MAKE

Make

- Flujo de trabajo de programación:
  - *Editar código fuente*
  - *Compilar*
  - *Debugging*
- A medida que el proyecto crece, estas tareas se vuelven más tediosas.
- *Make* nos permite automatizar los aspectos mundanos de transformar el código fuente a un ejectuable.

Make

- Los archivos que enviamos a `make` son llamados **makefiles**. Para el programa “Hello World”, el archivo makefile se vería así:

```
hello: hello.c
	gcc hello.c -o hello
```

- Luego, estando en el directorio donde está este archivo, ejecutamos:

```
$ make
gcc hello.c -o hello
```

- El **makefile** contiene una serie de **reglas**. La primera regla es **regla por defecto.**
- Una **regla** consiste de:
  - *Objetivo* → *archivo o cosa a construir*
  - *Pre-requisitos* → *archivos que deben existir ANTES de construir el objetivo*
  - *Comandos* → *Comandos a ejecutar*

```
objetivo: pre-req1 pre-req2
        comando1
        comando2
```

- Cuando `make` **evalua** una regla, empieza buscando los pre-requisitos y el objetivo. Primero trata de construir los pre-requisitos, y finalmente el objetivo.

Make

- Veamos un ejemplo, un programa para contar las ocurrencias de las palabras fee, fie, foe y fum, usando el scanner `flex`:

```
#include <stdio.h>

extern int fee_count, fie_count, foe_count, fum_count;

extern int yylex( void );

int main(int argc, char ** argv)
{
        yylex();
        printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
        exit( 0 );
}
```

- El scanner esta definido en `lexer.l`

```
int fee_count = 0;
int fie_count = 0;
int foe_count = 0;
int fum_count = 0;
%%
fee fee_count++;
fie fie_count++;
foe foe_count++;
fum fum_count++;
```

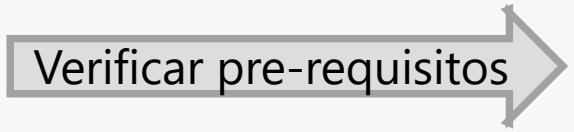
- Usaremos flex para generar el archivo `lexer.c` desde `lexer.l`

- El **makefile** sería:

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

- Si llamamos `make`, lo que hará el programa es construir el programa count_words

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```




```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

esperar

compilar

```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

esperar

compilado

Verificar pre-req.

```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```


```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

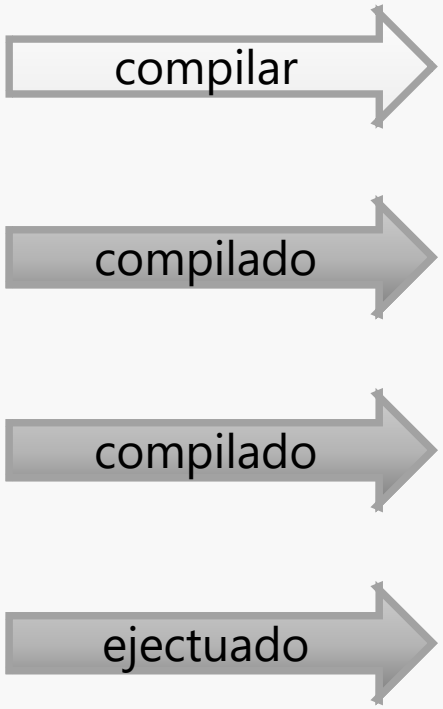
esperar

compilado

compilar

ejecutado

```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```




```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
	gcc -c count_words.c

lexer.o: lexer.c
	gcc -c lexer.c

lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

compilado

```
count_words: count_words.o lexer.o -lfl
	gcc count_words.o lexer.o -lfl -ocount_words
```

compilado

```
count_words.o: count_words.c
	gcc -c count_words.c
```

compilado

```
lexer.o: lexer.c
	gcc -c lexer.c
```

ejectuado

```
lexer.c: lexer.l
	flex -t lexer.l > lexer.c
```

- Digamos que modificamos `lexer.l` y corremos `make` de nuevo:

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

- `count_words.c` NO fue recompilado.
- Podemos llamar una regla en particular directamente:

```
$ make lexer.c
make: `lexer.c' is up to date.
```

- Si la regla no existe:

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.
Stop.
```

Make

- Los comandos en cada regla, empiezan siempre con un tab (tecla Tab)

- El símbolo para inicar una línea comentario es #

- Veamos las reglas en más detalle
  - ***Reglas explícitas*** → *especifican los objetivos y pre-requisitos:*
    - Una regla puede contener más de un objetivo
    - Objetivos son manejados de manera independiente.

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
        ...
```

  - *Esto equivale a:*

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
variable.o: make.h config.h getopt.h gettext.h dep.h
```

  - *Preferible una objetivo por regla.*

- Podemos tener el mismo objetivo en varias reglas
  - *Esto hará que make busque todas las dependencias para el objetivo en particular*

```
vpath.o: vpath.c make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h
vpath.h
```

  - *Distintas reglas para un mismo objetivo pueden manejarse de manera distinta:*

```
# Make sure lexer.c is created before vpath.c is compiled.
vpath.o: lexer.c
...
# Compile vpath.c with special flags.
vpath.o: vpath.c
        $(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<
...
# Include dependencies generated by a program.
include auto-generated-dependencies.d
```

Make

- **Wildcards**

```
prog: *.c
        $(CC) -o $@ $^
```

- Esta regla compilar el programa prog, cuyos pre-requisitos son TODOS (*) los archivos .c en el directorio
  - *S@ → nombre del objetivo que esta siendo procesado.*
  - *$^ → nombre de todos los prerrequisitos (con espacios)*
  - *Son llamadas variables automáticas:*
  - *https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html*

Make

Objetivos *phony*

Usados para reglas que no generan o actualizan archivos. Aquí podemos especificar cualquier comando que usaríamos en la terminal

```
clean:
        rm -f *.o lexer.c
```

```
$ make clean
make: `clean' is up to date.
```

Make

Objetivos *phony*

Para evitarlo, usamos la directiva .PHONY

```
.PHONY: clean
clean:
	rm -f *.o lexer.c
```

Make

Variables

- Se definen antes de las REGLAS.

```
VARIABLE=valor
```

- Para usarlas en las reglas: `$(VARIABLE)`

```
CC=gcc

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

Make

Variables automáticas

- Establecidas por make una vez que una regla es correspondida
- `$@` → nombre de archivo del objetivo
- `$%` → elemento de nombre de una especificación de miembro de archivo
  - *archivo(miembro) → S@ dá "archivo", S% dá "miembro"*
- `$<` → nombre del primer pre-requisito.
- `$?` → nombres de todos los pre-requisitos más actuales que el objetivo.
- `$^` → nombres de todos los pre-requisitos separados con un espacio
- `$+` → igual que $^, incluidos duplicados
- `$*` → La raíz de un nombre de archivo. Usualmente el nombre sin la extensión

- Ejemplo del **makefile** modificado del contador de palabras

```
count_words: count_words.o counter.o lexer.o -lfl
	gcc $^ -o $@

count_words.o: count_words.c
	gcc -c $<

counter.o: counter.c
	gcc -c $<

lexer.o: lexer.c
	gcc -c $<

lexer.c: lexer.l
	flex -t $< > $@
```