VietNam National University Ho Chi Minh City
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Operating System Course
Lab 1 Full Report

CC07

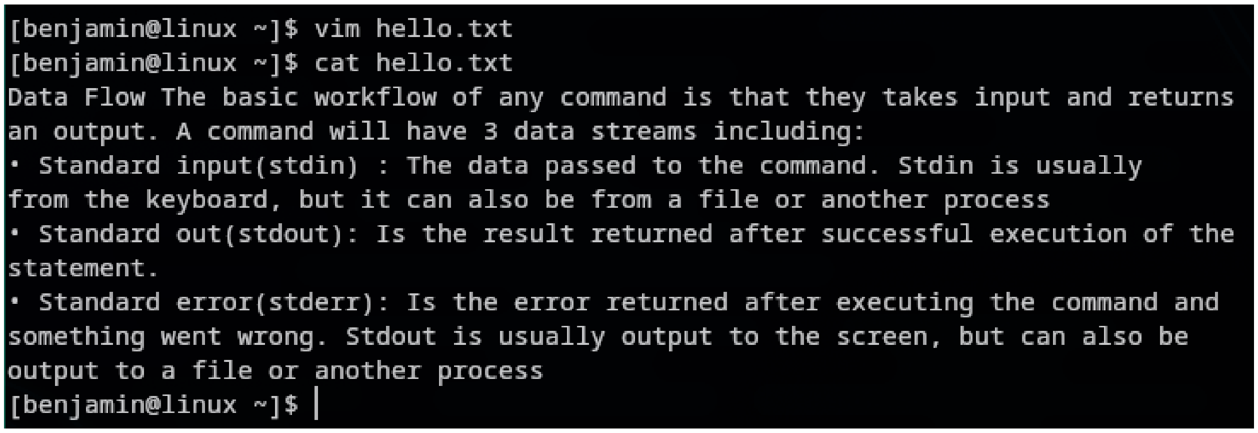
Student Name	**Student ID**
Tran Cong Hoang Phuoc	2352966

Contents

1 In-class Exercises

1.1 Exercise 1

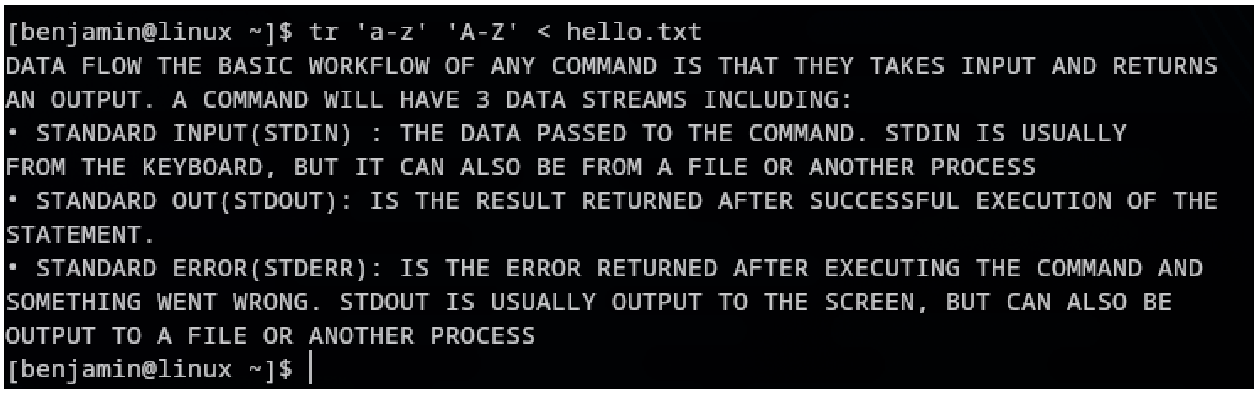
Use vim/vi to edit document hello.txt:



```
[benjamin@linux ~]$ vim hello.txt
[benjamin@linux ~]$ cat hello.txt
Data Flow The basic workflow of any command is that they takes input and returns
an output. A command will have 3 data streams including:
• Standard input(stdin) : The data passed to the command. Stdin is usually
from the keyboard, but it can also be from a file or another process
• Standard out(stdout): Is the result returned after successful execution of the
statement.
• Standard error(stderr): Is the error returned after executing the command and
something went wrong. Stdout is usually output to the screen, but can also be
output to a file or another process
[benjamin@linux ~]$ |
```

1.2 Exercise 2

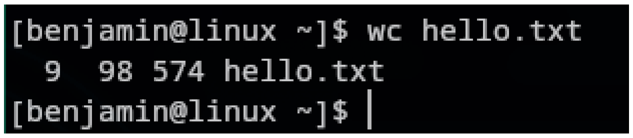
Convert all characters of hello.txt into uppercase:



```
[benjamin@linux ~]$ tr 'a-z' 'A-Z' < hello.txt
DATA FLOW THE BASIC WORKFLOW OF ANY COMMAND IS THAT THEY TAKES INPUT AND RETURNS
AN OUTPUT. A COMMAND WILL HAVE 3 DATA STREAMS INCLUDING:
• STANDARD INPUT(STDIN) : THE DATA PASSED TO THE COMMAND. STDIN IS USUALLY
FROM THE KEYBOARD, BUT IT CAN ALSO BE FROM A FILE OR ANOTHER PROCESS
• STANDARD OUT(STDOUT): IS THE RESULT RETURNED AFTER SUCCESSFUL EXECUTION OF THE
STATEMENT.
• STANDARD ERROR(STDERR): IS THE ERROR RETURNED AFTER EXECUTING THE COMMAND AND
SOMETHING WENT WRONG. STDOUT IS USUALLY OUTPUT TO THE SCREEN, BUT CAN ALSO BE
OUTPUT TO A FILE OR ANOTHER PROCESS
[benjamin@linux ~]$ |
```

1.3 Exercise 3

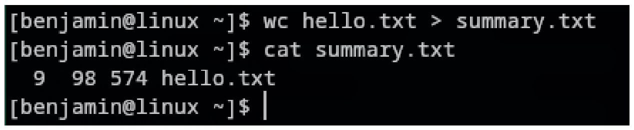
Count the number of words, lines, and characters of hello.txt:



```
[benjamin@linux ~]$ wc hello.txt
  9  98 574 hello.txt
[benjamin@linux ~]$ |
```

