Sabanci University

Operating Systems

CS 307

Programming Assignment - 1: Shell Simulation of the TreePipe Command

Release Date: 18 October 2024
Deadline: 28 October 2024 23.55

1 Problem Description

In the lectures, we have seen that shell (or Command Line Interface - CLI) is a regular user program that can be implemented using `fork`, `wait` and `exec` system calls. We analysed some sample C programs that simulate execution of `wc` (word count) command in the shell and how to perform file redirection operations. Moreover, we claimed that piped commands can be simulated using UNIX pipe system calls.

In this Programming Assignment (PA), we expect you to implement a C program that simulates a command called TREEPIPE .TREEPIPE is a new addition to the shell command palette and you will be the first programmers simulating this command in a C implementation. As the name implies, this command creates a full binary tree[1] of which nodes correspond to processes. Depending on whether a process is a left (first) or right (second) child of its parent, it executes one of two possible programs. Input of these programs might be coming from current process' descendants or its parent process. In addition, a process may forward the result of its own computation to its children or its parent. Hence, processes have to communicate with each other for streaming their input and output. You have to use UNIX pipes for these purpose which solves the mystery in the *pipe* part of TREEPIPE .

We will explain how TREEPIPE is supposed to work next by going over the example tree presented in Figure 1. As stated earlier, each TREEPIPE node (process) is supposed to execute one of two programs. Let us call them *left* and *right*. Both programs have the same input-output format: They take two integer inputs *num1* and *num2* and produce one integer output *res*. Considering the example in Figure 1, the *left* program sums two inputs ($res = num1 + num2$) and the *right* program multiplies two inputs ($res = num1 \times num2$).

Deciding on a process' program depends on its parent. If it is the first (left) child of its parent, it executes the *left* program. Similarly, the second child (right) executes *right* program. You can safely assume that the `root` process always executes *left*.

The most important point in understanding how TREEPIPE works is to understand how a process gets its input and how it forwards its output. Basically,a process P takes its input *num1* from its parent. Then, it passes

[1] A full binary tree is a rooted tree in which each node has either exactly two children or none.

