# *make*/ GNU *make*

ACU 2006 acu@epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

December 14, 2005

# *make*/ GNU *make*

# make

# Bases

# Aim of *make*

With the shell command :
$ make program

## Definition

1. Wants to "make" *program*.
2. *program* can be a file.
3. less operations as possible.

# Vocabulary

when you type : $ make program
call of the **rule** *program* in the definitions file.

## Syntax

```
target : prerequisites
        command
```

## Vocabulary

- *program* is the **target** of the operation (or rule).
- built from one or more files : **prerequisites** or **dependents**.

# Vocabulary

when you type : $ make program
call of the **rule** *program* in the definitions file.

## Syntax

```
target : prerequisites
        command
```

## Vocabulary

- *program* is the **target** of the operation (or rule).
- built from one or more files : **prerequisites** or **dependents**.

# Vocabulary

when you type : $ make program
call of the **rule** *program* in the definitions file.

## Syntax

```
target : prerequisites
        command
```

## Vocabulary

- *program* is the **target** of the operation (or rule).
- built from one or more files : **prerequisites** or **dependents**.

# Definitions file

If no definitions file is given in command line :

1. look for *Makefile* or *makefile* in a specified `PATH`.
2. If no rule is specified on command line, the first defined in the definition file is used.

**but definitions file is not necessary !**

```
$ ls
toto.c
$ cat toto.c
int main()
{
}
$ make toto
```

# Definitions file

If no definitions file is given in command line :

1. look for *Makefile* or *makefile* in a specified `PATH`.
2. If no rule is specified on command line, the first defined in the definition file is used.

**but definitions file is not necessary !**

```
$ ls
toto.c
$ cat toto.c
int main()
{
}
$ make toto
```

# Definitions file

If no definitions file is given in command line :

1. look for *Makefile* or *makefile* in a specified `PATH`.
2. If no rule is specified on command line, the first defined in the definition file is used.

## but definitions file is not necessary !

```
$ ls
toto.c
$ cat toto.c
int main()
{
}
$ make toto
```

# Separated Compilation

# Problematic of huge project

## With huge project

- There are lot's of source files.
- gcc *.c is suicide.
- However compile C language source file is quickly, but with C++ it's too much longer.

$\Longrightarrow$

Necessity to compile only what is required.

# Problematic of huge project

## With huge project

- There are lot's of source files.
- gcc *.c is suicide.
- However compile C language source file is quickly, but with C++ it's too much longer.

$\Longrightarrow$

Necessity to compile only what is required.

# Naive example 1/3

```
$ cat hello.c
#include <stdio.h>
#include "hello.h"

void helloworld(void)
{
  printf("Hello World !\n");
}
```

```
$ cat hello.h
#ifndef   __HELLO_H__
# define   __HELLO_H__
```

```
$ cat main_hello.c
#include <stdlib.h>
#include "hello.h"

int main(void)
{
  helloworld();
  return EXIT_SUCCESS;
}
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 2/3

```
$ ls
hello.c hello.h main_hello.c Makefile
```

## naive Makefile for 'hello world'

```
helloworld: hello.o main_hello.o
    gcc hello.o main_hello.o -o helloworld
hello.o: hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
main_hello.o : main_hello.c hello.h
    gcc -Wall -Werror -ansi -pedantic -W -c main_hello.c -o
main_hello.o
```

# Naive example 3/3

## Use cases

```
$ ls
hello.c hello.o hello.h main_hello.c main_hello.o \
 helloworld
$ make
make: ‘helloworld’ is up to date

$ touch hello.c
$ make
gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
gcc hello.o main_hello.o -o helloworld
```

# Naive example 3/3

## Use cases

```
$ ls
hello.c hello.o hello.h main_hello.c main_hello.o \
 helloworld
$ make
make: ‘helloworld’ is up to date