1.4 Exercise 4

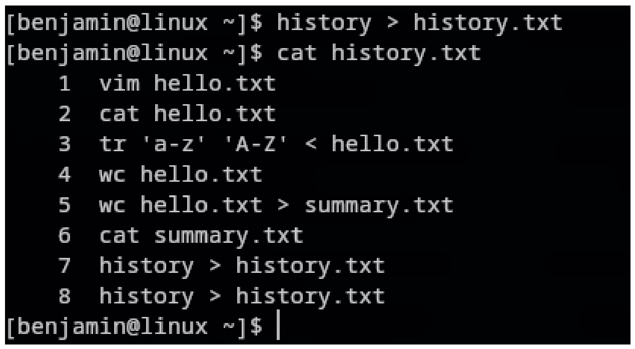
Output the ouput of **Exercise 3** to file summary.txt:



```
[benjamin@linux ~]$ wc hello.txt > summary.txt
[benjamin@linux ~]$ cat summary.txt
  9  98 574 hello.txt
[benjamin@linux ~]$ 
```

1.5 Exercise 5

List all recent commands into file history.txt:



```
[benjamin@linux ~]$ history > history.txt
[benjamin@linux ~]$ cat history.txt
    1  vim hello.txt
    2  cat hello.txt
    3  tr 'a-z' 'A-Z' < hello.txt
    4  wc hello.txt
    5  wc hello.txt > summary.txt
    6  cat summary.txt
    7  history > history.txt
    8  history > history.txt
[benjamin@linux ~]$ 
```

2 Q&A

2.1 Question 1

Some other popular Linux shells and their highlighted features:
Bash:

- Default shell on many Linux distributions.
- Command history and auto-completion for efficiency.
- Scripting capabilities with support for functions, loops, and conditionals.

Zsh:

- Powerful auto-completion and correction features.
- Plugin and theme support via frameworks like Oh My Zsh.
- Shared command history across sessions.

Fish:

- User-friendly syntax, eliminating the need for many complex Bash constructs.
- Autosuggestions based on command history.
- Color-coded syntax highlighting.

2.2 Question 2

Comparasion of the Output Redirection (>/>>) with the Piping (|) technique:

Feature	**Output Redirection (> / >>)**	**Piping (|)**
Purpose	Redirects command output to a file (stores data permanently)	Passes the output of one command directly as input to another (processes data temporarily)
Symbol	> for overwrite; >> for append	|
File Interaction	Writes output to a file, creating or modifying files	Does not create files; data is transferred between processes
Usage	Saving logs, reports, or command results	Chaining commands for filtering or transforming data
Example	`ls > files.txt` (overwrite) or `ls >> files.txt` (append)	`ls | grep ".txt"` (filters listing for `.txt` files)
Behavior	Output is stored persistently in a file	Output is passed directly between commands
Common Use Cases	Logging, output archiving, storing command results	Data processing, filtering, command chaining
Performance	Typically involves a single process writing to disk	May involve multiple processes; performance depends on the complexity of the pipeline

2.3 Question 3

Comparison of the sudo and the su command:

Feature	**sudo**	**su**
Purpose	Executes a single command or a series of commands with elevated privileges without switching the current shell session	Switches to the root (or another) user account, starting a new shell session
Usage	`sudo <command>`	`su -` (to become root) or `su <username>`
Authentication	Prompts for the invoking user's password	Prompts for the target user's (often root's) password
Security	Logs each command for auditing purposes; provides fine-grained control via the `/etc/sudoers` file	Does not log individual commands; offers full access once switched
Granularity	Grants temporary elevated privileges for specific commands	Can provides complete root privileges during the session
Session Handling	Maintains the user's environment while executing commands with elevated rights	Initiates a new session with the target user's environment
Best For	Running individual administrative commands or tasks that require limited root access	Performing multiple administrative tasks in a persistent root shell

2.4 Question 4

Discussion about the 777 permission on critical services (web hostings, sensitive databases,...):
Setting file or directory permissions to 777 means that anyone on the system can read, write, and execute that file or directory. While this might seem convenient in some cases, using 777 permissions on critical services—such as web hosting environments or sensitive databases—introduces several significant security risks:

- **Unrestricted Access:** Unauthorized users or compromised processes can change content, leading to website defacement, data corruption, or even system breaches.
- **Security Vulnerabilities:** Critical services like web servers and databases often contain sensitive information. Overly permissive access can make these services an attractive target for attackers looking to exploit vulnerabilities to compromise the entire system.

Best practices include:

- Only grant the minimum permissions necessary for a service or user to function. For example, web servers typically only need read (and sometimes write) access.
- Instead of making a file universally accessible, assign proper ownership and group rights that reflect the roles of different users and services.
- Continuously review and update permissions to ensure they are as restrictive as possible while still allowing necessary functionality.

2.5 Question 5

What are the advantages of Makefile? Give examples?

The advantages of Makefile include:

- **Automation of Build Processes:** A Makefile automates the compilation and linking process, reducing the need to manually enter complex commands.
  Example: Instead of manually compiling multiple C files, a Makefile can define rules that compile and link files with a single command.

  ```
  all: myapp

  myapp: main.o utils.o
     gcc -o myapp main.o utils.o

  main.o: main.c
     gcc -c main.c

  utils.o: utils.c
     gcc -c utils.c

  clean:
     rm -f *.o myapp
  ```

- **Dependency Management:** Makefiles keep track of dependencies between source files, ensure that only the necessary components are rebuilt when a change is made.
  Example: In the above example, if just only utils.c is modified, make will only recompile utils.o and then relink myapp, saving time.

