Problem #1

```
echo "`date +%d`>=15" | bc | mail -s "Payday yet?" <email address>
```

Explanation

- `echo "`date +%d`>=15"`

  This command mainly creates an expression that, when evaluated by another command, returns whether the date is on or after 15th.

  - `date +%d`

    This command within backquotes is going to be replaced with its output, which in turn will be part of the string being echoed. The command prints the date of the month. The "+" sign signifies that a flag list follows, which will format the output of date. The %d sign is the flag for the date of the month.

  - `echo "`date +%d`>=15`

    Prints a string which is of the following form: *day of the month*>=15. For example, if today is March 12, this command prints *12 > = 15*

- `bc`

  This is a command used for calculations. In this context, it takes the output expression of the previous command and prints 1 if True or 0 if False

- `mail -s "Payday yet?" <email address>`

  Sends an email to the specified recepient with the subject as "Payday yet?". The body is the output of *bc*

Problem #2

```
grep "^`date +%a`" weekly.txt | mailx -s "Today's tasks" <email address>
```

Explanation

- `grep "^`date +%a`" weekly.txt`

  This command searches for a line beginning with today's day of the week in *weekly.txt* and prints it out.

  - `date +%a`

    This command within backquotes is going to be replaced with its output, which in turn will be part of the string being used as a search expression by grep. It prints the day of the week in an abbreviated form. Sunday is Sun, Monday is Mon, etc.

  - `"^`date +%a`"`

    This regular expression searches for lines beginning with today's day. Omitting searching for the string in the beginning of the line will have unintended consequences like selecting lines which use the day of the month elsewhere.

    Example: Something like "*Sun: Get Wedding preparations done*" will be selected on Wednesday as well as

Sunday, which would be a mistake.

- `mailx -s "Today's tasks" <email address>`

  As in the previous solution, this merely sends an email with the subject line as "Today's tasks" to the given email address with the body being the output of grep.

Problem #3

```
0 6 * * 1-5 /home/<NetID>/problem2.sh
```

Explanation

- `0 6 * * 1-5 /home/<NetID>/problem2.sh`

  This line, when added into the crontab file by issuing the command *crontab -e* will schedule a task in system. The meaning of each argument here is:

  - `0`: Minute
  - `6`: Hour is 6AM
  - `*`: Run every day of the month
  - `*`: Run every month
  - `1-5`: Run only from Monday to Friday.
  - `/home/<NetID>/problem2.sh`: Command to run at the specified time, which is the solution to Problem 2.

Problem #4

```
paste -d "\n" song1.txt song2.txt
```

Explanation

- `paste -d "\n" song1.txt song2.txt`

  The paste command merges the lines from multiple files. In the above command, it is used to read lines from both `song1.txt` and `song2.txt` files alternately to create a remix.

  The `-d` option specifies the delimiter to use between the lines. New line character is specified as delimiter since we want to lines to appear alternately in the remixed file.

Problem #5

```
find cs -type f -exec grep \# {} + 2> /dev/null | sed 's/.*\///' | tr -d ':#'
```

Explanation

- `find cs -type f -exec grep \# {} + 2> /dev/null`

  This command does a number of things as explained below:

  - `find cs`

    finds all files within the `cs` directory and its subdirectories.

- `-type f`

  option instructs the find command to look for only files and not directories.

- `-exec grep \# {} +`

  part of the command uses grep to look for `#` within the files and it runs `grep` once for multiple files.

- `+`

  option at the end allows grep to be run once on multiple files instead of running once for each matched file.

- `2> /dev/null`

  redirects all errors (2 represents standard error stream) to the null stream. This helps to avoid display of any errors on the stdout.

- `sed 's/.*\///'`

  Sed is used to remove all the directory names and keep only the net ids.

  `.*\/`

  is a regular expression which removes everything till the last `/` (slash), which results in deletion of all the directory names and only the file names (NetIds) remain.

- `tr -d ':#'`

  The output from the previous command leaves us with the NetIds followed by ":#" (which is due to the output from grep in the first step).

  `tr -d ":#"`

  removes all ":#" from the output, resulting in the final NetIds of all the students who got an A+ in the class.