$ touch hello.c
$ make
gcc -Wall -Werror -ansi -pedantic -W -c hello.c -o hello.o
gcc hello.o main_hello.o -o helloworld
```

# Macros

# Why use of macros ?

- Computer Scientist is feignant.
- Redundant code is :
  1. Source of bugs
  2. Harsh to maintain

# Why use of macros ?

- Computer Scientist is feignant.
- Redundant code is :
  1. Source of bugs
  2. Harsh to maintain.

# Why use of macros ?

- Computer Scientist is feignant.
- Redundant code is :
  1. Source of bugs
  2. Harsh to maintain.

# Why use of macros ?

- Computer Scientist is feignant.
- Redundant code is :
  1. Source of bugs
  2. Harsh to maintain.

# Why use of macros ?

- Computer Scientist is feignant.
- Redundant code is :
  1. Source of bugs
  2. Harsh to maintain.

$\Longrightarrow$ Use of macros.

# Definition Syntax

## Syntax

PARAMETER = VALUE

1. (double)Quoting the `VALUE` is not necessary.
2. Can use \ at end of line for multi-lines definition.
3. No tabulation in begin of line (to avoid ambiguity in grammar).

# Definition Syntax

## Syntax

PARAMETER = VALUE

1. (double)Quoting the `VALUE` is not necessary.
2. Can use \ at end of line for multi-lines definition.
3. No tabulation in begin of line (to avoid ambiguity in grammar).

# Definition Syntax

## Syntax

PARAMETER = VALUE

1. (double)Quoting the `VALUE` is not necessary.
2. Can use \ at end of line for multi-lines definition.
3. No tabulation in begin of line (to avoid ambiguity in grammar).

# Definition Syntax

## Syntax

PARAMETER = VALUE

1. (double)Quoting the `VALUE` is not necessary.
2. Can use \ at end of line for multi-lines definition.
3. No tabulation in begin of line (to avoid ambiguity in grammar).

# Definition Syntax

## Syntax

PARAMETER = VALUE

1. (double)Quoting the `VALUE` is not necessary.
2. Can use \ at end of line for multi-lines definition.
3. No tabulation in begin of line (to avoid ambiguity in grammar).

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

**Warning !!!**

```
FOO = BAR
BAR = $FOO

BAR = "BAROO"
```

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

**Warning !!!**

```
FOO = BAR
BAR = $FOO

BAR = "BAROO"
```

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

**Warning !!!**

```
FOO = BAR
BAR = $FOO

BAR = "BAROO"
```

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

**Warning !!!**

```
FOO = BAR
BAR = $FOO

BAR = "BAROO"
```

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

## Warning !!!

```
FOO = BAR
BAR = $FOO

BAR = "BAROO"
```

# How to use?

- `$(MACRO)`
- `${MACRO}`
- `$M`

## Warning !!!

```
FOO = BAR
BAR = $FOO
```

BAR = "BAROO"

# Substitutions 1/2

## Example

```
FOO = foo
BAR = bar_${FOO}
```

⇓

## Environment

```
FOO = foo
BAR = bar_foo
```

## Example

```
FOO = foo_${BAR}
BAR = bar
```

⇓

## Environment

```
FOO = foo_bar
BAR = bar
```

$\Longrightarrow$ Definition order doesn't matter.

# Substitutions 1/2

## Example

```
FOO = foo
BAR = bar_${FOO}
```

⇓

## Environment

```
FOO = foo
BAR = bar_foo
```

## Example

```
FOO = foo_${BAR}
BAR = bar
```

⇓

## Environment

```
FOO = foo_bar
BAR = bar
```

$\Longrightarrow$ Definition order doesn't matter.

# Substitutions 1/2

## Example

```
FOO = foo
BAR = bar_${FOO}
```

⇓

## Environment

```
FOO = foo
BAR = bar_foo
```

## Example

```
FOO = foo_${BAR}
BAR = bar
```

⇓

## Environment

```
FOO = foo_bar
BAR = bar
```

$\Longrightarrow$ Definition order doesn't matter.

# Substitutions 1/2

## Example

```
FOO = foo
BAR = bar_${FOO}
```

⇓

## Environment

```
FOO = foo
BAR = bar_foo
```

## Example

```
FOO = foo_${BAR}
BAR = bar
```

⇓

## Environment

```
FOO = foo_bar
BAR = bar
```

$\Longrightarrow$ Definition order doesn't matter.

# Substitutions 1/2

**Example**

```
FOO = foo
BAR = bar_${FOO}
```

⇓

**Environment**

```
FOO = foo
BAR = bar_foo
```

**Example**

```
FOO = foo_${BAR}
BAR = bar
```

⇓

**Environment**

```
FOO = foo_bar
BAR = bar
```

$\Longrightarrow$ Definition order doesn't matter.

# Substitutions 2/2

## Recursive Definition

```
BAR = bar_${FOO}
FOO = foo_${BAR}
all:
        echo ${BAR}
```

$\Longrightarrow$

*** Recursive variable 'BAR' references itself (eventually). Stop.

## Warning !!!

Recursive definitions are not allowed.

# Substitutions 2/2

## Recursive Definition

```
BAR = bar_${FOO}
FOO = foo_${BAR}
all:
        echo ${BAR}
```

$\Longrightarrow$

*** Recursive variable ‘BAR’ references itself (eventually). Stop.

## Warning !!!

Recursive definitions are not allowed.

# Substitutions 2/2

## Recursive Definition

```
BAR = bar_${FOO}
FOO = foo_${BAR}
all:
        echo ${BAR}
```

$\Longrightarrow$

*** Recursive variable ‘BAR’ references itself (eventually). Stop.

## Warning !!!

Recursive definitions are not allowed.

# Restrictions

## Undefine variable in prerequesites