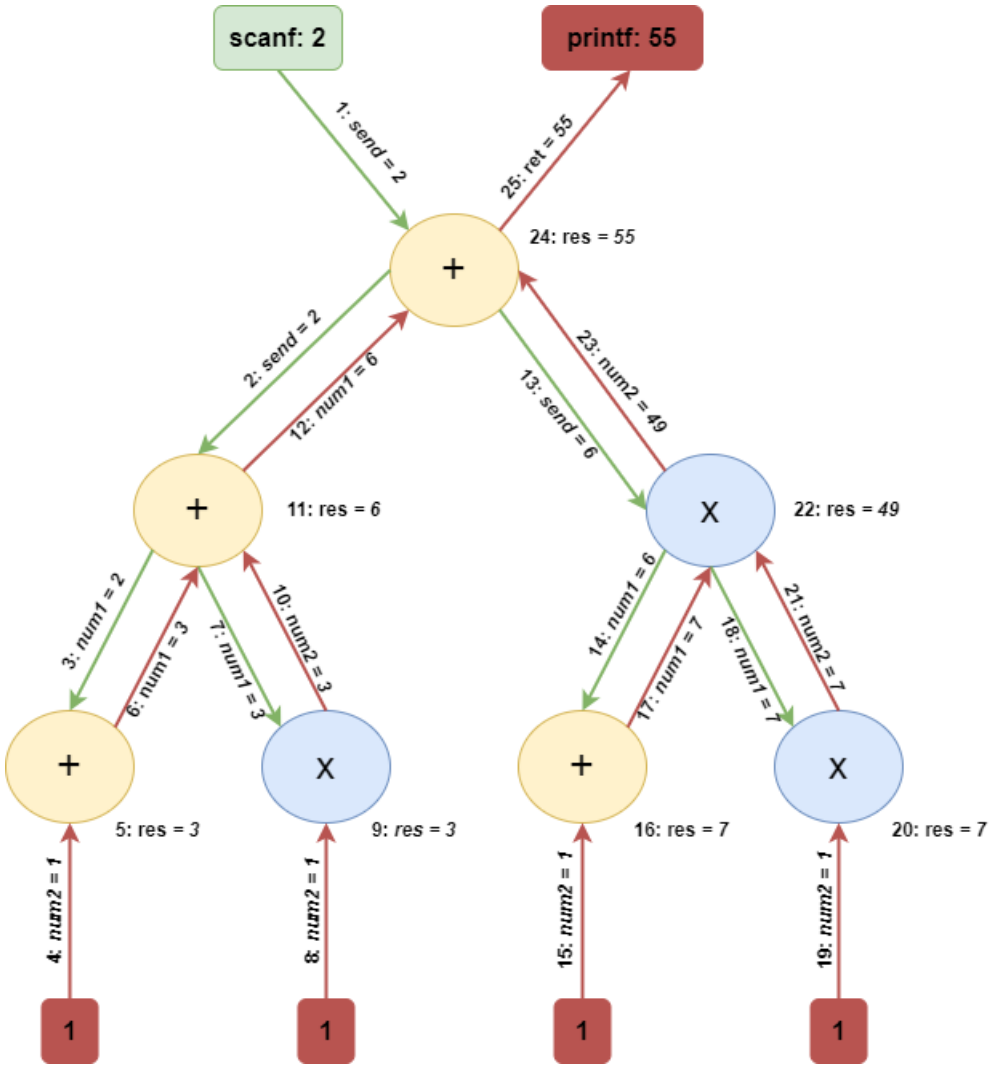

Figure 1: Sample TreePipe execution. Left children are colored with yellow and they perform addition whereas right children are colored blue and they execute multiplication operations. Leaf nodes' *num2*s are provided as default values from the red boxes which do not correspond to a process. The root takes its input from the console through `scanf` and prints its result to the console through `printf`. Data flow among processes are marked with arrows. Green arrows represent flow from parents to children and red arrows represent the reverse direction. Each edge is labeled with its execution order and the variable in target node that takes its value from.

num1 as an input to its left child and waits until its left sub-tree completes its computation.

In this tree's post-order traversal, each node initially passes num1 to its left child and waits until its entire left subtree has completed its computations. The result returned by the left child is accepted as num2 by node P, and it is also sent to the right child of P as num1. At this point, P is ready to run its program for the right subtree, following its parent's process flow. After processing its right subtree, where P passes the result of the left subtree as num1 to its right child, P waits until this right subtree completes its computation and returns a result to P. The result returned from the right child is then taken as another input by P for its own computation, alongside the result from the left child. If P is the root, then the value it computes last, which integrates results from both the left and right subtrees, is the final result of the TreePipe computation.

For a better understanding of the protocol, lets try to understand what the left child of the root node in Figure 1 does:

- Initially, the left child of the root receives a num1 value of 2 from the root, indicated by the green edge labeled 2. It forwards this value to its own left child, highlighted by the edge labeled 3.

- The leftmost child processes this num1 value of 2, performing its computation and resulting in a value of 3, as indicated by the red edge labeled 5. This result of 3 is then sent back up to its parent and also forwarded to its right sibling.

- The right sibling receives the value of 3 as num1 and gets num2 = 1 since this is a leaf node, conducts its own computation which results in a value of 3 (step 9), and then sends this result back up to their shared parent.

- The parent node, having received the computed results of 3 from both its left and right children, then performs its own computation. By combining these results (3 from the left and 3 from the right), it calculates a final output of 6, as shown by label 11 next to the parent node.

- This resultant value of 6 is then communicated upwards to the root node, marked by the right edge labeled 12, facilitating further integrative computations at the tree's higher levels.

