a multi-variable form like BSD "ifdef VARIABLE op VARIABLE …" where "op" can be "||" or "&&" - though that could be easily added to the spec, since current GNU make files would be compatible with this extension.

A problem with GNU make's syntax is that it doesn't begin with "." - which in theory could complicate using the syntax. In practice, no one will write a portable makefile using "ifdef" "else" or "endif"; GNU make is too popular, people would just "fix" such makefiles if they were intended to be portable.

GNU make's notion of what is "true" in ifdef is somewhat surprising, but I find it promising. In general GNU make considers an empty string to be false, and non-empty to be true. I think this is very good convention, since historically macros and variables are simply strings. These values are also easy to update in the command line. What's unusual about GNU make is that it considers the *unevaluated* form of the variable, e.g., if "x=$(y)" then x is always true for ifdef, even if "y" is empty. This is an unusual design choice, but I think it actually makes sense. This approach avoids the potentially exponential effort it takes to evaluate a variable; if you really want to force a value, use ::= (as has already been defined in POSIX).

BSD make in contrast uses ".ifdef [!]VAR1 op VAR2 …". I prefer the "||" and "&&" possibilities; this is more flexible. However, as far as I can tell, BSD make considers setting the macro at all as a definition. I do not see an obvious way to undefine a value at the command line, and there is no standard way to undefine something. This seems like an obvious problem; it's trivial to say "VAR=" at the command line and render it undefined in the GNU case, but not in the BSD case. BSD make's ".if empty(VAR)" has a similar effect to GNU make, but note that they are not interchangeable; "empty(VAR)" always evaluated VAR.

Full “if”

GNU make’s “ifeq” is incredibly limited. It has an ugly syntax, and can only compare to determine if two values are equal - no &&, ||, parentheses, “!” (not). There isn’t much to recommend GNU make’s ifeq.

BSD make’s “.if” has the advantage of providing &&, ||, and !. However, its “==” magically converts strings to numbers when it can (and supports 0x), which is dangerous. Its expression notation has a very few function operators (defined, empty, make, exists, target) that only work there (and not more generally). It also auto-magically adds “== 1” if not otherwise done, which is confusing and likely to cause trouble. BSD make’s variable expansion is really convoluted, using “:” followed by an array of letters (the GNU make variable expansion is much easier to read). Note that the GNU make and BSD make notions of “defined” cannot precisely emulate each other.

I think GNU make’s “ifeq” is a bad idea. Something like BSD make’s “.if” would be better, but the BSD semantics are a little dicey (it guesses numeric vs. string, for example). It might be better to create a new comparator syntax, one based on existing systems but combining the best features of each. Note that it’s probably a bad idea to use “if” by itself, since tools like automake have their own syntax. Perhaps something like “iftrue” or “.iftrue”.

The AT&T/Alcatel-Lucent “nmake” has “if…” as its syntax. The semantics of its conditionals are little unusual, but they do produce clean-looking code.

The final syntax should be consistent. E.G., if “ifdef” is used, then another lowercase term should be used for full “if”.

An alternative Desired Action - nmake-inspired version

We could develop a conditional system based on nmake. Here is some text that tries to do this.

Given the concerns listed above, I think the best approach is to add new syntax. This version builds on nmake for “if”. It uses GNU make’s “ifdef” for variable definitions (which are easier to disable on the command line or via include files), but I’ve added BSD make’s .ifdef capabilities which support ||, &&, and !. Both GNU make “ifeq” and BSD make’s “.if” have issues, so those are not directly included (though permitted as extensions). Existing simple GNU make files that only use “ifdef” or nmake “if” would be portable. Typical automake files passed through would also work. This proposal does not add macro functions, but is completely consistent with them; see bug report 512.

In line 97115, change “include lines,” to “include lines, conditional statements,”.

Before line 97166 (right after the “include lines”) section, add a new section, “Conditional statements” with the following content:

======

If the word “if”, “ifdef”, “ifndef”, “else”, “elif”, or “endif” appears at the beginning of a line after 0 or more <blank> characters, and if followed by or one or more <blank> characters or the end of line, the line is part of a conditional statement. A conditional statement is of the form:

<conditional_statement> ::= <begin_condition> <content>
(<continued_condition> <content>)*
(<else_condition> <content>)? <endif_line>
<begin_condition> ::= (“ifdef” <space>+ <ifdef_condition>
| “ifndef” <space>+ <ifdef_condition>
| “if” <space>+ <if_condition>) <eol>
<continued_condition> ::= “else” <space>+ <begin_condition>
| “elif” <space>+ <if_condition> <eol>
<else_condition> ::= “else” <eol>
<content> ::= <line>*
<endif_line> ::= “endif” | “end” <eol>
<eol> ::= <return>? <newline>
<ifdef_condition> ::= “(” <ifdef_condition> “)” |
“!”? <value> { “&&” <ifdef_condition> | “||” <ifdef_condition> }
<if_condition> ::= <expression> { “&&” <if_condition> | “||” <if_condition> }
<expression> ::= “(” <expression> “)” | “!” <space>+ <expression> ||
<value> [<op> <value>]
<value> ::= (<macro_expansion> | <other_character>)+
<op> ::= “==” || “!=” | “-lt” | “-le” | “-gt” | “-ge” | “-eq” | “-ne”

The precedence from highest to lowest is parentheses, “!” (not), “&&” (logical and) and” (logical or). When executed both “&&” and “||” short-circuit, left-to-right. An <escape_sequence> begins with “\” and is as defined in “awk”. An “other_character” is a character other than “$”, “\”, whitespace, or a control character (note that newline is a control character). A <macro_expansion> begins with “$” and is described in the “Macros” section below, e.g., “${HOME}” is a macro expansion. Outside of macro expansions, <space> characters in <ifdef_condition> and <if_condition> are ignored except that they separate tokens.

An “op” is one of the following operators: “==” (case-sensitive string equality), “!=” (case-sensitive string inequality), “-lt” (numeric less-than), “-le” (numeric less-than or equal-to), “-gt” (numeric greater-than), “-ge” (numeric greater-than or equal-to), “-eq” (numeric equality), or “-ne” (numeric inequality).

If an expression consists of just a value without the optional “op” it is considered true if it evaluates to at least one character, and false otherwise. In ifdef_condition, a <value> is evaluated and considered the name of a macro to determine if it is defined. A macro is considered defined if ifdef_condition if it has a nonempty value before evaluation (use “::=” to precalculate a macro).

Conditional statements are evaluated as the makefile is read in, in the order specified. If the relevant condition evaluates as true, then the content is read and used; otherwise its content is ignored.