```
all: $(FOO)
	echo all

bar:
	echo foo
FOO = bar
```

⇓

```
$ make all
echo all
```

## Now defined before prerequesites

```
FOO = bar
all: $(FOO)
	echo all

bar:
	echo foo
```

⇓

```
$ make all
echo foo
```

# Restrictions

## Undefine variable in prerequesites

```
all: $(FOO)
	echo all

bar:
	echo foo
FOO = bar
```

⇓

```
$ make all
echo all
```

## Now defined before prerequesites

```
FOO = bar
all: $(FOO)
	echo all

bar:
	echo foo
```

⇓

```
$ make all
echo foo
```

# Restrictions

## Undefine variable in prerequesites

```
all: $(FOO)
	echo all

bar:
	echo foo
FOO = bar
```

⇓

```
$ make all
echo all
```

## Now defined before prerequesites

```
FOO = bar
all: $(FOO)
	echo all

bar:
	echo foo
```

⇓

```
$ make all
echo foo
```

# Restrictions

## Undefine variable in prerequesites

```
all: $(FOO)
	echo all

bar:
	echo foo
FOO = bar
```

⇓

```
$ make all
echo all
```

## Now defined before prerequesites

```
FOO = bar
all: $(FOO)
	echo all

bar:
	echo foo
```

⇓

```
$ make all
echo foo
```

# Restrictions

## Undefine variable in prerequesites

```
all: $(FOO)
	echo all

bar:
	echo foo
FOO = bar
```

⇓

```
$ make all
echo all
```

## Now defined before prerequesites

```
FOO = bar
all: $(FOO)
	echo all

bar:
	echo foo
```

⇓

```
$ make all
echo foo
```

# Predefined Variables

| VARIABLE | DEFAULT VALUE | Definition |
|---|---|---|
| CC | cc | C language compiler |
| CFLAGS | | FLAGS for C compiler |
| CXX | g++ | C++ language compiler |
| CXXFLAGS | | FLAGS for C++ compiler |
| RM | rm -f | To remove a file |
| LDFLAGS | | FLAGS give to `ld` |
| LEX | lex | Lex Scanner |
| YACC | yacc | Yacc Parseur |
| TEX | tex | Convert TEX to DVI files |

# Priority of names environments

Order of priority, from least to greatest :

1. Internal (or default) definitions.
2. Shell environment variables.
3. Description file macro definitions.
4. Macros specified in `make` command line.

# Priority of names environments

Order of priority, from least to greatest :

1. Internal (or default) definitions.
2. Shell environment variables.
3. Description file macro definitions.
4. Macros specified in `make` command line.

# Priority of names environments

Order of priority, from least to greatest :

1. Internal (or default) definitions.
2. Shell environment variables.
3. Description file macro definitions.
4. Macros specified in `make` command line.

# Priority of names environments

Order of priority, from least to greatest :

1. Internal (or default) definitions.
2. Shell environment variables.
3. Description file macro definitions.
4. Macros specified in `make` command line.

# Macro string substitution

```
SRC = foo.c bar.c Oops.h
OBJ = $(SRC:.c=_obj.o)
```

$\Longrightarrow$ in the environnement :

```
OBJ = foo_obj.o bar_obj.o Oops.h
```

# Macro string substitution

```
SRC = foo.c bar.c Oops.h
OBJ = $(SRC:.c=_obj.o)
```

$\Longrightarrow$ in the environnement :

```
OBJ = foo_obj.o bar_obj.o Oops.h
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Helloworld with macros

```
# use of variables provide by gmake like ‘CC’ and ‘CFLAGS’
# definition of the ‘OUT’ variable for the binary.
CC = gcc
CFLAGS = -Wall -Werror -ansi -pedantic -W
OUT = helloworld
${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o ${OUT}
hello.o:  hello.c hello.h
     ${CC} ${CFLAGS} -c hello.c -o hello.o
main_hello.o :  main_hello.c hello.h
     ${CC} ${CFLAGS} -c main_hello.c -o main_hello.o
```

# Internals Macros

# Definition

## Some Macros have a special meaning

| Macro | Available | Meaning |
|---|---|---|
| $@ | command line | target name |
| $? | command line | prerequesites newer than target |
| $$@ | prerequisites | target name |

# Examples

```
$ cat Makefile
foo : Makefile $$@_bar
        echo "current target: $@"
        echo "newer prereq: $?"
        touch $@

foo_bar:
$ ls
Makefile
$ make
echo "current target: foo"
current target: foo
echo "newer prereq: Makefile foo_bar"
newer prereq: Makefile foo_bar
```

```
$ ls
Makefile foo
$ make
echo "current target: foo"
current target: foo
echo "newer prereq: "
newer prereq:
touch foo
$ touch Makefile
$ make
echo "current target: foo"
current target: foo
echo "newer prereq: Makefile"
newer prereq: Makefile
```

# Suffix Rules

# Definition

## Definition

1. Thanks to convention, possibility to define suffix rules
2. Suffix rule .A.B. Generic rule to create a file B from a file A.
3. Someone are defined by default
4. Possibility to overload them or define others.

