

COMP2310-2002-02

Assignment 1: cee (Concurrent Expression Evaluator)

Proposed due date: 11.59 p.m. EST, Friday 13 September 2002

This assignment is worth 15% of your final mark. It will be marked out of 50. Of the 50 marks, the documentation is worth 30, the code 15, and tests of your code 5. **Take note!**

A few years ago I would have indicated some sort of time frame for this assignment (I guess about 10–15 hours), and I would have said that those who have already done all of the parts of the labs, and spent some time reading manuals and documentation and trying out process and pipe programming might need considerably less. I think that's still basically right, but I have observed that there is an order of magnitude difference between the 'fastest' and the 'slowest' students. So you could have it done in well under ten hours, but it might take you fifty.

The program you will write is not going to be long (fewer than 150 lines of code); the main difficulty is figuring out how to use all of the various library and system calls. If you are having problems, seek help *early!* I will not consider an assignment submission to be complete unless it contains both a substantial amount of the necessary code and coherent, consistent documentation. (So, even though the documentation is worth 60% of the mark, I won't give you the necessary 40% overall if you don't submit any code.)

I will answer questions as necessary during lecture time, and, of course, on the newsgroups.

Part Two of this specification is on the web page. It contains an up-to-date list of corrections, hints and tips for getting the assignment done. You should refer to it frequently! The web page also contains a link to the marking guide that the tutors and I will use to assess your work.

1 Introduction

The term 'workflow' has been a buzzword for some time. The basic idea is that a master business process is composed of number of individual processes, and there is some mechanism that routes intermediate products between these lower-level processes. A typical example is high-end publishing. Between creating content and the finished product, documents pass through stages such as preflight, trimming and bleeding, colour conversion, colour separation, and ripping. In a particular publishing solution, some of these steps may be performed by separate software products; a workflow manager combines them seamlessly (by feeding intermediate files through the separate programs in the correct sequence) so that in most cases there need be no user interaction at all.

In this assignment you will implement a very simple workflow manager called cee, which will connect a number of programs that compute the values of integer functions.

2 Components

Each component is a stand-alone program (they can be run from the command line, for example), with a filename which consists of exactly one letter. (Examples: A, J, x; case is significant.) Each component reads a series of lines from standard input until end of file is reached, at which point the component exits.

The component expects to find exactly one natural number (i.e. non-negative integer) on each line of input, according to the following syntax:

```
number ::= digit | digit number
digit := "0" | "1" | . . . | "9"
```

Examples: 0, 123, 33283.

Having read a line of input, the component evaluates a function (which depends on the particular component) and outputs the result – also a natural number – on standard output. In the following, we notate the function computed by a component by using italics. For instance, we say that program called W computes a function $W : \mathbb{N} \rightarrow \mathbb{N}$.

I have provided a number of example components for you to use; see Part Two for details. Of course, you are free to write your own, but you do not have to do so (and no marks are allocated for it).

3 Workflow specifications

Components are joined together in a workflow specification. This takes the form of an arithmetic expression according to the following syntax:

```
workflow ::= letter |
           "(" workflow "+" workflow ")" |
           "(" workflow "*" workflow ")"
letter := "A" | . . . | "Z" | "a" | . . . | "z"
```

Examples: I, (A+b), ((X+(y*Z))*w). A letter may appear multiple times, e.g. (X+(y*X)).

A given workflow specification can be applied to a particular natural number. This is done by applying each of the components mentioned in the workflow to this natural number, and combining the results according to the workflow formula.

For example, consider the example ((X+(y*Z))*w). This workflow applied to the number 5 gives ((X(5) + (y(5)*Z(5))) * w(5)). And (X+(y*X)) applied to the number 7 gives (X(7) + (y(7)*X(7))).

4 Input/output

Your cee program then reads the following from standard input:

- one line containing a workflow specification (according to the syntax in Section 3),
- zero or more lines, each containing a natural number (according to the syntax in Section 2).

For each line containing a number, cee outputs the result of applying the given workflow to that number.

You may assume that the input I test your program with will be well-formed. However, you are free to write code that checks the validity of the input.

5 How cee must do it

You will implement the following basic algorithm:

1. Read the workflow specification from standard input.
2. Construct the workflow according to the specification and connect standard input to the workflow.

