GREP:

- It is a program to find patterns in text files.
- With a combination of commands, grep is a powerful tool through which we an find our required pattern.
- You might have faced "Unresolved reference" warning from the linker. The grep command can be used to find the misspelled module (if the module was misspelled).
- Grep finds the required pattern using the Regular expressions, commonly known as regex.
- Regex expressions are a combination of characters and special symbols that are called as metacharacters

Exploration of the meta characters:

^ - It matches the beginning of a line

$ - It matches the end of a line

\ - It matches with a character placed next to it. Ex; ***** matches with the character ***** (asterisk) , **,** with a comma, **]** with a closing rectangular bracket and so on…

. -Matches with any character.

[] – This specifies a character class. Any single character that is mentioned in the brackets as a string matches with any character that is present within the string.

Ex: **[qwe]** matches with **q** or **w** or **e**.

- A range of character can be using by using []
- If the first symbol is mentioned as ^ then, a negative character class is specified.
- Ex of the above statement would be:
  [^a-z] that means, the string matches all characters except lower letters.

***** It matches zero or more matches of the regular expression.

+ A Regex followed by + matches one or more matches of the regular expression.

ee Two concatenated regular expressions march a match of the first followed by a match of the second.

| Two or more regular expressions can be separated by using **|**. It would match the first or the next expression that are separated by **|** .

The order of precedence of the above are as below:

[]
*
concatenation
|
newline

— —

Many UNIX systems have 3 versions of grep. **grep**, **egrep** and **fgrep** .

Among the mentioned, **egrep** is the powerful operator.

As an example

egrep -nf index.exp *.c is typed on the cmd line.

Here **-nf** means prefix each line of output with the 1 based line number within its input file and obtain the patterns from the mentioned file, one per line, **index.exp** is a file and ***.c** represents all the c files in the current directory.

The **index.exp** contains the following lines:

^[a-z A-Z _]+.*([^;]*) [^;]*$
^#define[ ]+[a-z A-Z _] +(

The first line says:

search for the pattern from beginning of the line ^
that is followed by any letter or underscore **[a-z A-Z _]**
and repeated at least one time **+**
followed by any character repeated zero or more times **.***
and followed by open parenthesis **(**
followed by any character except a semicolon that repeated zero or more times **[^;]***
and followed by a closed parenthesis **)**
followed any character except a semicolon **[^;]***
repeated zero or more time that is followed by the end of line **$**

The Second line says:

Search for the pattern from the beginning of line ^
followed by the string **#define**
followed by at least one space or tab **[ ]+**
and one tab character (tab between the brackets)
followed by at least one letter or underscore **[a-z A-Z _]**
and followed immediately by an open parenthesis **(**

Instead of using a file, the above can be directly implemented by using the | as following:
egrep "^**[a-z A-Z _]+.*([^;]*) [^;]*$|^#define[ ]+[a-z A-Z _] +(." ***.c**

Yes, it is confusing, hence we can choose the file option.

LINT

- It is a C syntax checker.
- It find things like number of parameters passed to subroutines
- parameters of wrong type
- return value of a subroutine that is not assigned to a variable of correct type.
- It is also helpful in flagging
    - potential errors
    - unusual use of operators
    - truncation caused by implicit type conversion….
- Although lint is useful in finding simple errors, it is also the one that causes errors.(funny but true)
- Linit is a version of compiler, hence , instead of generating the error messages as a code, it would get them as a output.
- Lint would be operating on the source code , so there is no equivalent linker.

MAKE

- Okay, why do you even use it? A typical question among beginners, even to me, until I started to integrate and compile and recompile multiple files. Make is a friendly utility that gets rid of a lot of headache to the coder.
- Make is an automated program compilation utility
- Consider you have multiple files, some like 40, and you have changed some 17 in them. They are interlinked, now it is difficult to recompile those modified files individually. Even if you write a small script to compile all 40 files, its waste of compilation time to compile the remaining unchanged files. By using the make, the utility automatically recompiles only the modified files. Cool, isn't it?
- The make utility is also available in MS-DOS
- The make utility takes input a *makefile.*
- A *makefile* is a gfile that describes all the modes in a large program and the relationship between those modules.

An example of usage of make

The objective is to create an executable file named *farm.*

Consider there are 3 files named *cow.c, pig.c* and *farm.c* . the *pig.c* and *farm.c* have another header named *animals.h*

The final executable depends on the object files i.e *cow.o*, *pig.o* and *farm.o*

The make utility creates a relation on the way we implement the script in the makefile. To create a simple makefile, we can create a file named "***Makefile***" by using the touch command or some editor such as vi or nano and input the following script.

```
farm: cow.o pig.o farm.o
        cc -o farm cow.o pig.o farm.o
cow.o: cow.c stdio.h animals.h
        cc -c cow.c
pig.o: pig.c stdio.h animals.h
        cc -c pig.c
farm.o: farm.c stdio.h
       cc -c farm.c
```

Be careful with the tabs and spaces when you are scripting this.

- It seems difficult to understand when you try to read it from top to bottom. Instead, follow the bottom up approach. The object file farm.o is dependent on farm.c and stdio.h and in the next line, you are instruction what you need to do with the c file. Yours simply linking. The same goes with pig.o and cow.o. Once all the object files are create you are creating an executable file named *farm* . That executable is dependent on the object files of cow, pig and farm. Once you mention them in that manner, and instruct what to do, your makefile is done.
- The final command would be simply running the "**make**" on the command line.

Well, Stay tuned, see you soon.