## Caution

Extensions that appears in your own suffix rule be be specified as prerequisites of `.SUFFIXES` special target.

# Definition

## Definition

1. Thanks to convention, possibility to define suffix rules
2. Suffix rule .A.B. Generic rule to create a file B from a file A.
3. Someone are defined by default
4. Possibility to overload them or define others.

## Caution

Extensions that appears in your own suffix rule be be specified as prerequisites of `.SUFFIXES` special target.

# Definition

## Definition

1. Thanks to convention, possibility to define suffix rules
2. Suffix rule .A.B. Generic rule to create a file B from a file A.
3. Someone are defined by default
4. Possibility to overload them or define others.

## Caution

Extensions that appears in your own suffix rule be be specified as prerequisites of `.SUFFIXES` special target.

# Definition

## Definition

1. Thanks to convention, possibility to define suffix rules
2. Suffix rule .A.B. Generic rule to create a file B from a file A.
3. Someone are defined by default
4. Possibility to overload them or define others.

## Caution

Extensions that appears in your own suffix rule be be specified as prerequisites of `.SUFFIXES` special target.

# Examples 1/2

## .c.o

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

## Definition

- Describe how to create a object file from a C language source file.
- $< has a meaning akin to $?
- But only in suffix rules context.

# Examples 1/2

## .c.o

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

## Definition

- Describe how to create a object file from a C language source file.
- $< has a meaning akin to $?
- But only in suffix rules command line!

# Examples 1/2

## .c.o

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

## Definition

- Describe how to create a object file from a C language source file.
- $< has a meaning akin to $?
- But only in suffix rules command line!

# Examples 1/2

## .c.o

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

## Definition

- Describe how to create a object file from a C language source file.
- $< has a meaning akin to $?
- But only in suffix rules command line!

# Examples 2/2

## Use of $*

```
.c.o:
        cp $< $*.tmp
        /* modification of the tmp file */
        $(CC) $(CFLAGS) -c $*.tmp -o $@
```

## Definition

$* is the filename part of the prerequisite.

# Examples 2/2

## Use of $*

```
.c.o:
        cp $< $*.tmp
        /* modification of the tmp file */
        $(CC) $(CFLAGS) -c $*.tmp -o $@
```

## Definition

$* is the filename part of the prerequisite.

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:

     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:

     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:

     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@
```

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:

     ${CC} ${CFLAGS} -c $< -o $@
```

No dependencies with headers

# Helloworld

```
# use of :
# * implicit rule
# * variables associated with implicit rules
# * automatic variables

CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic -W
OUT     = helloworld

${OUT}:  hello.o main_hello.o
     ${CC} hello.o main_hello.o -o $@

.c.o:
     ${CC} ${CFLAGS} -c $< -o $@

hello.o:  hello.h
```

# Double Colons Operator

# Definition

## Definition

1. Possibility to define multi prerequisites for a rule :

   ```
   foo: bar1 bar2 ...
   ```

2. But only 1 command line for a target.

## Example

```
foo:
        @echo foo1
foo:
        @echo foo2

$ make
Makefile:4: warning: overriding commands for target ‘foo’
Makefile:2: warning: ignoring old commands for target ‘foo’
```

# Definition

## Definition

1. Possibility to define multi prerequisites for a rule :

   ```
   foo: bar1 bar2 ...
   ```

2. But only 1 command line for a target.

## Example

```
foo:
        @echo foo1
foo:
        @echo foo2

$ make
Makefile:4: warning: overriding commands for target ‘foo’
Makefile:2: warning: ignoring old commands for target ‘foo’
```

# Definition

## Definition

1. Possibility to define multi prerequisites for a rule :

   ```
   foo: bar1 bar2 ...
   ```

2. But only 1 command line for a target.

## Example

```
foo:
        @echo foo1
foo:
        @echo foo2

$ make
Makefile:4: warning: overriding commands for target ‘foo’
Makefile:2: warning: ignoring old commands for target ‘foo’
```

# Definition

## Definition

1. Possibility to define multi prerequisites for a rule :

   ```
   foo: bar1 bar2 ...
   ```

2. But only 1 command line for a target.

## Example

```
foo:
        @echo foo1
foo:
        @echo foo2

$ make
Makefile:4: warning: overriding commands for target ‘foo’
Makefile:2: warning: ignoring old commands for target ‘foo’
```

# Example

## Multi command for a rule

```
foo::
        @echo bar1
foo::
        @echo bar2

$ make
bar1
bar2
```

## Caution

Use of : xor :: for all occurences of a target.

# Example

## Multi command for a rule

```
foo::
        @echo bar1
foo::
        @echo bar2

$ make
bar1
bar2
```

## Caution

Use of : xor :: for all occurences of a target.