- **Modularity and Reusability:** Rules in a Makefile are modular, meaning you can define and reuse commands across multiple parts of your project.
  Example: You can define variables for common compiler flags.

  ```
  CC = gcc
  CFLAGS = -Wall -g

  all: myapp

  myapp: main.o utils.o
  $(CC) $(CFLAGS) -o myapp main.o utils.o

  main.o: main.c
  $(CC) $(CFLAGS) -c main.c

  utils.o: utils.c
  $(CC) $(CFLAGS) -c utils.c
  ```

Compiling a program in the first time usually takes a longer time in comparison with the next re-compiling. What is the reason?

The first compilation is slower due to two main reasons:

- **Initial Full Build:** On the first run, all source files are compiled into object files, and then linked to create the final executable. This process processes every file, even if many of them haven't changed.
- **Incremental Compilation:** Tools like Make detect which files have changed. If only a small subset of files are modified, only those files are recompiled, saving time by reusing previously built object files.

Is there any Makefile mechanism for other programming languages? If it has, give an example?

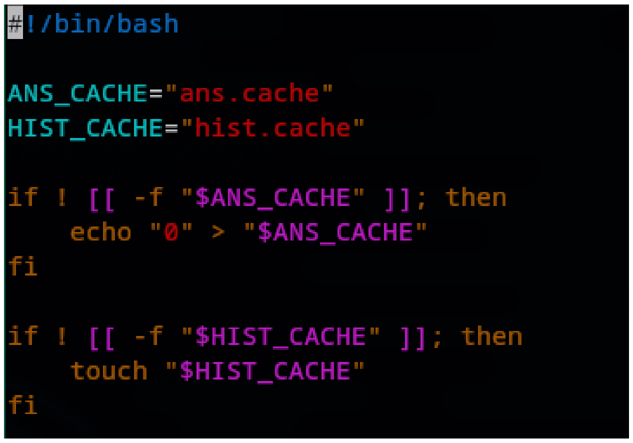
Makefiles are not limited to C/C++ projects - they can be used with virtually any programming language to automate tasks such as compilation, testing, packaging, or even deployment.
Example: This Makefile automates the compilation of a Java source file into class file and includes a clean target.

```
JAVAC = javac
JFLAGS = -g

all: myapp.class

myapp.class: myapp.java
   $(JAVAC) $(JFLAGS) myapp.java

clean:
   rm -f *.class
```

3 Practice Exercises 3.6

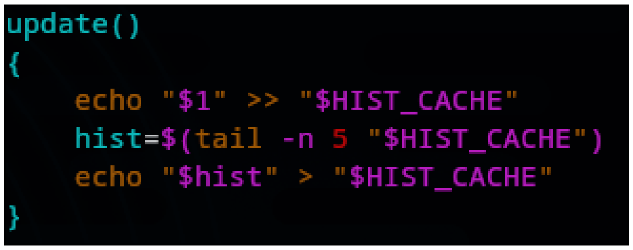
3.1 Initialize

Initiating caches for ANS and HIST:



```
#!/bin/bash

ANS_CACHE="ans.cache"
HIST_CACHE="hist.cache"

if ! [[ -f "$ANS_CACHE" ]]; then
    echo "0" > "$ANS_CACHE"
fi

if ! [[ -f "$HIST_CACHE" ]]; then
    touch "$HIST_CACHE"
fi
```

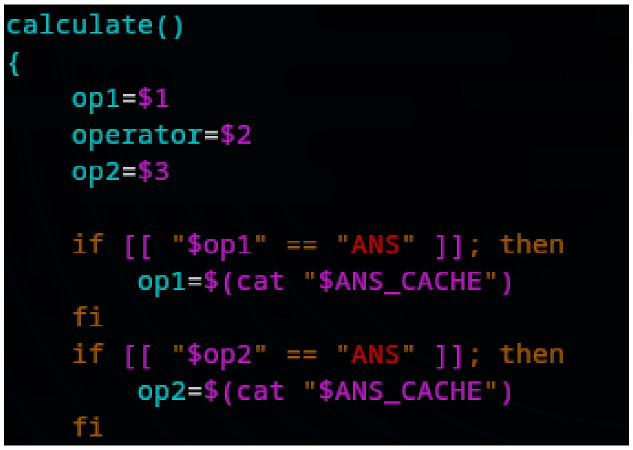
Update function to update the history cache:



```
update()
{
    echo "$1" >> "$HIST_CACHE"
    hist=$(tail -n 5 "$HIST_CACHE")
    echo "$hist" > "$HIST_CACHE"
}
```

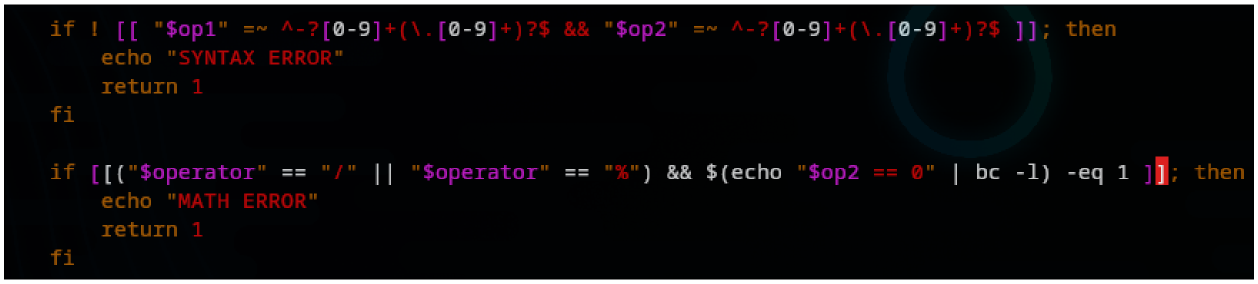
3.2 Calculate Function

Check if the operands are ANS:

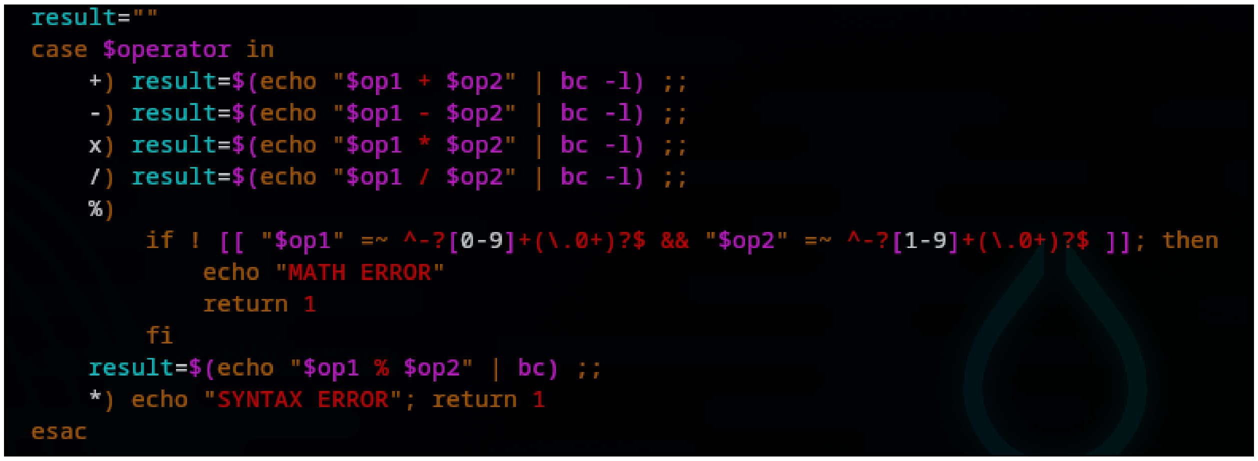


```
calculate()
{
    op1=$1
    operator=$2
    op2=$3

    if [[ "$op1" == "ANS" ]]; then
        op1=$(cat "$ANS_CACHE")
    fi
    if [[ "$op2" == "ANS" ]]; then
        op2=$(cat "$ANS_CACHE")
    fi
```

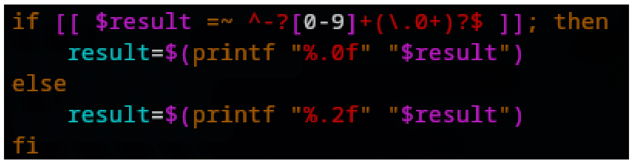
Check if there are any errors from input:


```
if ! [[ "$op1" =~ ^-?[0-9]+(\.[0-9]+)?$ && "$op2" =~ ^-?[0-9]+(\.[0-9]+)?$ ]]; then
    echo "SYNTAX ERROR"
    return 1
fi

if [[("$operator" == "/" || "$operator" == "%") && $(echo "$op2 == 0" | bc -l) -eq 1 ]]; then
    echo "MATH ERROR"
    return 1
fi
```

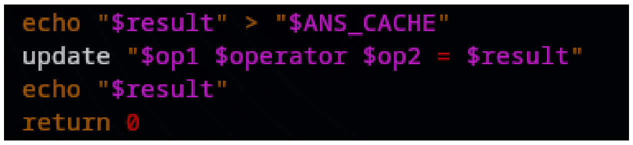
Doing calculation:


```
result=""
case $operator in
    +) result=$(echo "$op1 + $op2" | bc -l) ;;
    -) result=$(echo "$op1 - $op2" | bc -l) ;;
    x) result=$(echo "$op1 * $op2" | bc -l) ;;
    /) result=$(echo "$op1 / $op2" | bc -l) ;;
    %)
        if ! [[ "$op1" =~ ^-?[0-9]+(\.0+)?$ && "$op2" =~ ^-?[1-9]+(\.0+)?$ ]]; then
            echo "MATH ERROR"
            return 1
        fi
    result=$(echo "$op1 % $op2" | bc) ;;
    *) echo "SYNTAX ERROR"; return 1
esac
```

Format to 2 decimal digits:

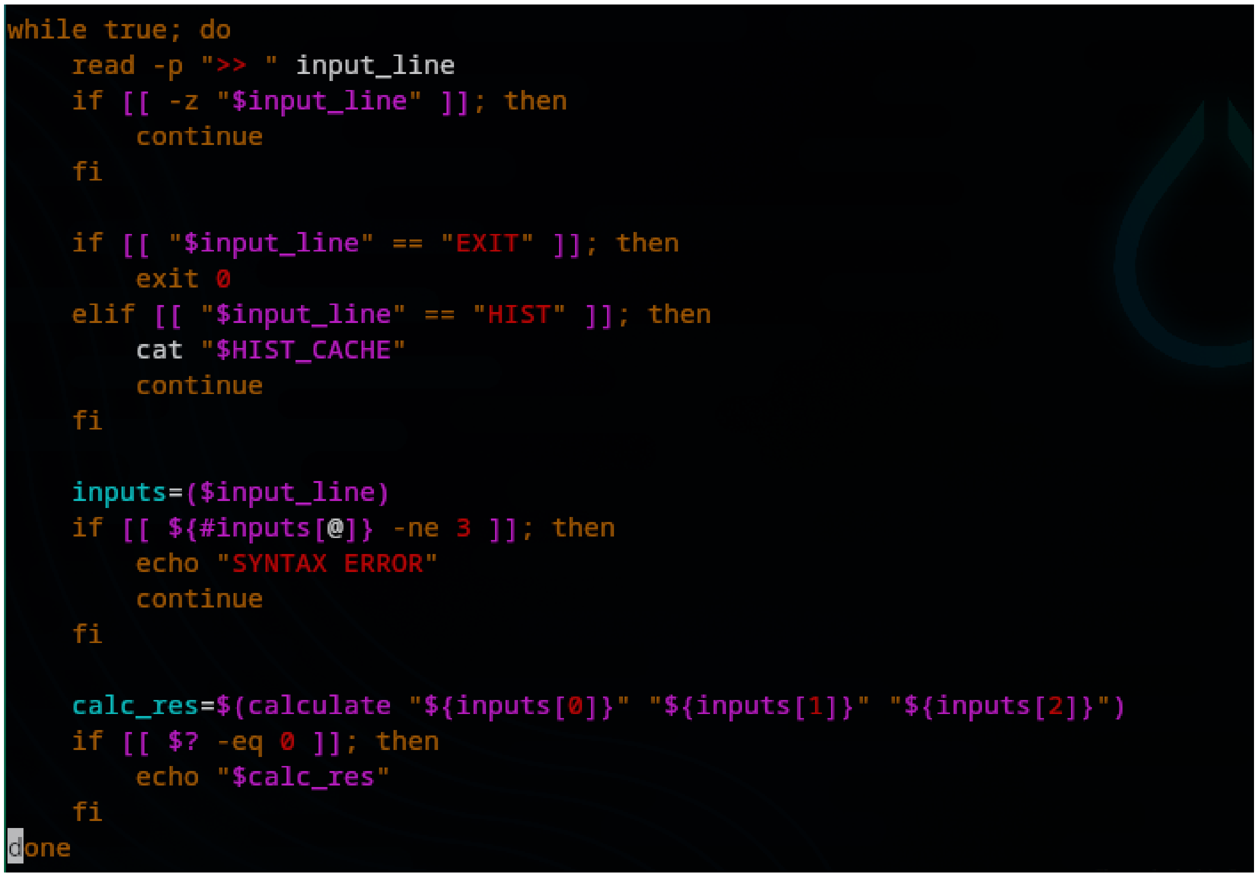

```
if [[ $result =~ ^-?[0-9]+(\.0+)?$ ]]; then
    result=$(printf "%.0f" "$result")
else
    result=$(printf "%.2f" "$result")
fi
```

