Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2023-2024

Homework 1 - Extended Shell

Due date: 31 03 2024, Sunday, 23:59

1 Overview

In this homework, you will implement an extended shell called *eshell*, which supports the sequential and parallel execution of programs together with a pipeline. It should also support subshell operations.
Keywords: *unix, shell*

2 Background

A pipeline is a sequence of processes chained together. Each process's standard output (stdout) feeds the standard input (stdin) of the following process via pipes, which provide unidirectional inter-process communication support.

A pipeline can be created using '|' delimiter on a Unix Shell such that:

```
{command1} | {command2} | ... | {commandN}
```

Where each *command* contains an executable name with command line arguments, when a pipeline is created, all processes run concurrently and the shell blocks until all processes in the pipeline terminate.

The following is a sample command that a regular shell supports:

```
ps aux | grep python | wc -l
```

where the list of all user processes containing the python is counted.

Sequential execution is a list of commands separated by a semicolon `;` in the format:

```
{command1} ; {command2} ; ... ; {commandN}
```

Where the *command*s are executed one after the other. Let us look at an example:

```
echo "Hello"; sleep 5; echo "World"
```

This Example will print `Hello`, wait 5 seconds, and then print `World`. Pipelines can be incorporated into sequential execution. This means that pipelines can be executed one after another instead of commands, or they can be mixed. For Example:

```
echo "Hello" | cat ; cat config.txt; cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/
```

In this example `echo "Hello" | cat` executed first, then `cat config.txt`, and finally `cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/` will be executed.

Parallel execution is an extension to the UNIX shell you will implement in this homework. Similar to the sequential execution, it will contain a list of commands. Unlike sequential execution, though, they will all be executed concurrently. The separator for parallel execution will be comma `,`. If we look at the first sequential Example:

```
echo "Hello", sleep 5, echo "World"
```

This Example will print `Hello`, `World` and wait 5 seconds in parallel.

Like sequential execution, parallel execution can contain a pipeline. Similar to the previous Example:

```
echo "Hello" | cat , cat config.txt , cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/
```

In this example `echo "Hello" | cat`, `cat config.txt`, and `cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/` will be executed in parallel.

Finally, subshells are a group of programs executed in a child shell. They can contain sequential execution or parallel execution but not both. The syntax for a subshell operation is parenthesis `(sequential or parallel commands)`. Example:

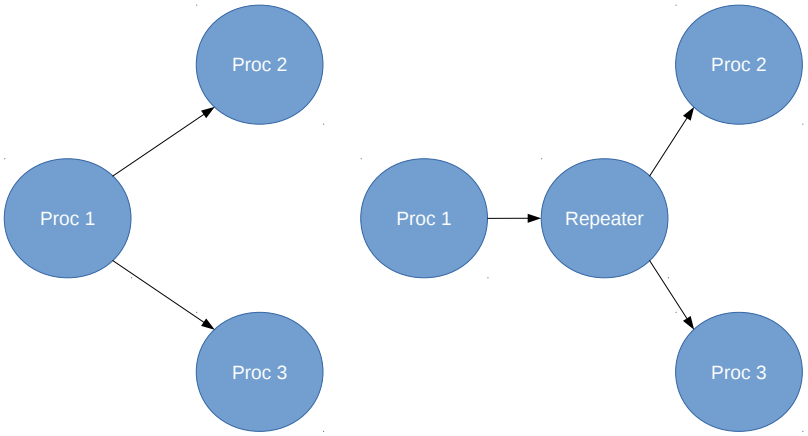
```
(echo "Hello"; sleep 3; echo "World")
(echo "Hello", sleep 2, echo "World")
```

Subshells are considered similar to individual programs. The commands inside are executed in a child shell launched by the parent program. Their I/O can be redirected to another process or subshell inside a pipeline. Sequential Example:

```
(echo "Hello"; cat config.txt; cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/) |
(cat | wc -l; echo "Finished")
```

In this Example, the output of the first subshell is redirected to the second subshell. The first pipeline in the second subshell reads the output and prints its results.