# Command line

# Filename Pattern Matching

## Available characters

| character | behaviour |
|---|---|
| * | the same as shell globbing |
| ? | the same as shell globbing |
| [ ] | the same as shell globbing |

# Environment

1. Each line is executed in its own shell.
2. Environments are independant.

**My sources files!**

```
clean:
        cd obj
        $(RM) *
```

***make* user!**

```
clean:
        cd obj ; \
        $(RM) *
```

# Environment

1. Each line is executed in its own shell.
2. Environments are independant.

**My sources files!**

```
clean:
        cd obj
        $(RM) *
```

***make* user!**

```
clean:
        cd obj ; \
        $(RM) *
```

# Environment

1. Each line is executed in its own shell.
2. Environments are independant.

**My sources files!**

```
clean:
        cd obj
        $(RM) *
```

***make* user!**

```
clean:
        cd obj ; \
        $(RM) *
```

# Environment

1. Each line is executed in its own shell.
2. Environments are independant.

## My sources files!

```
clean:
        cd obj
        $(RM) *
```

## *make* user!

```
clean:
        cd obj ; \
        $(RM) *
```

# Environment

1. Each line is executed in its own shell.
2. Environments are independant.

## My sources files!

```
clean:
        cd obj
        $(RM) *
```

## *make* user!

```
clean:
        cd obj ; \
        $(RM) *
```

# Shell Variables

## *make* variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $FOO
```

## shell variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $$FOO
```

# Shell Variables

## *make* variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $FOO
```

$\Longrightarrow$ foo

## shell variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $$FOO
```

# Shell Variables

## *make* variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $FOO
```

$\Longrightarrow$ foo

## shell variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $$FOO
```

# Shell Variables

## *make* variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $FOO
```

$\Longrightarrow$ foo

## shell variable

```
FOO = foo
all:
        FOO=bar ; \
        echo $$FOO
```

$\Longrightarrow$ bar

# Special Built-in Targets

# Echoing & .SILENT 1/2

## Definition

1. By default, commands are displayed before beeing executed.
2. When a line starts with '@', the echoing of that line is suppressed.

### Without

```
foo:
        echo foo


$ make foo
echo foo
```

### With

```
foo:
        @echo foo


$ make foo
```

# Echoing & .SILENT 1/2

## Definition

1. By default, commands are displayed before beeing executed.
2. When a line starts with '@', the echoing of that line is suppressed.

### Without

```
foo:
        echo foo


$ make foo
echo foo
```

### With

```
foo:
        @echo foo


$ make foo
```

# Echoing & .SILENT 1/2

## Definition

1. By default, commands are displayed before beeing executed.
2. When a line starts with '@', the echoing of that line is suppressed.

### Without

```
foo:
        echo foo


$ make foo
echo foo
```

### With

```
foo:
        @echo foo


$ make foo
```

# Echoing & .SILENT 1/2

## Definition

1. By default, commands are displayed before beeing executed.
2. When a line starts with '@', the echoing of that line is suppressed.

## Without

```
foo:
        echo foo

$ make foo
echo foo
```

## With

```
foo:
        @echo foo

$ make foo
```

# Echoing & .SILENT 1/2

## Definition

1. By default, commands are displayed before beeing executed.
2. When a line starts with '@', the echoing of that line is suppressed.

## Without

```
foo:
        echo foo

$ make foo
echo foo
```

## With

```
foo:
        @echo foo

$ make foo
foo
```

# Echoing & .SILENT 2/2

## .SILENT special target

- Do not displayed command of targets given as prerequisites of .SILENT.
- if .SILENT is specified without prerequisites, *make* doesn't print any command before executing them.

# Echoing & .SILENT 2/2

## .SILENT special target

- Do not displayed command of targets given as prerequisites of .SILENT.
- if .SILENT is specified without prerequisites, *make* doesn't print any command before executing them.

# Error in commands 1/2

## Behaviour

1. *rule* completes successfully if all its commands returns 0.
2. *Commands* of a *rule* are executed one after the others.
3. If an error occured, execution is stopped and the rule is not completed.

# Error in commands 1/2

## Behaviour

1. *rule* completes successfully if all its commands returns 0.
2. *Commands* of a *rule* are executed one after the others.
3. If an error occured, execution is stopped and the rule is not completed.

# Error in commands 1/2

## Behaviour

1. *rule* completes successfully if all its commands returns 0.
2. *Commands* of a *rule* are executed one after the others.
3. If an error occured, execution is stopped and the rule is not completed.

# Errors in commands 2/2

## Definition

1. Possibility to ignore returns value.

```
foo: bar
        -exit 1
        echo foo

bar:
        -exit 1
```

2. Rules specified as prerequisites of `.IGNORE` special target, theirs execution is ignored.
3. You can also use the option -i in command line to have this behaviour for all commands.

# Errors in commands 2/2

## Definition

1. Possibility to ignore returns value.

