Real-time realizations of a simple programming environment

copyright 2012 Miller Puckette
Creative Commons share-alike license

The computer science and electronic music context

The language of real-time computer music borrows from two antecedents that were fairly well in place over the period during which the field of real-time computer music took its current form (roughly between 1957 and 2000.) Over much the same period of time the field of computer science, perticularly operating systems, was also coming into maturity; perhaps it may be said to have matured by 1980 with the widespread adoption of Unix. Meanwhile, a loosely connected network of electronic music studios arose in the 1950s whose design is directly reflected in the patching paradigm that is nearly universal in modern computer music environments.

Both computer science and music practice relied on a notion of parallelism, albeit in very different forms and languages. In computer science, abstractions such as *process* and *thread* arose from the desire to allcate computing resources efficiently to users. In music, thousand-year old terms like *voice* and *instrument* imply parallelism, both on written scores as multi-part music were notated for both practical and dogmatic reasons, and in real time as live performers sang or played the music in ensembles.

Computer science terminology

The most fundamental abstraction in computer science is the computer, and a subsidiary abstraction is the *job*. An early prototypical computer (fig. 1) sat in a room waiting for jobs to be submitted to it, perhaps in the form of stacks of punched cards. The computer would execute each job in turn, hopefully producing output which could also have been stacks of punched cards. At least three problems arise in this scenario:

- **Idleness.** The computer sometimes had nothing to do and would be idle; idel time reduced the total amount of computation the computer could carry out.

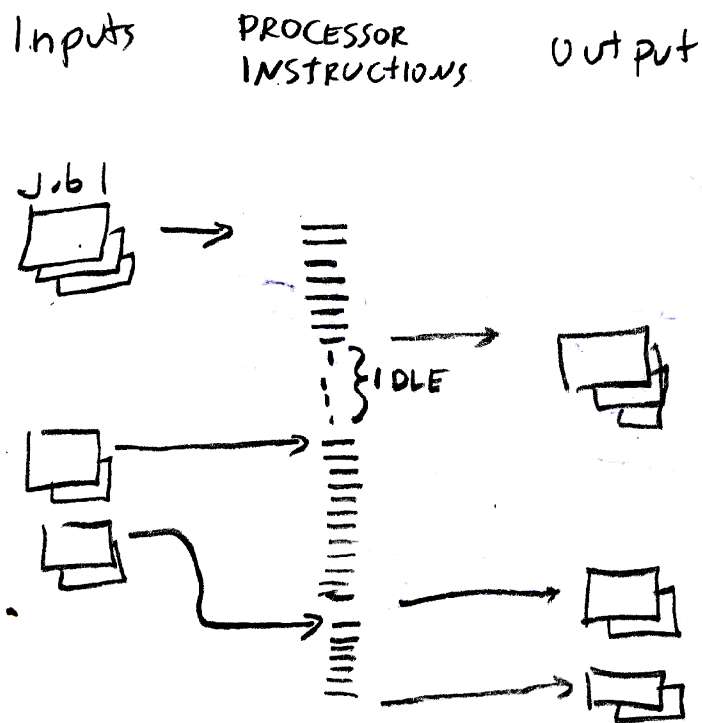

Figure 1: Submitting jobs to a computer circa 1960.

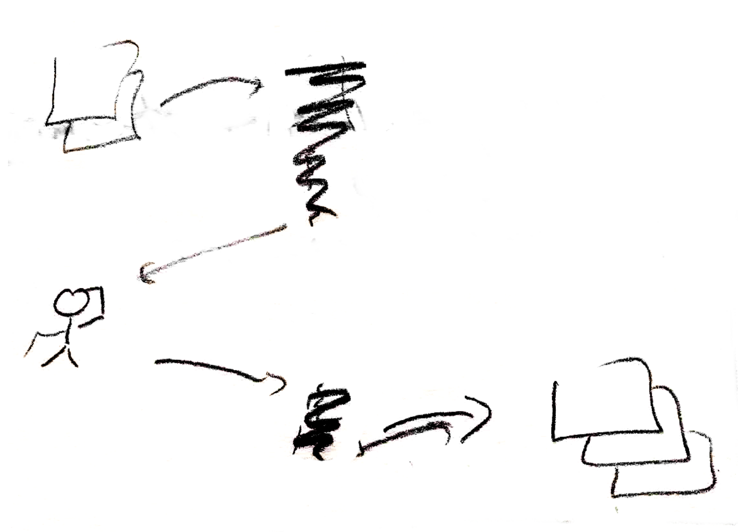

Figure 2: Interactive jobs may stall in mid-computation to ask the operator for more information.

- **Latency.** Sometimes a jobwould be submitted while another job was running (as in job number 2 in the figure); in this case the job would join a queue of waiting jobs. This meant that the submitter of job 2 had to wait longer to harvest the output.
- **Unanticipated data needed.** For many types of jobs you might not be able to predict at the outset what data will be needed during the computation. The "job" model doesn't offer a way for the computer to ask the operator for additional information it might need.