Caching ANS and return result:


```
echo "$result" > "$ANS_CACHE"
update "$op1 $operator $op2 = $result"
echo "$result"
return 0
```

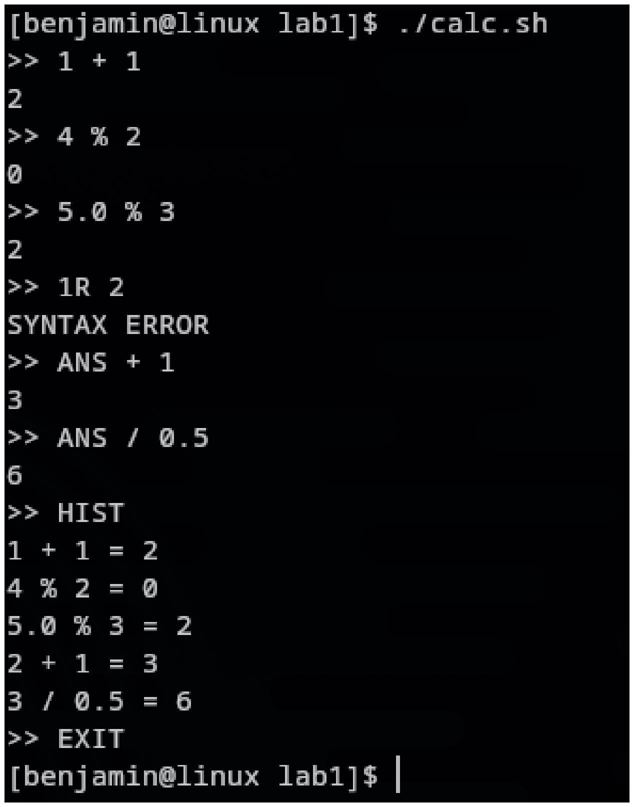
3.3 Main Loop

Reading input line and check if it is EXIT prompt or HIST prompt to do respective tasks.
Then if it is neither it must be an calculation, so seperate it into 2 operands and a operator and do the calculation by the function above.



```
while true; do
    read -p ">> " input_line
    if [[ -z "$input_line" ]]; then
        continue
    fi

    if [[ "$input_line" == "EXIT" ]]; then
        exit 0
    elif [[ "$input_line" == "HIST" ]]; then
        cat "$HIST_CACHE"
        continue
    fi

    inputs=($input_line)
    if [[ ${#inputs[@]} -ne 3 ]]; then
        echo "SYNTAX ERROR"
        continue
    fi

    calc_res=$(calculate "${inputs[0]}" "${inputs[1]}" "${inputs[2]}")
    if [[ $? -eq 0 ]]; then
        echo "$calc_res"
    fi
done
```

3.4 Testing

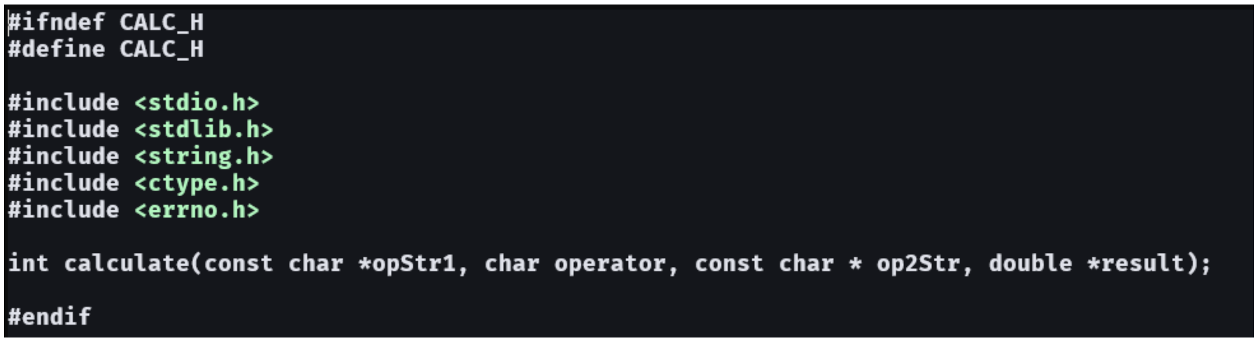


```
[benjamin@linux lab1]$ ./calc.sh
>> 1 + 1
2
>> 4 % 2
0
>> 5.0 % 3
2
>> 1R 2
SYNTAX ERROR
>> ANS + 1
3
>> ANS / 0.5
6
>> HIST
1 + 1 = 2
4 % 2 = 0
5.0 % 3 = 2
2 + 1 = 3
3 / 0.5 = 6
>> EXIT
[benjamin@linux lab1]$ |
```

4 Practice Exercise 5.3

4.1 Header File

Every neccesary libraries and the declaration of the calculating function is contained inside `calc.h`:

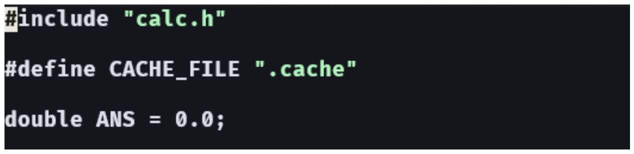


```c
#ifndef CALC_H
#define CALC_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>

int calculate(const char *opStr1, char operator, const char * op2Str, double *result);

#endif
```

4.2 Source Files

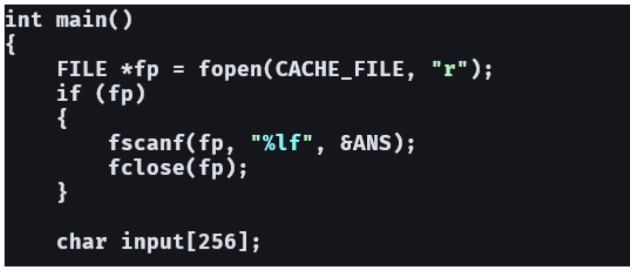
4.2.1 `calc.c`

Include the header file, define an alias for cache file, and initiate `ANS` value:



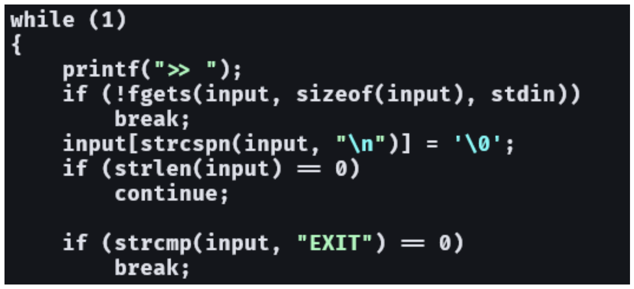
```c
#include "calc.h"

#define CACHE_FILE ".cache"

double ANS = 0.0;
```

In the `main()` function, open the cache file to read the `ANS` value and declare an input buffer:



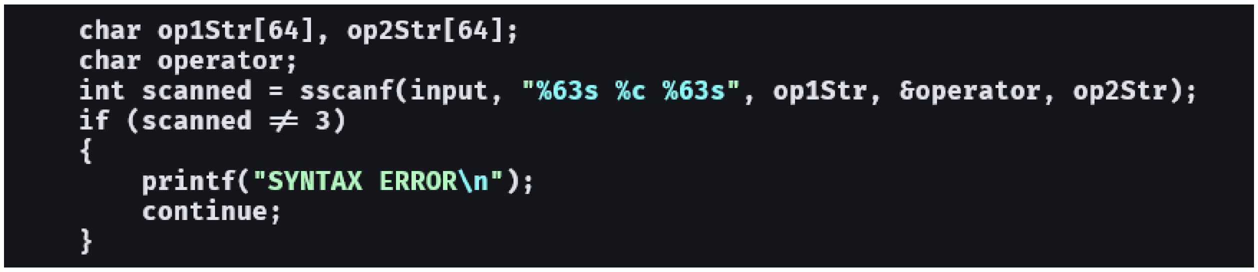
```c
int main()
{
    FILE *fp = fopen(CACHE_FILE, "r");
    if (fp)
    {
        fscanf(fp, "%lf", &ANS);
        fclose(fp);
    }

    char input[256];
```

In the main loop, print the >> at the begin of each iteration and read the input into the buffer. Then format the buffer to end with character `'\0'` and check to skip the current iteration if the buffer is empty or to exit the program if the buffer contains `"EXIT"`:



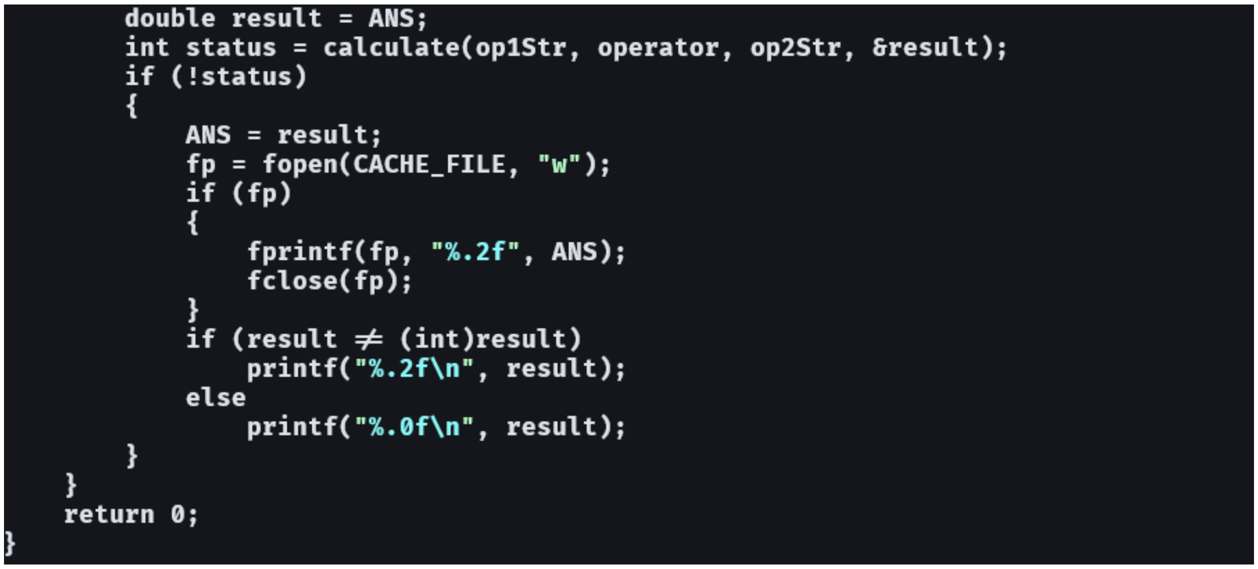
```c
while (1)
{
    printf(">> ");
    if (!fgets(input, sizeof(input), stdin))
        break;
    input[strcspn(input, "\n")] = '\0';
    if (strlen(input) == 0)
        continue;

    if (strcmp(input, "EXIT") == 0)
        break;
```