- Additionally, this result of 6 is sent laterally to the right sibling node, the right child of the root, which uses it for its further computations. This forwarding action supports the flow and integration of data across the tree, ensuring comprehensive processing in accordance with the post-order traversal method.

There are two corner cases to consider for a successful TreePipe computation. The first one concerns the leaf nodes. Since leaves do not have any left child, their *num2* value is not determined. You can safely assume that for the leaf nodes, *num2* has the default value of 1.

The second corner case is about the root node. Since it does not have any parents, its *num1* is not determined. We expect that the `root` takes this value from the user via `scanf` method.

1.1 Post-Order Traversal

While implementing the TreePipe command, you can also mimic the *post-order traversal*, which is another systematic way to traverse a binary tree that you learn in data structures and algorithms. *Post-order traversal* is similarly defined recursively and locally on each node of the tree. When performing a post-order traversal from a node, you first visit all the nodes in the left sub-tree, followed by the nodes in the right sub-tree, and only after both subtrees have been traversed, do you visit the current node. This traversal method ensures that a node is not visited until all its children have been processed. Figure 2 shows the order of the visited nodes in post-order traversal of a tree:

2 C Implementation

In your current working directory, it is expected that there are three C source code files:

- **left.c:** This file contains a very simple main method that reads two integers with `scanf` method, performs some computation and then writes the result to the console via `printf` method. In the PA1 package, we will provide you a sample `left.c` files. You can write your own versions to test your TreePipe implementation. However, we are not expecting

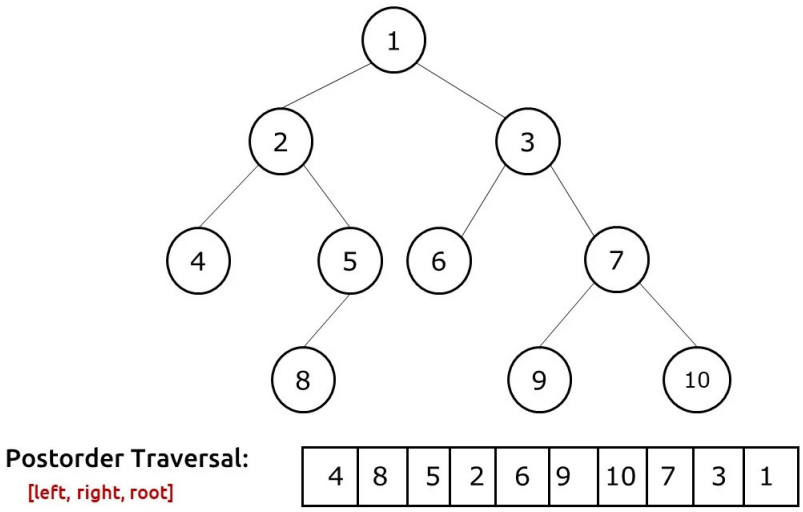

Figure 2: Post-order traversal of a binary tree.

you to develop or submit it. We expect your TreePipe implementation to work with any left program obeying the input-output format described above. For your TreePipe implementation, you should assume that the compiled binary executable version of `left.c` called `left` exists in the same directory with the TreePipe executable.

- **right.c:** It has the same structure (input-output) and requirements with `left.c`. Its computation might be different from or the same with `left.c`. You should also assume that its binary file called `right` exists in the same directory with your TreePipe executable.

- **treePipe.c:** This file will contain your TreePipe implementation and must be developed by you. It will take three arguments from the command line[2]: *curDepth*, *maxDepth* and *lr*. All three input arguments must be treated as integers. Meaning and usage of these integers are explained in the next section.

[2]See Section 2.4 for more on how to write a C program that takes input arguments from the command line.

2.1 TreePipe Program Inputs

Each node/process in the tree is expected to run the program implemented in **treePipe.c**. This might sound confusing since earlier we stated that each node runs either `left` or `right`. The explanation is that TreePipe program you implement will be an orchestrator that creates the left and right children processes of this node and also creates a worker process to run its own program (left or right) on her behalf.

