

COMP2310 Assignment 1

Felix Andrews 3285754

September 19, 2002

Contents

1	Workflow Concepts	2
1.1	In Abstract	2
1.2	UNIX IPC	2
2	The Concurrent Expression Evaluator	2
2.1	Workflow Tree	3
2.2	Recursive Tree Construction	3
3	Implementation	5
3.1	Reading the workflow specification	6
3.2	Using the workflow specification	6
3.2.1	Simple function	6
3.2.2	Composite specification	7
3.3	Plumbing	10
3.4	Evaluation Loop	12
4	Appropriateness of Design	15
5	List of refinements	16
6	Index	16

Emil, sorry this is a day late. I was caught on some obscure snags. This is only my second C program, and I had to learn to use things like headers, `gdb` and the horrors of string manipulation (coming from Perl!). I was stuck for a while until I realised that the example components were compiled on Sun machines, and couldn't be executed on Linux! (I rewrote them in Perl).

1 Workflow Concepts

1.1 In Abstract

A workflow is a system of connected components that together does a job. This modular division of labour has many advantages including flexibility, maintainability, and potential concurrency. Workflows are usually directed: the output of one component feeds along the “processing chain” to the input of the next process. Of course, it may be more complex than a simple linear chain, having cyclic, parallel or tree structures. As such there may be concurrent components involved, but the word “flow” implies that the process is broadly sequential. However, it is theoretically possible for the serial components to be working simultaneously on different work units.

1.2 UNIX IPC

We can model workflows conveniently in UNIX by representing components by processes. Indeed, this is how much of the operating system works anyway (especially micro-kernels). One-way communication channels called pipes can be established, the main mechanism for Inter-Process Communication (IPC). Unified input/output allows a child process to be connected to a pipe transparently (without the child knowing). New processes are created in UNIX by the forking (replicating) of existing ones; this is integral to the establishment of IPC channels, but the details are outside the scope of this description.

2 The Concurrent Expression Evaluator

The workflow developed here is for the evaluation of simple mathematical expressions. It reads a workflow specification from the user, which is composed of additions or multiplications of arbitrary functions, according to the following format.

$$\text{flowspec} ::= \text{letter} \mid (\text{flowspec} \odot \text{flowspec}) \quad \text{where } \odot ::= + \mid *$$

An example of a valid workflow specification is $((p + q) * R)$. Each letter is a function in the expression (a component in the workflow). It is assumed that all listed components exist as executable programs.

The next phase of operation is the evaluation of the expression on a series of positive integers. These are read from the user and the results of each function operating on them combined as specified. The functions are “black boxes” from our perspective; they must simply accept integer input and produce positive integer output.

2.1 Workflow Tree

We can construct a workflow tree to solve this problem, reflecting the inherently recursive structure of our expressions. This is easily defined: an expression always either has exactly 2 subexpressions which can be evaluated independently (forming a node with 2 children), or it is an external function which can be called directly (forming a leaf node). See Figure (1).

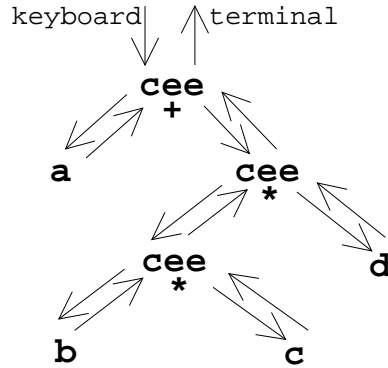


Figure 1: A “workflow tree” composed of processes connected by pipes. This tree corresponds to the expression $(a + ((b * c) * d))$. The symbol beneath each `cee` process shows which operation it uses to combine the output of its children. The overall tree is shown connected to the keyboard and terminal, as they are the default standard input and output streams, but they could just as well be other devices, processes or files.

2.2 Recursive Tree Construction

The main problem in constructing a tree of processes for function evaluation is the passing of input (a number) to all nodes and the results back to the root (giving the final result). We set this up by connecting each node’s standard input and output via pipes to its parent. This happens, as we will soon see, when new subexpressions (children) are forked; each child then enters the following function and can safely use standard input/output without worrying about its position in the tree.