```
foo: bar
        -exit 1
        echo foo

bar:
        -exit 1
```

2. Rules specified as prerequisites of `.IGNORE` special target, theirs execution is ignored.
3. You can also use the option -i in command line to have this behaviour for all commands.

# Errors in commands 2/2

## Definition

1. Possibility to ignore returns value.

```
foo: bar
        -exit 1
        echo foo

bar:
        -exit 1
```

2. Rules specified as prerequisites of `.IGNORE` special target, theirs execution is ignored.
3. You can also use the option -i in command line to have this behaviour for all commands.

# Use of .DEFAULT rule

## Definition

.DEFAULT target is used when no rule if found for a specified target.

## Example

```
foo:
        @echo $@
.DEFAULT:
        @echo "default : $@"

$ make foo
foo
$ make bar
```

# Use of .DEFAULT rule

## Definition

.DEFAULT target is used when no rule if found for a specified target.

## Example

```
foo:
        @echo $@
.DEFAULT:
        @echo "default : $@"

$ make foo
foo
$ make bar
```

# Use of .DEFAULT rule

## Definition

.DEFAULT target is used when no rule if found for a specified target.

## Example

```
foo:
        @echo $@
.DEFAULT:
        @echo "default : $@"

$ make foo
foo
$ make bar
```

# gmake

# Automatic Variables

# List of Prerequisites 1/2

## Definition

1. The variable $\$\hat{}$ contains list of all prerequisites.
2. Only in the command line.
3. Multi occurence of prerequesite $\longmapsto$ appears only once in $\$\wedge$.

# List of Prerequisites 1/2

## Definition

1. The variable $\$\hat{}$ contains list of all prerequisites.
2. Only in the command line.
3. Multi occurence of prerequesite $\longmapsto$ appears only once in $\$\wedge$.

# List of Prerequisites 1/2

## Definition

1. The variable $\$^\wedge$ contains list of all prerequisites.
2. Only in the command line.
3. Multi occurence of prerequesite $\longmapsto$ appears only once in $\$\wedge$.

# List of Prerequisites 2/2

## Example

```
helloworld: $(OBJ)
        $(CC) $^ -o $@
```

# Stem Variable

# Stem Automatic Variable 1/2

## Definition

1. Possibility to define generic target :
2. Example :

```
t-%:
        @echo target: $@
```

rule match the regular expression t-%

# Stem Automatic Variable 1/2

## Definition

1. Possibility to define generic target :
2. Example :

```
t-%:
	@echo target: $@
```

rule match the regular expression : `t-*`

# Stem Automatic Variable 1/2

## Definition

1. Possibility to define generic target :
2. Example :

```
t-%:
	@echo target: $@
```

rule match the regular expression : `t-*`

# Stem Automatic Variable 1/2

## Definition

1. Possibility to define generic target :
2. Example :

```
t-%:
	@echo target: $@
```

rule match the regular expression : `t-*`

# Stem Automatic Variable 2/2

## Example

```
foo_%.o: %.txt
        @echo $*

bar.txt:
        @echo $@

$ make foo_bar.o
```

# Stem Automatic Variable 2/2

## Example

```
foo_%.o: %.txt
        @echo $*

bar.txt:
        @echo $@

$ make foo_bar.o
```

# Stem Automatic Variable 2/2

## Example

```
foo_%.o: %.txt
        @echo $*

bar.txt:
        @echo $@

$ make foo_bar.o

bar.txt
bar
```

# Variable assignment

There are two ways :

## Classic *make* expansion

```
x = foo\\
y = $(x) bar
x = later

all:
        @echo $y
```

## Simply variable expansion

```
x := foo\\
y := $(x) bar
x := later

all:
        @echo $y
```

# Variable assignment

There are two ways :

## Classic *make* expansion

```
x = foo\\
y = $(x) bar
x = later

all:
        @echo $y
```

## Simply variable expansion

```
x := foo\\
y := $(x) bar
x := later

all:
        @echo $y
```

# Variable assignment

There are two ways :

## Classic *make* expansion

```
x = foo\\
y = $(x) bar
x = later

all:
        @echo $y
```

$\Longrightarrow$ later bar

## Simply variable expansion

```
x := foo\\
y := $(x) bar
x := later

all:
        @echo $y
```

# Variable assignment

There are two ways :

## Classic *make* expansion

```
x = foo\\
y = $(x) bar
x = later

all:
        @echo $y
```

## Simply variable expansion

```
x := foo\\
y := $(x) bar
x := later

all:
        @echo $y
```

# Variable assignment

There are two ways :

## Classic *make* expansion

```
x = foo\\
y = $(x) bar
x = later

all:
        @echo $y
```

## Simply variable expansion

```
x := foo\\
y := $(x) bar
x := later

all:
        @echo $y
```

$\Longrightarrow$ foo bar

# Conditionnal Part

# Introduction

## Definition

Like preprocessing in C.

## syntax