For parallel execution, there is a difference. Suppose there is an input pipe to a subshell containing parallel commands. In that case, you must create a repeater process that replicates the input and then sends it to related processes with additional pipes. It would look like this:



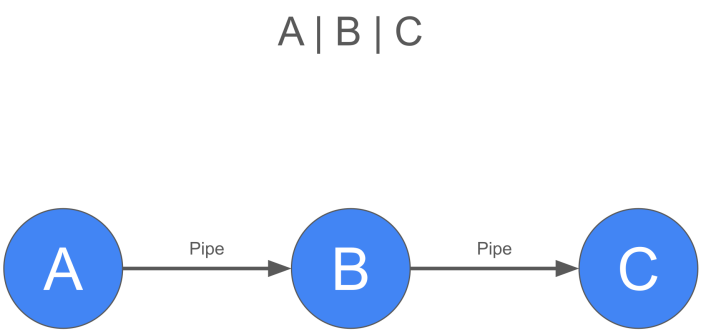
If we look at the previous Example:

```
(echo "Hello"; cat config.txt; cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/) |
(wc -l, wc -c)
```

The outputs of `echo "Hello"`, `cat config.txt`, and `cat manifest.txt | grep "Ship" | tr /a-z/ /A-Z/` commands will be replicated and sent to `wc -l` and `wc -c` commands. Both `wc -l` and `wc -c` receive the full input.

It is important to note that **PIPE** operation takes precedence over sequential or parallel operators. This means pipelines will be executed first if there is no subshell. Let us look at some generic examples:

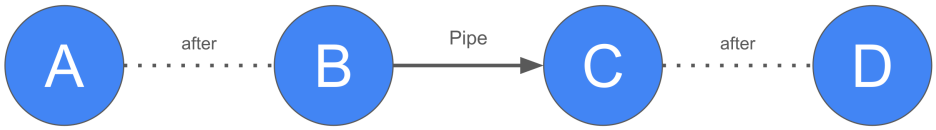
- Simple pipe operations where the process will be piped to one another and run together.



Process A's output will be redirected to a pipe connected to process B's input. Similarly, process B's output will be redirected to a pipe connected to process C's input. It is also important to note that these processes are executed concurrently.

- Sequential and pipe execution where we can see the precedence.

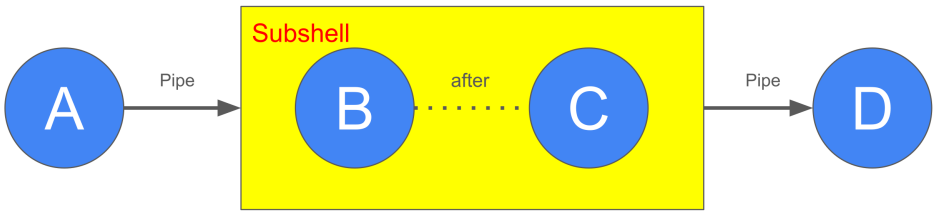
A ; B | C ; D



It is clear that instead of `A;B` piped to `C;D`, we are executing process A first, then the `B|C` pipeline and finally the process D.

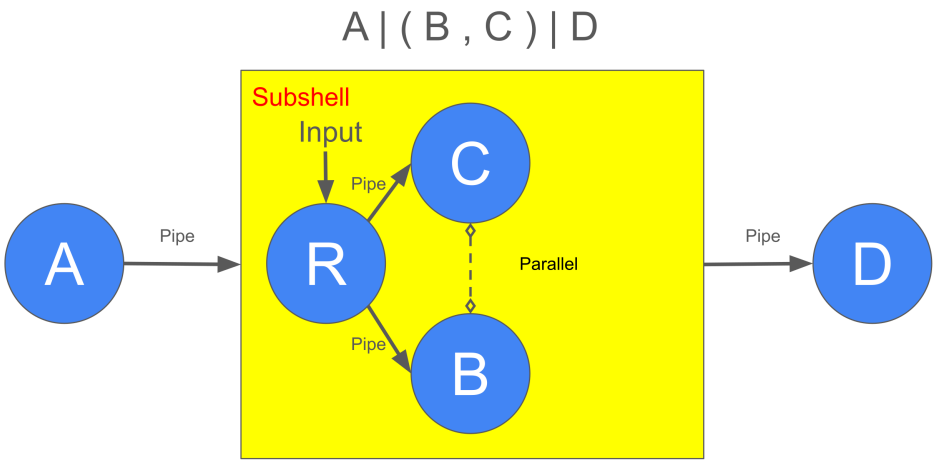
- Pipeline that includes a subshell containing sequential execution.

A | (B ; C) | D



We can see that process A's output is piped to the entire subshell, and the subshell is piped to process D. Inside the subshell, B and C are executed sequentially.

- Pipeline that includes a subshell containing sequential execution.