Problem #6

```
find cs -type f -exec cp {} all \;
```

Explanation

- `find cs -type f -exec cp {} all \;`

  This command finds all files (only files and not directories as specified by `-type f` option) in the `cs` directory and all its subdirectories and copies each file to the `all` directory.

  `-exec cp {} all \;`

  This part of the command runs `cp` command once for each file given by find command. The `\;` at the end specifies that `cp` must be run once for each file.

  After this command is run, `all` directory contains all the student files with no directory structure.

Problem #7

```
find . -type f -exec grep -i "^from:" {} \; | grep -Eoi "[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}" |
sort -u &
```

Explanation

- `find . -type f`

  Find all files in the currect directory and all subdirectories.

- `-exec grep -i "^from:" {} \;`

  Once files are found, we are going to use -exec to run grep on the files. Basically, this option is telling "whenever you find a file matching the criteria I gave you, substitute {} with the file name and run the command." Let's say **find** found a file named "abc.html". Then it will basically run this following command and put it onto the pipeline: *grep -i "^from:" abc.html*

  We are using a semicolon at the end. Why? You should read this if you are not clear which one you should use.

- `grep -Eoi "[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"`

  We're grepping what's passed from the previous find command. *E* option means that we'll be using the extended RegEx syntax. *i* option is for case-insensitive matching, and *o* option displays only portions that match.

  Now to the regular expression (source). This attempts to match "name@domain.suffix" form. [A-Z0-9._%+-]+ will match one or more characters that are alphabets, numbers, or any of these special characters: ._%+-. It's similar with [A-Z0-9.-]+. Then [A-Z]{2,4} matches two-to-four-letter-long alphabet words.

  In practice, the email address format is surprisingly complex, and there is no single regular expression that can match all valid email addresses one hundred percent. Check out these examples: some of the addresses look like they were typed by cats walking on keybaords, yet they are all valid email addresses! We do not expect students to come up with a regular expression that matches all valid email addresses, so any reasonable regular expressions in the form of "name@domain" will be accpted as correct answers.

- `sort -u`

  This sorts and lists only unique elements. It creates an identical result as "sort | unique" does. We couldn't use this in the previous assignment because we needed *-c* option for unique: "sort | unique -c".

- `&`

  This puts the commands to run in the background.

Problem #8

```
grep -Eo "[[:alpha:]]+" text.txt | grep -ivwf english.txt
```

Explanation

- `grep -Eo "[[:alpha:]]+" text.txt`

  -E flag declares the usage of extended RegExp expressions. -o flag displays only matches (without this, grep will display a whole line when only a part of it contains a match.)

  This command takes input and extracts substrings that consist of one or more alphabet letters. For example, when a phrase *gentleman's* is fed, it will print out *gentleman* and *s* on separate lines. We are doing this because we don't

want to look up the whole phrase *gentleman's* with the apostrophe on the dictionary.

- `grep -ivwf english.txt`

  This is the major workhorse. To explain the options,

  - **i**: matches case-insensitively
  - **v**: invert the sense of matching, to select non-matching lines.
  - **w**: word-wise matching. Each matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Maybe the following example will help you understand this:

    echo "rebus" | grep -w "bus"
    echo "busted" | grep -w "bus"
    echo "re bus" | grep -w "bus"

    Although "rebus" contains "bus", the first command will not print anything. Similarly, the second string begins with "bus", but it will print nothing. The third command displays "bus" because it is a word by itself.
  - **f**: retrieve query strings from a file. In this case, from short_english.txt.

Problem #9

```
sed -E 's/(.+,)*([A-Za-z]+),[A-Za-z]+;.+/\2/g' restaurants.txt
```

Explanation

- `sed -E 's/(.+,)*([A-Za-z]+),[A-Za-z]+;.+/\2/g' restaurants.txt`

  E flag declares the usage of extended RegExp expressions.