```
conditional-directive
text-if-true
endif
```

```
conditional-directive
text-if-true
else
text-if-false
endif
```

# Introduction

## Definition

Like preprocessing in C.

## syntax

```
conditional-directive
text-if-true
endif
```

```
conditional-directive
text-if-true
else
text-if-false
endif
```

# ifeq / ifneq

Different correct directives

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

## Behaviour

1. Evaluation of the first argument
2. Evaluation of the second argument
3. For ifeq, execute the *text-if-true* if their are identical. Otherwise execute the *text-if-false*
4. For ifneq, it has the contrary

# ifeq / ifneq

Different correct directives

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

## Behaviour

1. Evaluation of the first argument
2. Evaluation of the second argument
3. For ifeq, execute the *text-if-true* if their are identical. Otherwise execute the *text-if-false*
4. For ifneq, it has the contrary

# ifeq / ifneq

Different correct directives

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

## Behaviour

1. Evaluation of the first argument
2. Evaluation of the second argument
3. For ifeq, execute the *text-if-true* if their are identical. Otherwise execute the *text-if-false*
4. For ifneq, it has the contrary

# ifeq / ifneq

Different correct directives

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

## Behaviour

1. Evaluation of the first argument
2. Evaluation of the second argument
3. For ifeq, execute the *text-if-true* if their are identical. Otherwise execute the *text-if-false*
4. For ifneq, it has the contrary

# ifeq / ifneq

Different correct directives

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

## Behaviour

1. Evaluation of the first argument
2. Evaluation of the second argument
3. For ifeq, execute the *text-if-true* if their are identical. Otherwise execute the *text-if-false*
4. For ifneq, it has the contrary

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO =
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO =
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO =
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO = kikoo
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO = kikoo
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 1/2

## Behaviour

The condition if true if the variable given as parameter has a non-empty value.

```
FOO = kikoo
all:
ifdef FOO
    @echo FOO is defined
else
    @echo FOO is undefined
endif
```

# ifdef / ifndef 2/2

## Caution

GNU *make* only checks that the variable is not-empty.
It doesn't try to expand it :

```
BAR =
FOO = $(BAR)
all:
ifdef FOO
        @echo FOO is defined
else
        @echo FOO is undefined
endif
```

# ifdef / ifndef 2/2

## Caution

GNU *make* only checks that the variable is not-empty.
It doesn't try to expand it :

```
BAR =
FOO = $(BAR)
all:
ifdef FOO
        @echo FOO is defined
else
        @echo FOO is undefined
endif
```

# Functions

# Call syntax

## Syntax

```
$(function arguments)
OR
${function arguments}
```

## Example

Call of

```
$(sort foo bar lose)
```

returns the value ‘bar foo lose’.

# Call syntax

## Syntax

```
$(function arguments)
OR
${function arguments}
```

## Example

Call of

```
$(sort foo bar lose)
```

returns the value ‘bar foo lose’.

# String substitutions & analysis

There are many functions. Most important are :

| Function | Example of call | Result |
|---|---|---|
| patsubst | `$(patsubst %.c,%.o,x.c.c bar.c)` | `x.c.o bar.o` |
| findstring | `$(findstring a,a b c)` | a |
| filter | `$(filter %.c %.h, f.c b.s f.h` | f.c f.h |
| words | `$(words foo.c foo.h)` | 2 |
| word | `$(word 2 foo.c foo.h main.c)` | foo.h |

# Functions for file names

There are many functions. Most important are :

| Function | Example of call | Result |
|---|---|---|
| dir | `$(dir src/foo.c foo.h)` | `src/ ./` |
| addsuffix | `$(addsuffix .c,foo bar)` | `foo.c bar.c` |
| addprefix | `$(addprefix src/,f.c b.c)` | `src/f.c src/b.c` |
| wildcard | `$(wildcard *.[ch])` | `foo.c bar.h` |

# if function

# if function

## syntax

```
$(if condition,then-part[,else-part])
```

## Behaviour

1. expand condition
2. strip preceeding and trailing whitespace
3. if non-empty string → the second argument is evaluated
4. Otherwise the third argument is evaluated

# if function

## syntax

```
$(if condition,then-part[,else-part])
```

## Behaviour

1. expand condition
2. strip preceeding and trailing whitespace
3. if non-empty string $\Longrightarrow$ the second argument is evaluated. Otherwise the third argument is evaluated.

# if function

## syntax

```
$(if condition,then-part[,else-part])
```

## Behaviour

1. expand condition
2. strip preceeding and trailing whitespace
3. if non-empty string $\Longrightarrow$ the second argument is evaluated. Otherwise the third argument is evaluated.

# if function

## syntax

```
$(if condition,then-part[,else-part])
```

## Behaviour

1. expand condition
2. strip preceeding and trailing whitespace
3. if non-empty string $\Longrightarrow$ the second argument is evaluated. Otherwise the third argument is evaluated.

# foreach function

# foreach function 1/2

## Syntax