As stated earlier, TreePipe program will take three input integers from the command line. The first one *curDepth*, is the depth of the current node. Root node is considered to have depth 0, its children have depth 1, its grandchildren have depth 2 and so on.

The second integer, *maxDepth* denotes the maximum depth of the execution tree. Nodes can use this parameter to detect whether they are leaf nodes or not. If $curDepth = maxDepth$, then this process understands that it is a leaf node and does not create any children.

The last parameter *lr* shows whether this program will execute *left* or *right* programs for its computation. This parameter is either 0 or 1. If it is 0, it executes the *left* program. Otherwise, it executes *right*.

Since each tree node is created by its parent, the parent can determine all three parameters correctly from its local information. When a parent with input parameters $curDepth$, $maxDepth$, lr creates a left child, its input parameters are set as $curDepth + 1$, $maxDepth$ and 0, respectively.

How to write a program that takes input from the command line is explained in Section 2.4. The next section explains the behaviour of the TreePipe program.

2.2 TreePipe Program

The first thing your TreePipe implementation does is to interpret its input arguments as integers. Since main's input array argv is a string array, three elements in it must be converted to the integer type. Then, this process must take an input number using a scanf statement. If this node is the root, then the user must provide number from the console. For other nodes, its parent manipulates the file descriptors such that instead of reading from the console, scanf gets its input from the pipe established with its parent.

The next goal of TreePipe program is to get its num1 value. If the node is a leaf node, then the input that was previously taken is num1, otherwise

it must get num1 from its left child. For this purpose, it has to create its left child and wait for all its left sub-tree to finish its computation. Hence, TreePipe program forks a child. The left child transforms itself into TreePipe program again using execvp system call and with appropriate input parameters whereas the parent blocks until the left sub-tree completes its computation. Moreover, there must be a pipe between the parent and the child so that the parent can send the input number that it took previously to the child through the write end of the pipe, the child's file descriptors are modified so that when the child executes scanf, it reads from the read end of the pipe instead of STDIN. Moreover, when the left child terminates, it prints the result of its tree computation through printf statement. Child's STDOUT file descriptor must be redirected to the write end of the pipe so that this result becomes available to the parent instead of getting printed to the console.

In the context of a post-order traversal with interprocess communication, the operation starts when the left child terminates and the parent reads the result, known as num1, from the pipe connecting to the left child. This result is crucial for the parent's subsequent actions. If the parent needs to transform into a target program using execvp, it cannot directly propagate its result upwards or create the right child. To handle this, the parent forks a new child dedicated to running the target program, establishing a pipe for communication. The child's STDIN and STDOUT are redirected to the respective ends of this pipe, allowing it to receive inputs and send outputs seamlessly. The parent then blocks to wait for this child to execute and complete the program, reading the results back from the pipe upon completion.

Following the completion of the left child's processing and the target program execution, the parent then proceeds to fork again to create the right child, setting up another pipe for this purpose. The result previously obtained from the left subtree is sent to the right child through this new pipe, ensuring the right child receives the necessary data for computation. The right child processes this data, reads its input via scanf, and sends its computational results back through the pipe using printf. Once the right child completes and terminates, the parent reads this result from the pipe as num2. Note that a leaf node uses a constant value 1 as its num2 since it has no children node. Following the post-order execution the node executes its own program after the right child using num1 and num2.

If the parent is not the root, it forwards this integrated result to its own parent via another pipe, completing the post-order traversal process.

This approach ensures that each node in the tree processes its left and right children's data before finalizing its own computation, adhering strictly to the post-order logic while effectively managing interprocess communication and program transformations.