This pipeline is similar to the previous Example. The only difference is that B and C are executed in parallel and should receive the entire output of A coming into the subshell. More correctly, it should receive the entirety of the subshell's input. Therefore, there should be a repeater to replicate the incoming input to the B and C processes.

3 Extended Shell

Your task is to implement the *eshell*, which supports the shell operations of process execution, pipeline, sequential execution and subshell. In addition to these operations, parallel execution should also be supported. The eshell should take its inputs line by line like a regular shell and wait for the termination of every process that is being executed before accepting any new commands. Before any input, it should print a simple prompt followed by a single space in the following format:

```
/>
```

The general format for executing individual commands is given below:

```
/> command [args]*
```

The square brackets `[ ]` indicate that the inside is optional and asterisks to show there can be multiple.
The eshell should use the same shell symbol for the pipeline.
Format for pipeline:

```
/> {command or subshell} | [{command or subshell} |]*
{command or subshell}
```

Please note that the middle optional part can be repeated as many times as needed, which is why it is also marked with an asterisk. Examples:

```
/> ls -l | tr /a-z/ /A-Z/
/> cat input.txt | grep "log" | wc -c
```

The eshell should use the same shell symbols for sequential and subshell operations. The format for sequential execution:

```
/> {command or pipeline} ;[{command or pipeline} ;]* {command or pipeline}
```

Examples:

```
/> ls -l | tr /a-z/ /A-Z/ ; echo "Done."
/> cat input.txt | grep "log" | wc -c ; ls -al /dev
```

The parallel operation should use comma , for separation. The format for parallel execution:

```
/> {command or pipeline} ,[{command or pipeline} ,]* {command or pipeline}
```

Examples:

```
/> ls -l | tr /a-z/ /A-Z/ , echo "Done."
/> cat input.txt | grep "log" | wc -c , ls -al /dev
```

The format for subshell operations:

```
/> ({command or pipeline} {; or ,}[{command or pipeline} {; or ,}]* {command or pipeline})
```

Although the regular shell supports it, subshells cannot be executed sequentially or in parallel, and they cannot be nested. They can either contain sequential or parallel operations, but not both at the same time. Examples:

```
/> (ls -l | tr /a-z/ /A-Z/ ; echo "Done.")
/> (ls -l | tr /a-z/ /A-Z/ , echo "Done.")
/> (ls -l | tr /a-z/ /A-Z/ , echo "Done.") | (cat ; echo "Hello"; cat input.txt) |
cat | (wc -c , wc -l)
/> (cat input.txt | grep "c") | (tr /a-z/ /A-Z/ ; ls -al /dev) |
(cat | wc -l , cat , grep "A")
```

Finally, the command `quit` should terminate the program.
Example:

```
/> quit
```

3.1 Execution

If there is a single command, you need to fork a child process (see fork) and execute the command (see exec family of functions). Afterwards, reap the child process (see wait family of functions).

Otherwise, you need to consider four main scenarios. Moreover, they can be mixed when necessary. They are as follows:

- Pipeline Execution
- Sequential Execution
- Parallel Execution
- Subshell Execution

3.1.1 Pipeline Execution

Commands or subshells can be in a pipeline. They will be called CS in short for this section. The steps necessary to implement a pipeline are given below:

1. Assuming there are N number of CS in a pipeline, create $N-1$ pipes (see pipe)
2. Fork the CS processes (see fork)
3. Redirect their output and input to the correct pipes (see dup2)
4. Execute the CS (for commands: see exec family of functions/ see 3.1.4 for subshells)
5. Reap all of the CS that have finished executing(see wait family of functions) to prevent zombie processes. This also assumes subshell does the same for all its children (see Subshell Execution).

A basic pseudocode example is given below:

```
A | B | C
for (comm in A,B,C) {
    pipe_i = pipe()
        fork
           dup(pipe_i-1, 0)  (if i>0)
           dup(pipe_i, 1)    (if i<n)
           exec(comm)
}
for (process in A,B,C)
    wait  (after forks complete)
```

We can also see the timeline from the following graph1.

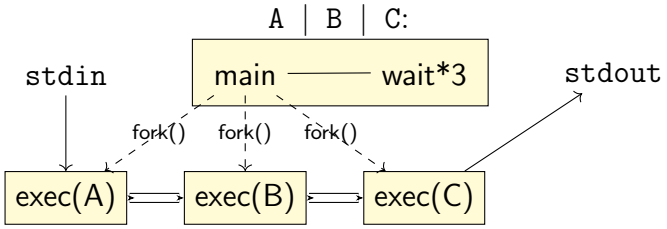


Figure 1: Pipeline Execution Example

3.1.2 Sequential Execution

Pipelines or commands can be executed in sequential order. They will be called PC in short for this section. The steps necessary to implement a sequential execution are given below:

1. For each PC in the list:
   (a) If it is a pipeline, execute the pipeline (see Pipeline Execution)
   (b) Else: Fork a process (see fork)
   (c) Execute the command (for commands: see exec family of functions)
   (d) Reap the finished process (see wait family of functions).