The workflow construction function is recursive, so we deal with the termination case first: if the expression is a single function we execute it (replacing the current process) and let it handle the input/output streams (any pipe connections remain).

```
3 <workflow construction function 3>≡ (5) 4▷  
  void construct_workflow(char *flow_spec)  
  {
```

```

    if (<workflow specification is a single letter 6c>) {
        <exec component in place 7a>
    }

```

Defines:

`construct_workflow`, used in chunks 4 and 5.

If we enter the following `else` branch, `flow_spec` must be of the form (F op G), where F and G are workflow specifications and op is either + or *. So we need to decompose it and create pipe-connected children to handle the subexpressions.

```

4 <workflow construction function 3>+≡ (5) <3
    else {
        <workflow variables 7b>
        <decompose workflow specification 7c>
        <create pipes 10b>
        if (fork() != 0) {
            if (fork() != 0) {
                /* parent */
                <setup pipes for parent 10c>
                while (<more standard input 12d>) {
                    <evaluate expression on an integer 13a>
                }
                <parent pipe cleanup 11b>
                <wait for children 12b>
                exit(0);
            }
            else {
                /* child handling workflow F */
                <setup pipes for F 11c>
                construct_workflow(F);
            }
        }
        else {
            /* child handling workflow G */
            <setup pipes for G 12a>
            construct_workflow(G);
        }
    }
}

```

Uses `construct_workflow` 3 and F 7b.

3 Implementation

The main program is very simple. It is essentially a wrapper around the recursive function above, to which it supplies the initial workflow specification (see 3.1).

```
5  <* 5>≡
    #define _POSIX_SOURCE

    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/types.h>
    #include <sys/wait.h>
    #include <string.h>
    #include <ctype.h>

    #define MAX_LINE 79

    static int debug = 0;

    <workflow construction function 3>

    int main(int argc, char *argv[])
    {
        char *workflow_spec;
        if (argc > 1) debug = 1;
        setvbuf(stdin, NULL, _IONBF, 0);
        <read workflow specification 6a>
        construct_workflow(workflow_spec);
        return 0;
    }
```

Defines:

- _POSIX_SOURCE, never used.
- debug, used in chunks 6b, 9c, and 14b.
- main, never used.
- MAX_LINE, used in chunks 6a and 7b.

Uses construct_workflow 3.

A couple of comments on that chunk: the `debug` flag is raised if there are any command line arguments. The `setvbuf` command turns off buffering of the standard input stream. It's not necessary with keyboard input (line buffered) but fixes problems involved with reading from a file (block buffered).

3.1 Reading the workflow specification

The workflow specification is read from the first line of standard input. All subsequent input is handled by the workflow tree. First it is necessary to allocate enough memory to handle the longest possible specification of `MAX_LINE` characters.

```
6a <read workflow specification 6a>≡ (5) 6b▷
    workflow_spec = malloc(sizeof(char) * MAX_LINE);
    fgets(workflow_spec, MAX_LINE, stdin);
    /* chop off trailing newline */
    workflow_spec[strlen(workflow_spec) - 1] = '\0';
```

Uses `MAX_LINE` 5 and `strlen` 6c.

In debugging mode we would like to know whether the initial specification has been read in correctly. Incidentally, the unbuffered output stream `stderr` remains connected to the terminal (by default), so these messages don't interfere with expression evaluation.

```
6b <read workflow specification 6a>+≡ (5) <6a
    if (debug)
        fprintf(stderr, "Initial spec='%s'\n", workflow_spec);
```

Uses `debug` 5.

3.2 Using the workflow specification

All code from here on is within `construct_workflow`. The current specification (string argument to the function) is called `flow_spec` to distinguish it from the initial one in the main program.

3.2.1 Simple function

To determine whether the specification at this point is a simple function, we compare its length to 1 (all functions must be one letter).

```
6c <workflow specification is a single letter 6c>≡ (3)
    strlen(flow_spec) == 1
```

Defines:

`strlen`, used in chunks 6a, 8, and 9.