```
$(foreach var,list,text)
```

## Behaviour

- Like *for* statement in bourne shell.
- Use repeatedly *text* each time with a different substitution performed on it.

# foreach function 1/2

## Syntax

```
$(foreach var,list,text)
```

## Behaviour

- Like *for* statement in bourne shell.
- Use repeatedly *text.* each time with a different substitution performed on it.

# foreach function 1/2

## Syntax

```
$(foreach var,list,text)
```

## Behaviour

- Like *for* statement in bourne shell.
- Use repeatedly *text.* each time with a different substitution performed on it.

# foreach function 1/2

## Syntax

```
$(foreach var,list,text)
```

## Behaviour

- Like *for* statement in bourne shell.
- Use repeatedly *text*. each time with a different substitution performed on it.

# foreach function 2/2

## Example

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

# eval function

# Parametric variable

## Example

```
foo = $(2) $(1)
```

## Multi-line definition

**syntax :**

```
define variable
endef
```

**Example :**

```
define foo
$(2) $(1)
```

# Parametric variable

## Example

```
foo = $(2) $(1)
```

## Multi-line definition

**syntax :**

```
define variable
endef
```

**Example :**

```
define foo
$(2) $(1)
```

# Parametric variable

## Example

```
foo = $(2) $(1)
```

## Multi-line definition

**syntax :**

```
define variable
endef
```

**Example :**

```
define foo
$(2) $(1)
```

# eval function 1/3

## Syntax

```
$(call variable,param,param,..)
```

## Behaviour

Expand *variable* according to *params*

## Properties

1. No maximum of parameter argument
2. each *param* is assign to $(1), $(2) ...
3. $(0) contains *variable*

# eval function 1/3

## Syntax

```
$(call variable,param,param,..)
```

## Behaviour

Expand *variable* according to *params*

## Properties

1. No maximum of parameter argument
2. each *param* is assign to $(1), $(2) ...
3. $(0) contains *variable*

# eval function 1/3

## Syntax

```
$(call variable,param,param,..)
```

## Behaviour

Expand *variable* according to *params*

## Properties

1. No maximum of parameter argument
2. each *param* is assign to `$(1)`, `$(2)` ...
3. `$(0)` contains *variable*

# eval function 2/3

```
_swap = $(2) $(1)

foo = $(call _swap,a,b)
```

*foo* variable will contains `b a`

# eval funtion 1/3

```
PROGRAMS    = server client

server_OBJS = server.o server_priv.o server_access.o
server_LIBS = priv protocol

client_OBJS = client.o client_api.o client_mem.o
client_LIBS = protocol

# Everything after this is generic

.PHONY: all
all: $(PROGRAMS)

define PROGRAM_template
 $(1): $$($(1)_OBJ) $$($(1)_LIBS:%=-l%)
 ALL_OBJS   += $$($(1)_OBJS)
```

# value function

# the value function

## Syntax

```
$(value variable)
```

## Behaviour

get the value of *variable* without having it expanded.

## Example

```
FOO = $PATH

all:
        @echo $(FOO)
        @echo $(value FOO)
```

# the value function

## Syntax

```
$(value variable)
```

## Behaviour

get the value of *variable* without having it expanded.

## Example

```
FOO = $PATH

all:
        @echo $(FOO)
        @echo $(value FOO)
```

# the value function

## Syntax

```
$(value variable)
```

## Behaviour

get the value of *variable* without having it expanded.

## Example

```
FOO = $PATH

all:
        @echo $(FOO)
        @echo $(value FOO)
```

# The shell function

# The shell function

## Behaviour

1. Execute the command shell given in parameter.
2. Returns the command output.
3. Replace each newline of the result with a single space.

## Example

```
contents := $(shell cat foo)
```

# The shell function

## Behaviour

1. Execute the command shell given in parameter.
2. Returns the command output.
3. Replace each newline of the result with a single space.

## Example

```
contents := $(shell cat foo)
```

# The shell function

## Behaviour

1. Execute the command shell given in parameter.
2. Returns the command output.
3. Replace each newline of the result with a single space.

## Example

```
contents := $(shell cat foo)
```

# Functions to control make

1. make
2. gmake
   - Automatic Variables
   - Stem Variable
   - Conditionnal Part
   - Functions
   - if function
   - foreach function
   - eval function
   - value function

# error & warning function

## Error

generate a fatal error. Display the argument on the error output.

```
ifdef ERROR1
$(error error is $(ERROR1))
endif
```

## Warning

Like *error* but *make* doesn't exit. processing of the Makefile continues.

# Bibliography


The art of unix programming.
http://www.catb.org/~esr/writings/taoup/.


The gmake home page.
http://www.gnu.org/software/make.


Recursive make considered harmful.
http://aegis.sourceforge.net/auug97.pdf.