A basic pseudocode example is given below:

```
A ; B ; C
for (comm in A,B,C) {
    fork (child)
       exec(comm)
    wait() (Parent)  (synchronously, wait inside the loop)
}
```

We can also see the timeline from the following graph2.

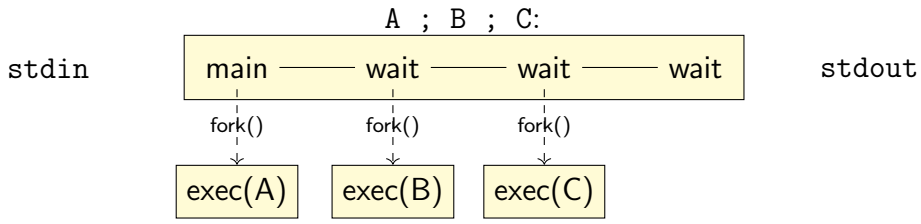


Figure 2: Sequential Execution Example

3.1.3 Parallel Execution

Pipelines or commands can be executed in parallel. They will be called PC in short for this section. The steps necessary to implement a sequential execution are given below:

1. For each PC in the list:
   (a) If it is a pipeline, execute the pipeline (see Pipeline Execution). Do not wait for the pipeline to finish.
   (b) Else: Fork a process (see fork)
   (c) Execute the command (for commands: see exec family of functions)
2. Reap all of the finished processes (see wait family of functions). This also includes the pipelines.

A basic pseudocode example is given below:

```
A , B , C
for (comm in A,B,C) {
    fork (child)
       exec(comm)
```

```
}
wait() (Parent)  (All of the children, wait outside of the loop)
```

We can also see the timeline from the following graph3.

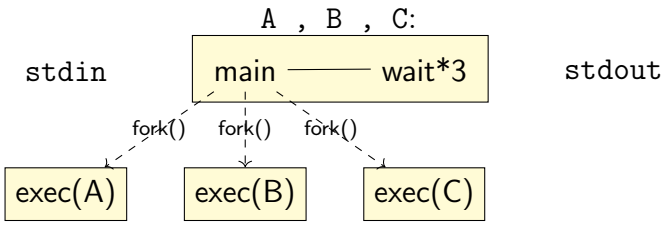


Figure 3: Parallel Execution Example

3.1.4 Subshell Execution

Parallel or sequential execution can be inside a subshell. The steps necessary to implement a subshell execution are given below:

1. Fork a process for the subshell (see fork).
2. If it just contains a single command or pipeline, execute it (see Pipeline Execution).
3. If it is a sequential execution, complete it (see Sequential Execution).
4. Else (it is a parallel execution):
5. Create N number of pipes for all pipelines or commands (PC) inside the subshell.
6. For each PC in the list:
   (a) Redirect the input of the command or the first command in the pipeline to one of the pipes created in step 4.
   (b) If it is a pipeline, execute the pipeline (see Pipeline Execution). Do not wait for the pipeline to finish.
   (c) Else: Fork a process (see fork)
   (d) Execute the command (for commands: see exec family of functions/ see 3.1.4 for subshell)
7. Fork a process for the repeater.
8. Repeater should work like the following:
   (a) While until EOF is received from standard input or pipes are no longer open on the other side:
   (b) Read stdin
   (c) Write the read data to every pipe
   (d) When EOF is received, close the pipes
9. Reap all of the finished processes (see wait family of functions). This also includes the pipelines.

Two basic pseudocode examples are given below:

```
(A ; B; C)
fork()
    call A;B;C handling code
wait (in subshell parent)

(A, B, C)
fork()
    for (comm in A,B,C)
       pipe_i = pipe()
       fork()
           dup(pipe_i, 0)
           exec(comm)
    fork()
        repeater code,
        for each line of stdin
           replicate to pipe_*[1]
for (process in A,B,C,repeater):
   wait  (after all forks finish, subshell parent)
```

We can also see the timeline from the following graphs45.