Separate the input buffer into 2 operand buffers and a operator character. If the input buffer can't be separated as said, throw a `SYNTAX ERROR` and skip the current iteration:



```c
    char op1Str[64], op2Str[64];
    char operator;
    int scanned = sscanf(input, "%63s %c %63s", op1Str, &operator, op2Str);
    if (scanned != 3)
    {
        printf("SYNTAX ERROR\n");
        continue;
    }
```

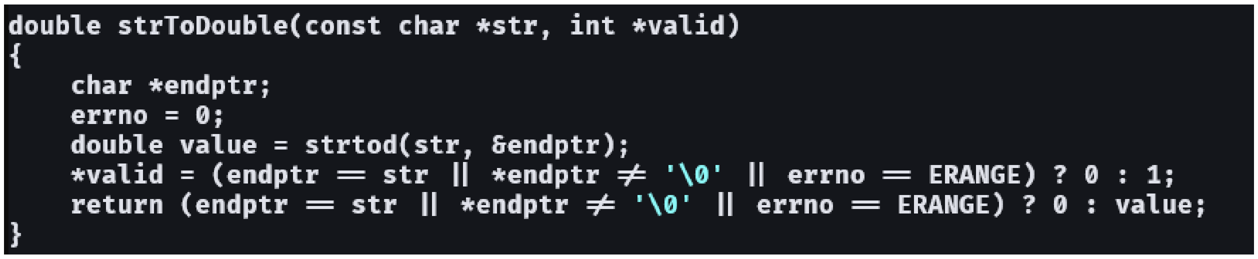
Declare a double `result` and temporarily store the `ANS` value in it, pass it by reference to `calculate()`, and call the function. The function will return 0 if there is no error so if it return 0, store the result in `ANS` to write it in the cache. Then format the result if it is not an integer and display:



```c
        double result = ANS;
        int status = calculate(op1Str, operator, op2Str, &result);
        if (!status)
        {
            ANS = result;
            fp = fopen(CACHE_FILE, "w");
            if (fp)
            {
                fprintf(fp, "%.2f", ANS);
                fclose(fp);
            }
            if (result != (int)result)
                printf("%.2f\n", result);
            else
                printf("%.0f\n", result);
        }
    }
    return 0;
}
```

4.2.2 logic.c

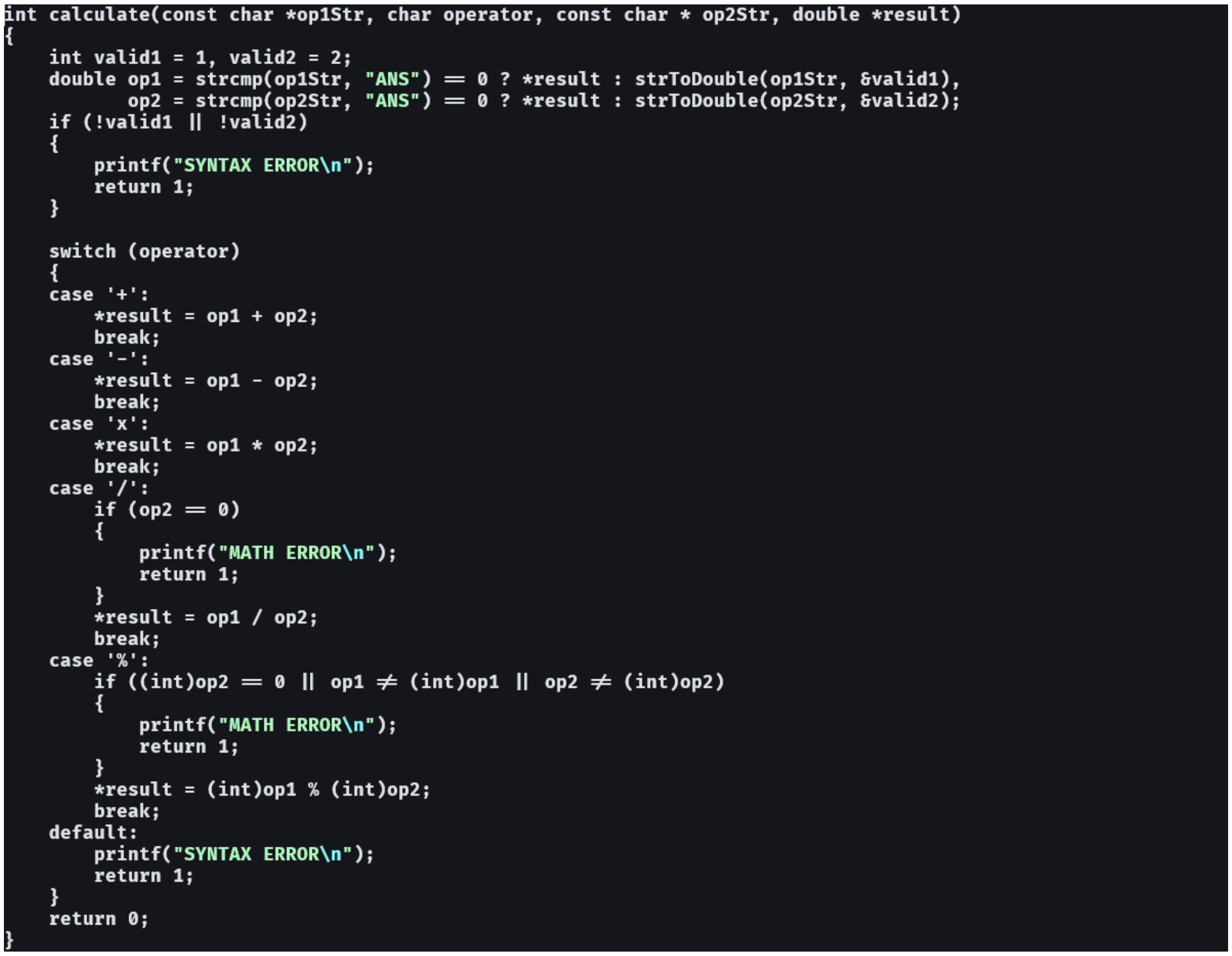
Implement a function `strToDouble()` to transform the operand buffers into double-type numbers with a `valid` variable to check if the conversion was successful:



```c
double strToDouble(const char *str, int *valid)
{
    char *endptr;
    errno = 0;
    double value = strtod(str, &endptr);
    *valid = (endptr == str || *endptr != '\0' || errno == ERANGE) ? 0 : 1;
    return (endptr == str || *endptr != '\0' || errno == ERANGE) ? 0 : value;
}
```