You will implement step 2 as follows:

```

construct_workflow(flow)
  if flow is a single letter then
    invoke exec() on that letter
  else
    flow must be of the form (F op G)
    (where F and G are workflows and op is either + or *)
    make four (one-way) pipes, P1, P2, P3, and P4
    fork two new processes
    in the parent process:
      close pipe file descriptors not needed
      while there is more input on standard input
        read a number (and new line character) from standard input
        send the number down P1 and P3
        read the results from P2 and P4
        combine the results using op
        print the final result on standard output (followed by a new line)
      end
      close P1 and P3
      wait for the two child processes to terminate
      exit(0)
    /* end parent process */
    in the first new child process:
      redirect standard input to come from P1
      redirect standard output to go to P2
      close all other pipe file descriptors
      construct_workflow(F)
    /* end child process 1 */
    in the second new child process:
      redirect standard input to come from P3
      redirect standard output to go to P4
      close all other pipe file descriptors
      construct_workflow(G)
    /* end child process 2 */
  end
end

```

Thus `cee` creates a number of processes arranged in a (virtual) tree structure, in which the leaf nodes are individual components, and non-leaf nodes combine the results generated by their children.

If there are n letters in the formula, `cee` will call `fork()` a total of $2n - 2$ times; thus there will be a total of $2n - 1$ processes. You may not assume that a component produces consistent output; thus if, say, `X` occurs twice in a workflow specification, you will have to execute two copies of `X`.

You'd better not call the pipes `P1`, `P2`, etc. Choose more meaningful names. (The same goes for the other variable names mentioned in the algorithm.)

A note about the instruction "invoke `exec()` on that letter": in fact, you should precede the letter by `./` to make sure you get the command that's in the current directory. For example, if the letter is `X`, you run the command `./X`.

6 Debugging mode

If `argc` is greater than 1, your program must run in a 'debugging mode', where extra information is printed on standard error:

- the (successively smaller) workflow specifications being considered by the `construct_workflow` algorithm,
- the input/output values seen and generated by the non-leaf nodes.

See the sample executable program to see the format I am after. Although the behaviour of your program is non-deterministic (due to the scheduling of processes), please follow the format of the output. If I *sort* the lines of output of your program and mine, there must be no difference. Please confirm that the output of your program – whether in debugging mode or not – matches mine, so that the automatic testing will work correctly.

7 Your submission

Do read this section carefully!

Write a noweb source `cee.nw` and a Makefile. (See Part Two for details.) **Hot tip:** before submitting, copy all the files you are going to submit into an empty directory and test your submission using the provided Makefile.

The Makefile specifies two important targets as follows. The command `make cee` runs `notangle` and compiles your program. The build should produce (at least) an executable program called `cee`. C code is to be compiled using `gcc -Wall`, and the compiler must issue no warnings!

We will look at your source. The command `make cee.dvi` runs `noweave` and `latex`. Make sure that this process produces a result which is pleasing to the eye and easy to follow! Remember that you are writing for a human reader, not a computer. The ratio of documentation to code should be greater than 1:1. You will be penalized (up to five marks, i.e. 10%) for any errors in spelling or grammar. Running the spelling checker is essential, but not sufficient! **There is an absolute page limit of 25 pages, not counting cover page, table of contents, or indexes. You will be penalized if you exceed this limit.**

Use pictures and diagrams where appropriate. You may produce them in \LaTeX , or use `xfig` to generate Encapsulated PostScript for inclusion with the `\includegraphics` command. (Any EPS files must have a `.eps` file suffix.) **Hot tip:** there are five marks allocated for ‘descriptive aids’.

You will be penalized for any use of ‘magic numbers’ and other dodgy coding practices.

There are marks allocated for an explanation of your solution, as follows (and referred to in the marking guide as ‘The three questions’). Include in documentation sections near the relevant code:

- your use of processes
- how you communicate data between processes
- a considered opinion of your program: is your solution more elegant than it would be if it were purely sequential (i.e. did not use multiple processes)?

Submit your assignment with the command:

```
mark comp2310 cee Makefile cee.nw *.eps
```

(make sure that you submit all of your PostScript figures!).

8 Debugging

Debugging multi-process programs is *hard*. Desk check your programs (i.e. on paper); you can have the separate processes print to the screen for debugging purposes – don’t forget to comment out (or `#define out`) that code before you submit! Tip: print debugging information to standard error.

9 And last of all ...

Have fun!