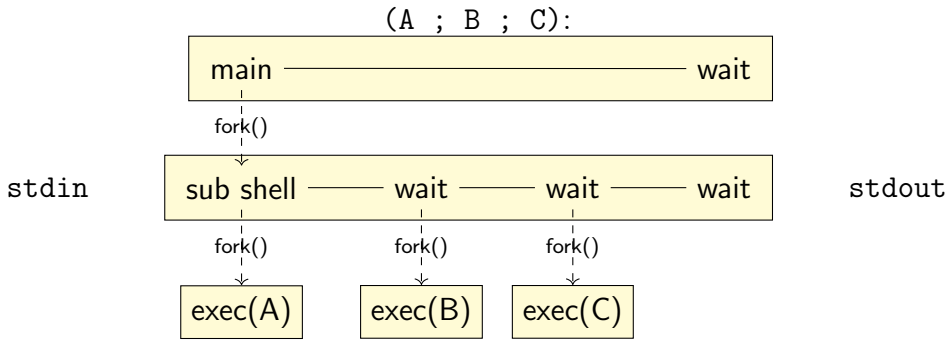


Figure 4: Subshell Sequential Execution Example

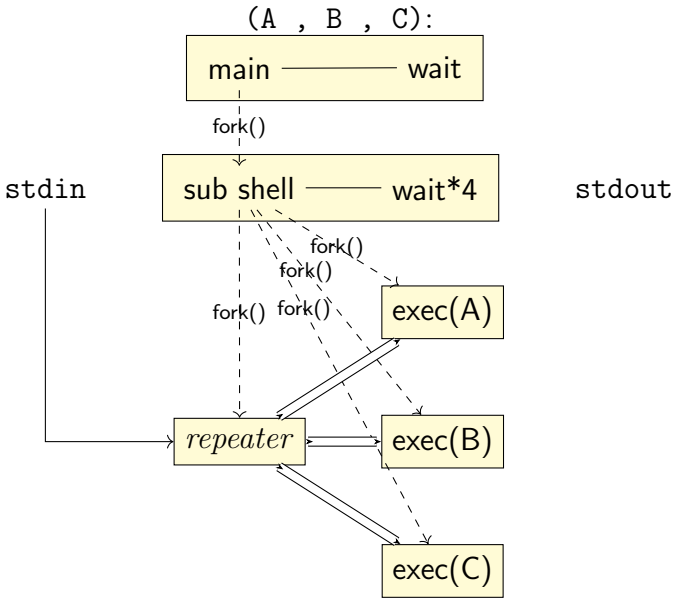


Figure 5: Subshell Parallel Execution Example

4 Input

- We will provide a parser for you. It will take a line of input and return the necessary information as an object. Details on how it works will be provided as comments. The only information it will need is one line of command.
- Commands are given in a single line.
- Executable names may be given as full path or not. The eshell must be able to execute both.
- All inputs will be proper. Non-existing executables will not be given. The `quit` command will not be used as an executable inside a subshell, pipeline, sequential, or parallel execution.

4.1 Parser

The parser will fill a struct name `parsed_input`. It will contain all of the information necessary to execute your commands. Here is the structure of that data:

```
typedef enum {
    INPUT_TYPE_NON, INPUT_TYPE_SUBSHELL, INPUT_TYPE_COMMAND, INPUT_TYPE_PIPELINE
} SINGLE_INPUT_TYPE;
typedef enum {
    SEPARATOR_NONE, SEPARATOR_PIPE, SEPARATOR_SEQ, SEPARATOR_PARA
} SEPARATOR;

typedef struct {
    char *args[MAX_ARGS]; // Null-terminated arguments
} command;

typedef struct {
    command commands[MAX_INPUTS]; // Array of commands
    int num_commands;
} pipeline;

typedef union {
    char subshell[INPUT_BUFFER_SIZE]; // Entire subshell string
    command cmd;                      // Single command
    pipeline pline;                   // Pipeline of commands
} single_input_union;

typedef struct {
    SINGLE_INPUT_TYPE type; // Type of the inputs
    single_input_union data; // Actual input which is union.
} single_input;

typedef struct {
    single_input inputs[MAX_INPUTS]; // Array of inputs
    SEPARATOR separator; // Separators for the input
    int num_inputs; // Number of inputs
} parsed_input;
```