2.3 Implementation Details & Corner Cases

- You can safely assume that data transferred through the pipes will not exceed 10 characters. Hence, when you want to read from a read end of a pipe, you can use a char array of size 11.
- The default *num*2 value for leaf nodes is 1.
- Since your program will be tested and graded automatically by another program, it is important that your program's output matches with the format of the sample output shown in Section 5. It means that all the nodes in the tree must print some lines to the console. However, all the nodes' except the root's file descriptors must be modified such that `STDOUT` is redirected to the write end of the pipe with its parent instead of the console. In this case, you are encouraged to use the standard error channel `STDERR` to print something to the console.
- In order to redirect `STDIN` or `STDOUT` to a pipe you are advised to use `dup2` system call. `dup2` lets you create a duplicate of an existing open file descriptor, allowing you to use the same file with two different names (descriptors) within your program. For more information you can check this link.
- In this programming assignment you are expected to use `execvp` to transform an existing process' program. In the lectures, we used `execvp` to transform a process into one of the existing Unix commands like `wc`. However, you can also use `execvp` to run your own executable files. For instance, if you have an executable file `p1` under the directory that your process runs, you can transform this process into `p1` by just setting the first argument of the `execvp` input array to `"./p1"`. For more information about exec functions and execvp you can refer this link.
- You are not allowed to use `system` or `sleep` system call for creating a new process or synchronizing processes. You must use `fork` for process

creation. You are not allowed to use any system calls for process manipulation except `fork`, `wait`, `execvp`, `pipe`, `dup2` and their variants.

- You might be tempted avoid using pipes and command line arguments to transfer input from the parent to the child because when the parent forks, *num1* and *curDepth* values will be duplicated on the child side as well. However, when the child transforms itself again by calling `execvp`, these variables are reinitialized and the previous values are lost. Hence, you need to use pipes and command line arguments.

- You might want to use the same pipe between multiple parent-child pairs. However, when several processes use the same pipe, concurrency problems become more difficult to address and synchronization becomes more difficult. Moreover, after calling `execvp`, processes forget previous pipes. Hence, it is strongly recommended that you create a separate pipe for each parent-child process pair.

- Even if you establish pipes correctly, there can be some incompatibility issues between `scanf`/`printf` procedures and `read`/`write` system calls. For instance, in order to send some integer from the parent to the child through a pipe, the parent might write some integer to a character array *buffer* using `sprintf` procedure and then it gives *buffer* as an input to the `write` system call to write to a pipe. On the other side, the child process might want to read this integer using `scanf`. In this case, the value read by `scanf` might be garbage due to buffer contents. To overcome this problem, you might use `dprintf` instead of `sprintf` and `write` system call. Detailed information on `dprintf` can be found here.

- If you cannot execute left or right programs, check permissions and change them using chmod if you do not have permissions, as such:

```
chmod u+x <filename>
```

2.4 Taking input arguments from the command line

In this PA, you are required to write a C program that takes inputs from the command line. To do this, you are suggested to use *argc* and *argv* parameters of your main function. Briefly, *argc* is an integer that stores the number of

command line arguments that are passed. *argv* is an array of character pointers. Each element of *argv* is an argument you passed, starting with *argv[0]* being the name of the program. To make it more clear, consider the following example:

Example 1: example1.c

```
#include <stdio.h>

// defining main with arguments
int main(int argc, char* argv[])
{
    printf("You have entered %d arguments:\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

example1.c is a C code that takes some arguments from the command line and prints out the number of arguments that are passed (including the name of the program), and each of these arguments. For example, after compiling example1.c with *gcc example1.c -o example1* command, if we execute example1 with the command: *./example1 hello cs307*, the output would be:
You have entered 3 arguments:
./example1
hello
cs307
Note that in the example code, ith argument is accessed with *argv[i]*.

3 Submission Guidelines

For this homework, you are expected to submit two files. The first one is a pdf that will contain your report. The name of this file must be *"report.pdf"*. The second file that you will submit is the code part of your assignment. Name of this file must be *"treePipe.c"* Below you can see the content of your files.

- treePipe.c: This file will contain your TreePipe implementation.

- report.pdf: Your report that explains your code briefly. Please note that all the reports are read carefully to understand your program in a better sense. In your report, you are expected to make the instructor understand such details as : What your program does, how your functions work (if there is any), what is the role of pipe on that function etc.

During the submission of this homework, you will see two different sections on SUCourse. For this assignment you are expected to submit your files seperately. You should NOT zip any of your files. While you are submitting your homework, please submit your report to "PA1 – REPORT Submission" and your code to "PA1 – CODE Submission". The files that you submit should NOT contain your name or your id. SUCourse will not except if your files are at another format, so please be careful. If your submission does not fit to the format specified above, your grade may be penalized up to 10 points.

4 Grading

Dear students, your first programming assignment will be graded using automated test cases. Please keep in mind that, even slight discrepancies on output will make you lose points.

- Compilation (10 pts): Your program compiles, runs and terminates without an error.
- Shell (CLI) Input (10 pts): Your program must get three integers as input when it is called from shell and use it while making calculations correctly. Please, note that if use `scanf` or other file operations to take *curDepth*, *maxDepth* and *lr* values, you will get 0 from this part.
- Depth Zero (10 pts): Your treePipe implementation works correctly for depth = 0 i.e., when there is single root node executing a left computation.
- Depth One (10 pts): Your treePipe implementation works correctly for depth = 1 i.e., when there are three nodes: root, left and right and the right computes the correct result.

- Correct System Calls (10 pts): Your program only uses a subset of `fork`, `dup`, `exec`, `open`, `pipe`, `read`, `write` and `close` system calls or its variants and it does not employ other system calls like `sleep` and `system`.

- Tree Printing Format (20 pts): Your tree output must **completely** have the same format with the sample runs (see Section 5) since the grading will mainly be done by automated test cases. By tree output, we mean the dashes and > sign printed at the beginning of each line that shows the depth of the which prints this line. For each depth level, you must print three dashes.

- Test Cases (20 pts): Your assignment will be graded using different test cases. Test cases will vary in *left* and *right* programs and the depth of the three. Since the test results will be auto-graded your output must match with the sample output. Please do not forget to leave a single white space between two words and move to the next line immediately after the last character. In the sample runs, some console lines do not fit into the page and they span multiple lines in the document. You should not move to a new line in your program for these cases. Please check the line numbers on the left of sample output.

- Report (10 pts): Your report has to meet the requirements.

5 Sample Runs

As described in the previous sections, the main program requires two executable files called **right** and **left** to be executed by the left or right children. The core functionality of these programs involve accepting two integers as input via the scanf function. Upon receiving these inputs, the programs proceed to perform a series of predefined operations on these numbers. The outcome of these operations is then outputted as an integer result. In a nutshell, to run the main program, you need two executable files called right and left. Considering the running example in Figure 1, corresponding *left* and *right* programs are provided in Listings 2 and 3.

Example 2: Left program's source code: pl.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int num1, num2;
    // This program does not take any input
    // from the console.
     if (argc != 1) {
        printf("Usage: %s \n", argv[0]);
        return 1; // Error code for incorrect usage
    }
    scanf("%d", &num1);
    scanf("%d", &num2);
    // Calculate the multiplication
    int result = num1 + num2;

    // Print the result
    printf("%d\n", result);

    return 0; // Successful execution
}
```

Example 3: Right program's source code: pr.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int num1, num2;
    // This program does not take any input
    // from the console.
     if (argc != 1) {
        printf("Usage: %s \n", argv[0]);
```

```
        return 1; // Error code for incorrect usage
    }
    scanf("%d", &num1);
    scanf("%d", &num2);
    // Calculate the multiplication
    int result = num1 * num2;

    // Print the result
    printf("%d\n", result);

    return 0; // Successful execution
}
```

Examples 2 and 3 provide addition and multiplication code for creating executables corresponding to the left and right children, respectively. To compile these programs, one might use the following commands:

```
> gcc -o left pl.c -Wall
> gcc -o right pr.c -Wall
```

After compilation, you will have two executables, left and right, ready for execution. For a sample execution of TreePipe using these binaries, see Sample Run 2.

- Additionally, be aware that during the evaluation of your assignments, executable files performing operations other than the ones specified will also be considered.

Sample Run 1

Command: `treePipe 0 0`

Output:

```
1 Usage: treePipe <current depth> <max depth> <left-
    right>
```

Sample Run 2

Left Executable Operation: `Addition`
Right Executable Operation: `Multiplication`
Command: `treePipe 0 2 0`

Output:

```
> Current depth: 0, lr: 0
Please enter num1 for the root: 2
---> Current depth: 1, lr: 0
------> Current depth: 2, lr: 0
------> My num1 is: 2
------> My result is: 3
---> My num1 is: 3
------> Current depth: 2, lr: 1
------> My num1 is: 3
------> My result is: 3
---> Current depth: 1, lr: 0, my num1: 3, my num2: 3
---> My result is: 6
> My num1 is: 6
---> Current depth: 1, lr: 1
------> Current depth: 2, lr: 0
------> My num1 is: 6
------> My result is: 7
---> My num1 is: 7
------> Current depth: 2, lr: 1
------> My num1 is: 7
------> My result is: 7
---> Current depth: 1, lr: 1, my num1: 7, my num2: 7
---> My result is: 49
> Current depth: 0, lr: 0, my num1: 6, my num2: 49
> My result is: 55
The final result is: 55
```

Sample Run 3

Left Executable Operation: `Addition`
Right Executable Operation: `Multiplication`
Command: `treePipe 0 3 0`

Output:

```
> Current depth: 0, lr: 0
Please enter num1 for the root: 1
---> Current depth: 1, lr: 0
------> Current depth: 2, lr: 0
---------> Current depth: 3, lr: 0
---------> My num1 is: 1
---------> My result is: 2
------> My num1 is: 2
---------> Current depth: 3, lr: 1
---------> My num1 is: 2
---------> My result is: 2
------> Current depth: 2, lr: 0, my num1: 2, my num2:
   2
------> My result is: 4
---> My num1 is: 4
------> Current depth: 2, lr: 1
---------> Current depth: 3, lr: 0
---------> My num1 is: 4
---------> My result is: 5
------> My num1 is: 5
---------> Current depth: 3, lr: 1
---------> My num1 is: 5
---------> My result is: 5
------> Current depth: 2, lr: 1, my num1: 5, my num2:
   5
------> My result is: 25
---> Current depth: 1, lr: 0, my num1: 4, my num2: 25
---> My result is: 29
> My num1 is: 29
---> Current depth: 1, lr: 1
------> Current depth: 2, lr: 0
---------> Current depth: 3, lr: 0
```

```
---------> My num1 is: 29
---------> My result is: 30
------> My num1 is: 30
---------> Current depth: 3, lr: 1
---------> My num1 is: 30
---------> My result is: 30
------> Current depth: 2, lr: 0, my num1: 30, my num2:
    30
------> My result is: 60
---> My num1 is: 60
------> Current depth: 2, lr: 1
---------> Current depth: 3, lr: 0
---------> My num1 is: 60
---------> My result is: 61
------> My num1 is: 61
---------> Current depth: 3, lr: 1
---------> My num1 is: 61
---------> My result is: 61
------> Current depth: 2, lr: 1, my num1: 61, my num2:
    61
------> My result is: 3721
---> Current depth: 1, lr: 1, my num1: 60, my num2:
   3721
---> My result is: 223260
> Current depth: 0, lr: 0, my num1: 29, my num2:
   223260
> My result is: 223289
The final result is: 223289
```

Sample Run 4

Left Executable Operation: `Addition`
Right Executable Operation: `Multiplication`
Command: `treePipe 0 0 0`

```
> current depth: 0, lr: 0
Please enter num1 for the root: 10
> my num1 is: 10
> my result is: 11
```

```
The final result is: 11
```