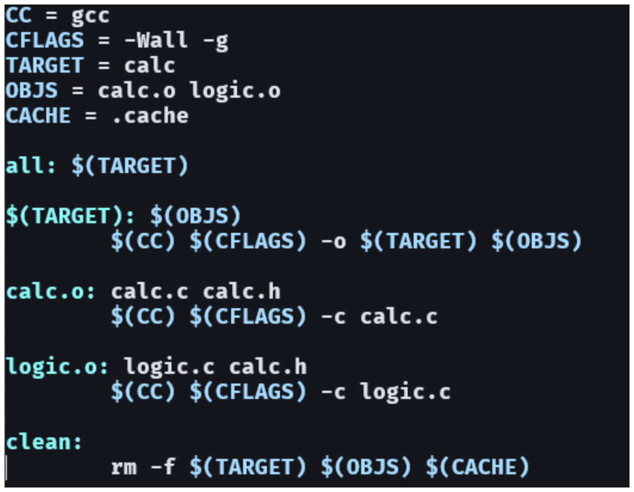
Implementatioon of the `calculate()` function:

1. Declare 2 double-type for the operands and assign them with the appropriate values base on theirs buffers. (As we temporarily stored `ANS` in `result`, we will use `*result` to get `ANS`'s value)
2. If both operand conversions weren't successful, throw a `SYNTAX ERROR` and return 1.
3. Perform calculation using switch-case. If the calculation produces an error, display it and return 1, if the calculation is successful, store the result and return 0.



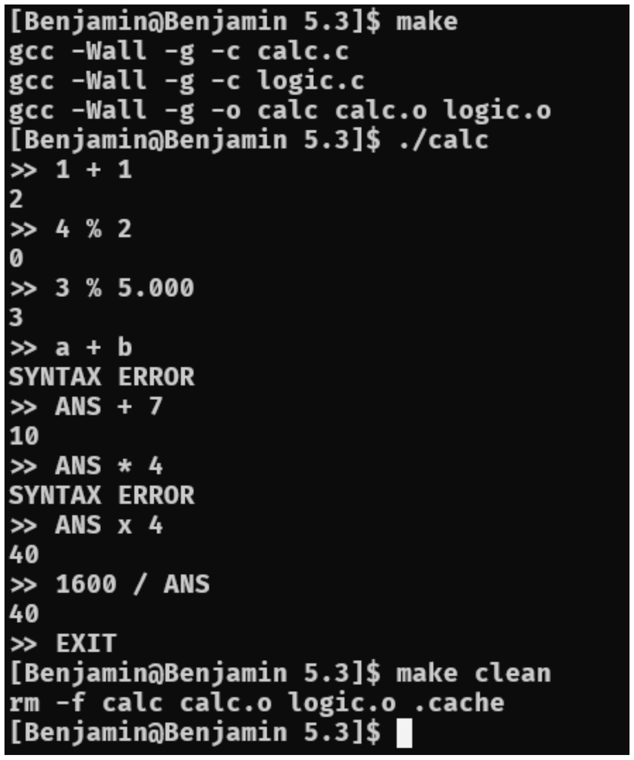
```c
int calculate(const char *op1Str, char operator, const char * op2Str, double *result)
{
    int valid1 = 1, valid2 = 2;
    double op1 = strcmp(op1Str, "ANS") == 0 ? *result : strToDouble(op1Str, &valid1),
           op2 = strcmp(op2Str, "ANS") == 0 ? *result : strToDouble(op2Str, &valid2);
    if (!valid1 || !valid2)
    {
        printf("SYNTAX ERROR\n");
        return 1;
    }

    switch (operator)
    {
    case '+':
        *result = op1 + op2;
        break;
    case '-':
        *result = op1 - op2;
        break;
    case 'x':
        *result = op1 * op2;
        break;
    case '/':
        if (op2 == 0)
        {
            printf("MATH ERROR\n");
            return 1;
        }
        *result = op1 / op2;
        break;
    case '%':
        if ((int)op2 == 0 || op1 != (int)op1 || op2 != (int)op2)
        {
            printf("MATH ERROR\n");
            return 1;
        }
        *result = (int)op1 % (int)op2;
        break;
    default:
        printf("SYNTAX ERROR\n");
        return 1;
    }
    return 0;
}
```

4.3 Makefile



```
CC = gcc
CFLAGS = -Wall -g
TARGET = calc
OBJS = calc.o logic.o
CACHE = .cache

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

calc.o: calc.c calc.h
	$(CC) $(CFLAGS) -c calc.c

logic.o: logic.c calc.h
	$(CC) $(CFLAGS) -c logic.c

clean:
	rm -f $(TARGET) $(OBJS) $(CACHE)
```

4.4 Testing



```
[Benjamin@Benjamin 5.3]$ make
gcc -Wall -g -c calc.c
gcc -Wall -g -c logic.c
gcc -Wall -g -o calc calc.o logic.o
[Benjamin@Benjamin 5.3]$ ./calc
>> 1 + 1
2
>> 4 % 2
0
>> 3 % 5.000
3
>> a + b
SYNTAX ERROR
>> ANS + 7
10
>> ANS * 4
SYNTAX ERROR
>> ANS x 4
40
>> 1600 / ANS
40
>> EXIT
[Benjamin@Benjamin 5.3]$ make clean
rm -f calc calc.o logic.o .cache
[Benjamin@Benjamin 5.3]$ 
```