The function that parses the input is called `parse_line`. It takes a line of input and a `parsed_input` pointer to fill the result. It will return false if it is a bad or unexpected input. There are also `free_parsed_input` and `pretty_print` functions to clean allocated commands inside the `parsed_input` and nicely print the parsed structure for checking purposes. Their prototypes are given below:

```
/***
 * Parses one input line and fills the parsed_input struct given as a pointer.
 * It can handle any number of spaces between arguments and separators.
 * It has support for single or double-quoted commands and arguments.
 * It returns 1 if it is a valid input and 0 otherwise.
 * @param line
 * @param input
 * @return
 */
int parse_line(char *line, parsed_input *input);

/***
 * Frees the allocated characters inside the inputs to prevent memory leaks.
 * It is recommended that you use this function after executing the commands inside the parsed
 * @param input
 */
void free_parsed_input(parsed_input *input);

/***
 * Prints the contents of the parsed_input struct nicely for checking.
 * You should look at how different inputs are stored to understand how parse_line works.
 * Please do not forget to delete this before submission to prevent unnecessary output from be
 * @param input
 */
void pretty_print(parsed_input *input);
```

These are important points for the `parsed_input`:

- There can only be one type of separator per line. There cannot be mixed pipe, semicolon or comma separators. That is why `parsed_input` has only one separator variable.
- If the separator is a pipe, the inputs can only be subshells or commands indicated by the input type.
- If the separator is a parallel or sequential separator, the inputs can only be pipelines or commands. Subshells cannot be chained with semicolon or comma.
- Subshell are recorded as complete strings. This means that after creating a child shell, it should be parsed again to be executed. Since the `parse_line` function has no idea whether it is parsing a normal line or a subshell, it will support nested subshell. However, you are not required to execute such commands, and no such case will be graded.
- Please do not modify the parser files, as they will be overwritten. This is because the constants can change during grading.

5 Specifications

- The eshell must terminate when the user gives the *quit* command. Otherwise, your homework will not be graded.
- All processes in a parallel execution or pipeline that are being executed should run in parallel. Therefore, the order of the outputs will be non-deterministic. In other words, executing the same commands may generate different outputs. Output order will not be important during evaluation.
- The eshell should reap all its child processes. It should not leave any zombie processes. This will lose points for each test case it happens in.
- When a subshell, sequential or parallel execution is happening, the eshell must wait for the termination of all the child processes before accepting new commands.
- If output is not redirected to a file, to a command or a subshell (via pipeline), it should use stdout as its output. Similarly, if its input is not redirected to a file to another command or subshell (via pipeline), it should use stdin as its input.
- Evaluation will be done using the black box technique. So, your programs must not print any unnecessary characters, and the outputs of the processes must not be modified. Grades are broken down(tentatively) according to this rubric:

  1. Single Command: 5 pts
     ```
     /> A
     ```
  2. Single Pipeline: 10 pts
     ```
     /> A | B | C
     ```
  3. Sequential Execution with commands: 5 pts
     ```
     /> A ; B ; C
     ```
  4. Sequential Execution with mixed commands and pipelines: 15 pts
     ```
     /> A | B ; C | D | E ; F
     ```
  5. Parallel Execution with commands: 5 pts
     ```
     /> A , B , C
     ```
  6. Parallel Execution with mixed commands and pipelines: 15 pts
     ```
     /> A | B , C | D | E , F
     ```
  7. Subshell with a single pipeline or sequential or parallel execution: 5 pts
     ```
     /> ( A | B | C | D )
     /> ( A ; B ; C ; D )
     /> ( A , B , C , D )
     ```
  8. Subshell with mixed types(sequential-pipeline or parallel pipeline): 10 pts
     ```
     /> ( A | B ; C | D | E ; F ; G | H )
     /> ( A | B , C | D | E , F , G | H )
     ```
  9. Subshell pipeline with mixed types(sequential-pipeline or parallel pipeline) without the need for repeaters: 15 pts
     ```
     /> ( A | B , C ) | ( D | E ) | ( F ; G ; H | I )
     ```
  10. Subshell pipeline with mixed types(sequential-pipeline or parallel pipeline) with repeaters necessary: 15 pts
      ```
      /> ( A | B , C ) | ( D | E , F ) | ( G , H | I ) | J
      ```

6 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your work. Any work used from third-party sources will be considered cheating and disciplinary action will be taken under the "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates daily.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

7 Submission

Submission will be done via ODTUClass. Create a tar.gz file named `hw1.tar.gz` that contains all your source code files with a makefile. The tar file should not contain any directories! The make should create an executable called `eshell`. Your code should be able to be executed using the following command sequence.

```
$ tar -xf hw1.tar.gz
$ make
$ ./eshell
```