The first two of these only impact the efficiency of computation, but the third requires that we go back and amend the job model altogether; so we will consider that first. Figure 2 shows an amended model of computation allowing *interactive* jobs that may stop executaion part way through and ask the operator for more information. When this happened the job was considered *stalled* and the computer sat idle, so now only the first two problems remain.

Computer science's answer to the problems of idleness and latency have been to introduce *time-sharing* and *multiprocessing.* Time-sharing is the practice of keeping several jobs in progress at teh same time, so that when one job stalls or finishes, the processor's time can then be devoted to some other job that needs running. Perhaps this second job will later stall or finish but meanhile, too, the first job may have become runnable again (having received a new dose of data it had stalled waiting for). The computer would then return to job 1. One

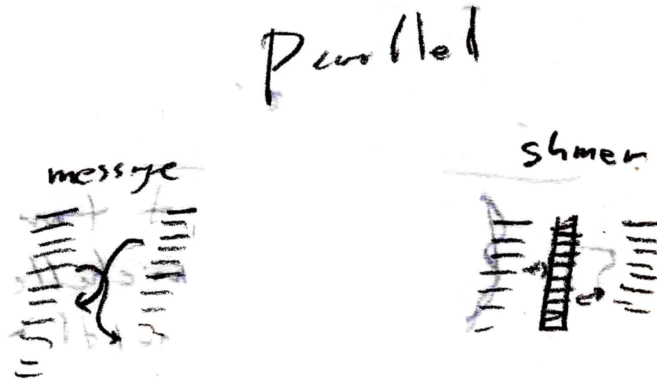

Figure 3: Process intercommunication using messages and shared memory.

could also fill idle time by keeping low-priority jobs waiting in the background (ones whose latency requirements were less strict) that would run whenever all higher-priority jobs were stalled.

The advent of multiprocessors made it possible to further improve throughput in the same way that having several short-order cooks in a diner can speed orders. As the number of jobs and the number of available processors increases, there should be fewer wild swings in the availability of processing power to satisfy the needs of submitted jobs.

The chief tool for time-sharing and multiprocesing is an abstraction calle a *process*, which can be thought of as a virtual computer. When a job is submitted, one creates a brand new (virtual) computer to carry it out, and once the job is finished, the virtual computer, or process, is scrapped. Each job may run in ignorance of all other jobs on the system. Each process gets its own memory and program to run, and its own *program counter*, abbreviated *PC*, that records where in the program the computer is now running. When the computer switches from running one process to another one, the memory and PC (and other context) of the first rocess are retained so that they are available again when the first process is again run in the future.

Although at the outset we could consider all processes to operate in complete ignorance of each other, at some point the need will certainly arise for processes to intercommunicate. Computer science offers at least two paradigms that we

will want to consider: *pessage passing* and *shared memory* (see fig. 3). Of these, the message passing paradigm is less general but easier to analyze and make robust. In message passing, one process can simply send another a packet or a stream of data, that the second one may read at any later time. This is similar conceptually to how people intercommunicate. The chief difficulty using this paradigm is that it does not allow a process to interrogate another directly, except by sending a message and then stalling until a return message is received. This might greatly increase the latency of computations, and worse yet, if we adopted this strategy for interrogation, two processes could conceivably interrogate each other at the same time, so that both end up deadlocked.

In the shared-memory paradigm two processes communicate by reading and writing to a shared area of memory. We can then arrange for one process to be able to interrogate another one simply by looking in the appropriate location in its memory (which, by prior arrangement, we had arranged to share). But now we have to work hard to make sure that our two processes will carry out their computations deterministically, because the order in which the two access the shared memory is not controlled. We would need to set up some convention to manage this. (One such convention could be to format the shared memory into message queues, thus returning us to the situation described in the previous paragraph.) In general, there is no final answer here; any paradigm will either be onerously restricitve or dangerously permissive, or both, and to make good choices will require careful attention to the particulars of the task at hand.

Electronic music terminology

The first widely used model for computer music performance was what is now called *music N*, developed over a series of programs written by Max Mathews starting in 1957[Mat69]; by 1959 his Music 3 program essentially put the idea in its modern form, as exemplified in Barry Vercoe's Csound program. These programs all act as "music compilers" or "renderers", taking a fixed text input and creating a soundfile as a batch output. Although Csound has provisions for using real-time inputs as part of its "rendering" process, in essence the programming model is not interactive.

The Music N input is in the form of an *prchestra* and a *score*, as depicted in fig. 4. The orchestra can be thought of as emulating a 1950s-era *elektronischemusik* studio, in which the hardware is organized in metal boxes with audio inputs and outputs, such as tape recorders, oscillators, pulse generators, filters, ring modulators, and so on. These would be connected by audio cables into a *patch*. Furthermore, the boxes had knobs and switches on them that allowed the user to supply parameters such as the frequency of an oscillator.