If so, we would like to run that external function in place of the current process. The final line of the following code chunk does that. Before that, we ensure that the name is valid (alphabetic), and prepend the sequence “./” to explicitly identify the executable in the current directory.

```
7a  <exec component in place 7a>≡ (3)
    char local_flow[] = "./ ";
    if (! isalpha(flow_spec[0])) {
        fprintf(stderr, "Invalid component: %c\n", flow_spec[0]);
        exit(1);
    }
    local_flow[2] = flow_spec[0];
    execlp(local_flow, local_flow, NULL);
Defines:
    local_flow, never used.
```

3.2.2 Composite specification

```
7b  <workflow variables 7b>≡ (4) 8b>
    char F[MAX_LINE], G[MAX_LINE];
    int i, pivot, paren_depth;
Defines:
    F, used in chunks 4 and 7–9.
    i, used in chunk 8a.
    paren_depth, used in chunks 7c and 8a.
    pivot, used in chunks 7–9.
Uses MAX_LINE 5.
```

The strings F and G will hold the left and right subexpressions, respectively. `pivot` will be the index of `flow_spec` at which to split into F and G (the index of the operator). `paren_depth` keeps track of our context (layers of parentheses) as we scan through `flow_spec`. They are initialised to be empty and zero:

```
7c  <decompose workflow specification 7c>≡ (4) 8a>
    F[0] = G[0] = '\0';
    pivot = 0;
    paren_depth = 0;
Uses F 7b, paren_depth 7b, and pivot 7b.
```

The following loop scans the characters of the specification, breaking and recording the pivot when an operator is found at the outermost level. Up to that point each character is copied into F; thus we have the first subexpression.

```
8a  <decompose workflow specification 7c>+≡ (4) <7c 8c>
    /* we omit 1 char at each end, assumed to be parentheses */
    for (i = 1; i < strlen(flow_spec) - 1; ++i) {
        switch (flow_spec[i]) {
            case '+':
            case '*':
                if (paren_depth == 0)
                    pivot = i;
                break;
            case '(':
                ++paren_depth; break;
            case ')':
                --paren_depth; break;
        }
        if (pivot) break;
        else F[i-1] = flow_spec[i];
    }
    F[i-1] = '\\0'; /* terminate F */
```

Uses F 7b, i 7b, paren_depth 7b, pivot 7b, and strlen 6c.

We store the operator in op.

```
8b  <workflow variables 7b>+≡ (4) <7b 10a>
    char op;
```

Defines:
op, used in chunks 8, 9, and 13e.

```
8c  <decompose workflow specification 7c>+≡ (4) <8a 9a>
    op = flow_spec[pivot];
```

Uses op 8b and pivot 7b.

Now we need to extract the substring of `flow_spec` from just after the pivot to the second-last character (omitting the closing bracket). The function `strcpy` copies one entire string to another, but we can pass it the address of the $[pivot + 1]^{st}$ character, and it will treat that as the start of the string. All that remains is removing the last character, by inserting the termination sequence (null character) where we would like to end.

```
9a  <decompose workflow specification 7c>+≡ (4) <8c 9b>
      strcpy(G, &flow_spec[pivot+1]);
      G[strlen(G)-1] = '\0'; /* chop last char */
```

Uses pivot 7b and `strlen` 6c.

Basic sanity checking. There are only two valid operators, and all valid expressions F satisfy $(|F| - 1) \bmod 4 = 0$.

```
9b  <decompose workflow specification 7c>+≡ (4) <9a 9c>
      if ((op != '+' && op != '*') ||
          ((strlen(F) - 1) % 4 != 0) ||
          ((strlen(G) - 1) % 4 != 0)) {
          fprintf(stderr, "Broken spec: '%s|%c|%s'\n", F, op, G);
          exit(1);
      }
```

Uses F 7b, op 8b, and `strlen` 6c.

In debug mode this line will show where each workflow specification was split.

```
9c  <decompose workflow specification 7c>+≡ (4) <9b>
      if (debug)
          fprintf(stderr, "Divided spec; F='%s',op='%c',G='%s'\n", F, op, G);
```

Uses debug 5, F 7b, and op 8b.

3.3 Plumbing

We now turn to the intricacies of establishing pipe connections between the parent process and its two children.