  This regular expression matches "name,name;" which are the two last names in the party, together with everything that may come before, which is captured by (.+,)* and after it, namely a ; followed by the name of the restaurant, captured by ;.+. Note this command will ignore statements about forever-alone people who visited restaurants by themselves (example: Zulma;Just A Taste). It will save the second to last name in the register called "\2" (we needed to use parenthesis to apply the start operator in (.+,)* and the expression within parenthesis got saved in \1). Finally, sed will substitute the whole lines that were captured by the regular expression with the value stored in the register \2.

Problem #10

```
grep -Eo "(\(?[[:digit:]]{3}\)?\s*[.-]?\s*)?[[:digit:]]{3,4}\s*[.-]?[[:digit:]]{4}" phone-data.txt
```

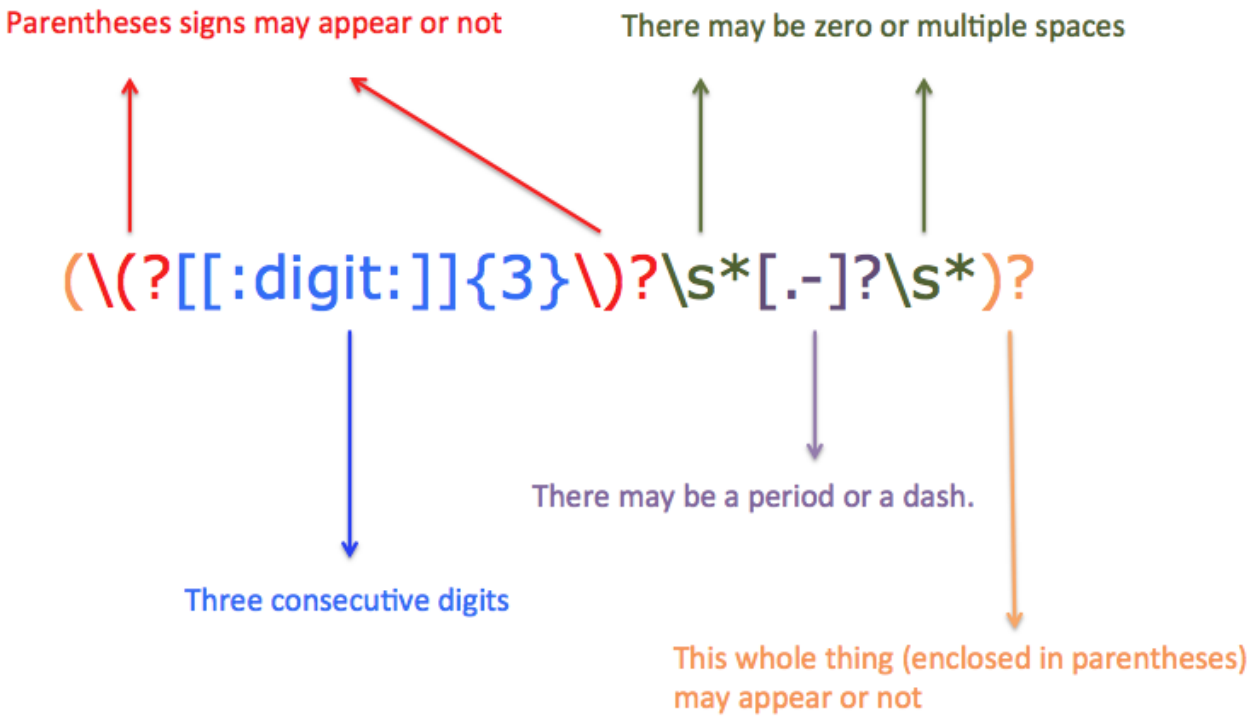
Explanation

- `grep -Eo`

  -E flag declares the usage of extended RegExp expressions. -o flag displays only matches (without this, grep will display a whole line when only a part of it contains a match.)

- `(\(?[[:digit:]]{3}\)?\s*[.-]?\s*)?`

  This is for matching the area code. *?* sign means that a preceding literal either appears only once or does not appear. The whole thing is enclosed in ()? because the area code may not appear at all.



- `[[:digit:]]{3,4}\s*[.-]?`

  This matches the middle digits of phone numbers. There must be either 3 or 4 consecutive numbers.

- `[[:digit:]]{4}"`

  This matches the last four digits of phone numbers.