A pipe is a one-way FIFO (first in first out) buffer. It is identified by a 2-element array, where [0] is the file descriptor for reading, and [1] is the file descriptor for writing to the pipe. Since we want to send and receive information to/from two child processes, we need four pipes: input to process F, output from F, input to G, output from G. The `pipe` command creates them.

```
10a  <workflow variables 7b>+≡ (4) <8b 10d>
      int F_in_pipe[2], F_out_pipe[2];
      int G_in_pipe[2], G_out_pipe[2];
```

Defines:

`F_in_pipe`, used in chunks 10–12.

`G_in_pipe`, used in chunks 10–12.

```
10b  <create pipes 10b>≡ (4)
      pipe(F_in_pipe);
      pipe(F_out_pipe);
      pipe(G_in_pipe);
      pipe(G_out_pipe);
```

Uses `F_in_pipe` 10a and `G_in_pipe` 10a.

It is good practice to close any file descriptors that aren't needed, particularly in a recursive context such as this (since the FDT could potentially fill up). The parent doesn't need to read from the F input pipe, write to the F output pipe, etc.

```
10c  <setup pipes for parent 10c>≡ (4) 11a>
      close(F_in_pipe[0]);
      close(F_out_pipe[1]);
      close(G_in_pipe[0]);
      close(G_out_pipe[1]);
```

Uses `F_in_pipe` 10a and `G_in_pipe` 10a.

Reading and writing is easier if we can use a higher-level construct than file descriptors, namely streams (type `FILE*`). They can be derived from our pipe file descriptors with `fdopen`.

```
10d  <workflow variables 7b>+≡ (4) <10a 12c>
      FILE *F_in, *F_out, *G_in, *G_out;
```

Defines:

`F_in`, used in chunks 11 and 13b.

`F_out`, used in chunks 11 and 13d.

`G_in`, used in chunks 11 and 13b.

`G_out`, used in chunks 11 and 13d.

11a $\langle \text{setup pipes for parent } 10c \rangle + \equiv$ (4) $\triangleleft 10c$

```

F_in = fdopen(F_in_pipe[1], "w");
F_out = fdopen(F_out_pipe[0], "r");
G_in = fdopen(G_in_pipe[1], "w");
G_out = fdopen(G_out_pipe[0], "r");

```

Uses `F_in` 10d, `F_in_pipe` 10a, `F_out` 10d, `G_in` 10d, `G_in_pipe` 10a, and `G_out` 10d.

Pipe file descriptors are closed when the associated streams are closed, so the following is a sufficient cleanup.

11b $\langle \text{parent pipe cleanup } 11b \rangle \equiv$ (4)

```

fclose(F_in);
fclose(F_out);
fclose(G_in);
fclose(G_out);

```

Uses `F_in` 10d, `F_out` 10d, `G_in` 10d, and `G_out` 10d.

To connect a child process transparently to the other end of the pipes, we want to use them as standard input (`stdin`) and standard output (`stdout`). `stdin` is by definition file descriptor 0, so we close that. Next we duplicate the file descriptor corresponding to the reading end of the pipe incoming to `F`: this copies it to the first available space which must be 0. Now, `stdin` comes from a pipe. An analogous operation sets up `stdout` (file descriptor 1) to write to the relevant pipe.

11c $\langle \text{setup pipes for } F \text{ } 11c \rangle \equiv$ (4) 11d \triangleright

```

close(0);
dup(F_in_pipe[0]);
close(1);
dup(F_out_pipe[1]);

```

Uses `F_in_pipe` 10a.

We close the other file descriptors; they are not needed here. Even though we close the file descriptors corresponding to pipes we need, they are maintained through the `stdin` / `stdout` entries.

11d $\langle \text{setup pipes for } F \text{ } 11c \rangle + \equiv$ (4) $\triangleleft 11c$

```

close(F_in_pipe[0]);
close(F_in_pipe[1]);
close(F_out_pipe[0]);
close(F_out_pipe[1]);
close(G_in_pipe[0]);
close(G_in_pipe[1]);
close(G_out_pipe[0]);
close(G_out_pipe[1]);

```

Uses `F_in_pipe` 10a and `G_in_pipe` 10a.

A similar procedure transparently connects G to the parent.

```
12a  <setup pipes for G 12a>≡ (4)
      close(0);
      dup(G_in_pipe[0]);
      close(1);
      dup(G_out_pipe[1]);
      close(F_in_pipe[0]);
      close(F_in_pipe[1]);
      close(F_out_pipe[0]);
      close(F_out_pipe[1]);
      close(G_in_pipe[0]);
      close(G_in_pipe[1]);
      close(G_out_pipe[0]);
      close(G_out_pipe[1]);
```

Uses F_in_pipe 10a and G_in_pipe 10a.

The `wait` call in the parent blocks until a child process dies. We want to wait for both children before exiting ourselves, so we call it twice. The return status is not important at this stage, so we ignore it with the `NULL` pointer.

```
12b  <wait for children 12b>≡ (4)
      wait(NULL);
      wait(NULL);
```

3.4 Evaluation Loop

This section is from the perspective of the parent, repeatedly reading a number, passing it on to the subexpression processors, and sending the combined result back (to `stdout`). We call the number *x*.

```
12c  <workflow variables 7b>+≡ (4) <10d 13c>
      unsigned int x;
```

Defines:

`x`, used in chunks 12–14.

The condition for continuing in the evaluation loop. We should continue until the end of standard input. The `fscanf` call below tries to read an unsigned integer from `stdin`, returning `EOF` on end-of-input. This has the “side effect” of reading in the next integer, *x*.

```
12d  <more standard input 12d>≡ (4)
      fscanf(stdin, "%u", &x) != EOF
```

Defines:

`fscanf`, used in chunk 13d.

Uses `x` 12c.

Next we elaborate the tasks involved in evaluating an expression...

13a \langle *evaluate expression on an integer* 13a $\rangle \equiv$ (4)
 \langle *send x to F and G* 13b \rangle
 \langle *collect results of F(x) and G(x)* 13d \rangle
 \langle *combine results according to workflow specification* 13e \rangle
 \langle *print final result on standard output* 14a \rangle

Passing the number on to children is just a matter of printing it (terminated by a newline) to the appropriate pipe stream. However, we want it to be available immediately, not buffered until some threshold of characters is reached; we force this with `fflush`.

13b \langle *send x to F and G* 13b $\rangle \equiv$ (13a)

```
fprintf(F_in, "%u\n", x);
fprintf(G_in, "%u\n", x);
fflush(F_in);
fflush(G_in);
```

Uses `F_in` 10d, `G_in` 10d, and `x` 12c.

`F_x` corresponds to the subexpression evaluation $F(x)$, similarly with `G_x`. Reading into these from the two child processes is essentially the inverse operation to printing: we scan the appropriate pipe stream for an unsigned integer.

13c \langle *workflow variables* 7b $\rangle + \equiv$ (4) <12c

```
unsigned int F_x, G_x, result;
```

 Defines:
`F_x`, used in chunks 13 and 14b.
`G_x`, used in chunks 13 and 14b.
`result`, used in chunks 13 and 14.

13d \langle *collect results of F(x) and G(x)* 13d $\rangle \equiv$ (13a)

```
fscanf(F_out, "%u", &F_x);
fscanf(G_out, "%u", &G_x);
```

Uses `F_out` 10d, `F_x` 13c, `fscanf` 12d, `G_out` 10d, and `G_x` 13c.

The `result` of the current expression evaluation is obtained by adding or multiplying the 2 subexpression evaluations.

13e \langle *combine results according to workflow specification* 13e $\rangle \equiv$ (13a)

```
if (op == '+')
    result = F_x + G_x;
else if (op == '*')
    result = F_x * G_x;
```

Uses `F_x` 13c, `G_x` 13c, `op` 8b, and `result` 13c.

Send it back towards the root of the tree; if this is the root, the result is visible to the user.

14a \langle *print final result on standard output 14a* $\rangle\equiv$ (13a) 14b \triangleright
 fprintf(stdout, "%u\n", result);
 fflush(stdout);

Uses result 13c.

14b \langle *print final result on standard output 14a* $\rangle+\equiv$ (13a) \triangleleft 14a
 if (debug)
 fprintf(stderr,
 "Spec '%s' forwarded %u, received back %u and %u, and returned %u\n",
 flow_spec, x, F_x, G_x, result);

Uses debug 5, F_x 13c, G_x 13c, result 13c, and x 12c.

4 Appropriateness of Design

Is this concurrent approach an appropriate solution? Two variants of a single-process design are presented for comparison.

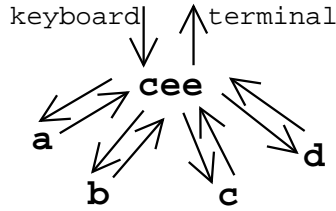


Figure 2: An expression evaluator which has the components as direct children connected by pipes. This diagram is analogous to the Figure (1) and the comparison shows how this “flat” design centralises the combination of results.

Point Evaluation All components are connected by pipes directly to the process (see Figure (2)). When a number is read, the workflow is decomposed recursively until components are reached; then the number is sent down the associated pipe and the result read back. These are combined in the recursive function, ultimately giving the final evaluation. The main problem with this is that the process has to wait for each component in turn to run before continuing. Thus there is no concurrency. Also, there is the overhead of expression decomposition with each number input.

Pre Evaluation Again, all components are connected by pipes directly to the process. When a number is read, it is immediately passed to all components and the results read back. Then, the workflow can be recursively decomposed and evaluated using the known results. This assumes that the results of component evaluations are constant with a given input (deterministic); however even if we don’t want to assume that, we can fork multiple instances of each. This design allows all components to run concurrently, which is nice. Like Design 1, there is the overhead of expression decomposition with each number input.

Both of these designs would be plagued by pipe chaos—there are an indeterminate number of pipes needed, depending on the workflow specification. Keeping track of them all could be messy.

Workflow Tree The design developed here stands up well in comparison. It allows the components to run concurrently. In addition, it allows the combination of results to occur concurrently—if this was computationally intensive, it could be run on a distributed system. A further advantage is

the once-off workflow decomposition: the single-process solutions had to do this on each evaluation. On the other hand, the tree design uses $2n - 1$ processes, whereas the single-process designs use only $\text{uniq}(n) + 1$ or $n + 1$ (where n is the number of components in the workflow, and $\text{uniq}(n)$ is the number of unique components).

Finally, the workflow tree is elegant. Pipes are handled relatively easily because each subexpression evaluator only has 2 children. The inter-process communication is intuitive.

5 List of refinements

- < * 5>*
- < collect results of $F(x)$ and $G(x)$ 13d>*
- < combine results according to workflow specification 13e>*
- < create pipes 10b>*
- < decompose workflow specification 7c>*
- < evaluate expression on an integer 13a>*
- < exec component in place 7a>*
- < more standard input 12d>*
- < parent pipe cleanup 11b>*
- < print final result on standard output 14a>*
- < read workflow specification 6a>*
- < send x to F and G 13b>*
- < setup pipes for F 11c>*
- < setup pipes for G 12a>*
- < setup pipes for parent 10c>*
- < wait for children 12b>*
- < workflow construction function 3>*
- < workflow specification is a single letter 6c>*
- < workflow variables 7b>*

6 Index

`_POSIX_SOURCE`: [5](#)
`construct_workflow`: [3](#), [4](#), [5](#)
`debug`: [5](#), [6b](#), [9c](#), [14b](#)
`F`: [4](#), [7b](#), [7c](#), [8a](#), [9b](#), [9c](#)
`F_in`: [10d](#), [11a](#), [11b](#), [13b](#)
`F_in_pipe`: [10a](#), [10b](#), [10c](#), [11a](#), [11c](#), [11d](#), [12a](#)
`F_out`: [10d](#), [11a](#), [11b](#), [13d](#)
`F_x`: [13c](#), [13d](#), [13e](#), [14b](#)

fscanf: 12d, 13d
G_in: 10d, 11a, 11b, 13b
G_in_pipe: 10a, 10b, 10c, 11a, 11d, 12a
G_out: 10d, 11a, 11b, 13d
G_x: 13c, 13d, 13e, 14b
i: 7b, 8a
local_flow: 7a
main: 5
MAX_LINE: 5, 6a, 7b
op: 8b, 8c, 9b, 9c, 13e
paren_depth: 7b, 7c, 8a
pivot: 7b, 7c, 8a, 8c, 9a
result: 13c, 13e, 14a, 14b
strlen: 6a, 6c, 8a, 9a, 9b
x: 12c, 12d, 13b, 14b