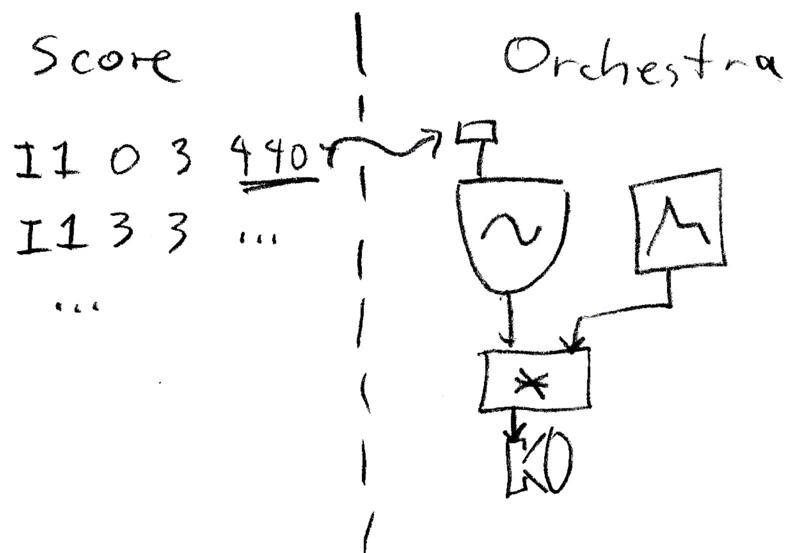

Figure 4: The Music N paradigm

In the music N paradigm, the studio and its patch are represented by the orchestra. Although the actual orchestra file is in a programming language, when publishing algorithms it has traditionally been represented as a block diagram showing a collection of *unit generators* and the audio connections between them.

The score is organized as a list of time-tagged lines that are (either nostagically or deprecatingly) called *score cards*. In additon to one or two time tags (a "note" has two, one for its start and one for its end), a score card has some number of numerical parameters that may be supplied to the unit generators.

The fun begins when we try to adapt this paradigm to run in real time. We could simply connect the Music N output to a real-time audio output; but presumably our reason fro wanting to run in real time was to be able to use live inputs to affect the sound output. Taking the opposite direction, we could require that the user or musician supply all the parameters in real time using knobs and switches, but this quickly revels itself to be unmanageable for the human. We will need to make intelligent decisions, probably different for any two musical situations, as to how the live inputs will affect the production of sound. More generally, our problem is to design a software environment that will give a musician the freedom to make these choices.

In the early 1980s two influential real-time synthesizers were designed, the ystems Concepts Digital Synthesizer (or "Samson Box") at Stanford[Loy], and the 4C synthesizer at IRCAM[MCA+79][Abb81]. Both machines ran a fixed computation loop with a fixed number of steps, with one loop finishing at each tick of the sample clock,

Each of these machine designs got some things right for the first time. The Samson box was the first working machine that could do sample-accurate parameter updates in real time. To do this, the fixed program contained an update period during which items were taken off the head of a time-tagged parameter update queue. This queue was filled by the Foonly controlling computer some tenths of seconds, or whole seconds, in advance, so that the Foonley did not have to react at low latencies to the condition of the queue. This approach had one major limitation: it did not take into account the possibility of real-time interaction. It was physicaly possible to jump the queue for "real-time" parameter updates, but then one lost any ability to determine the timing of such updates accurately.

The 4C machine and its controlling software 4CED were more explicitly designed with real-time interaction in mind, although the timing was less accurate than with the Samson Box. In the 4C parameter updates were effected at interrupt level from the controlling computer; the computer was interrupted by the 4C when one of a bank of timers ran out.

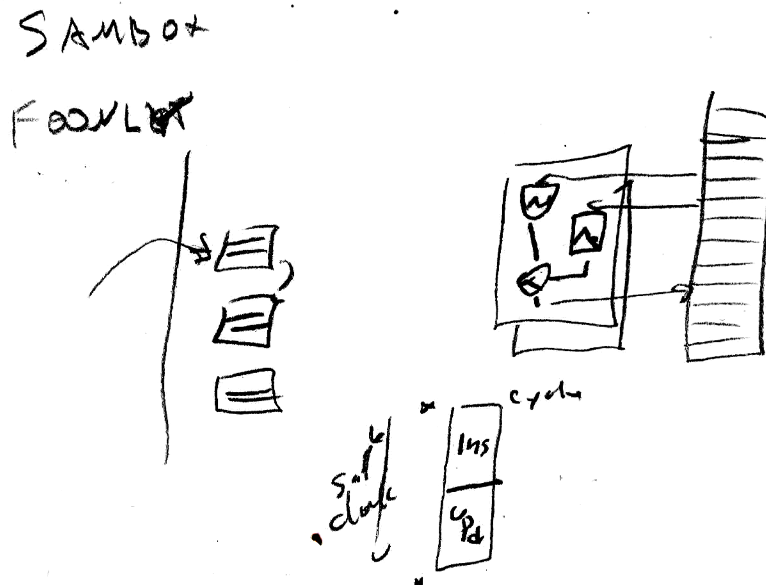

Figure 5: The Samson Box DSP loop and update structure

The 4CED user conceptualized the 4C as a collection of 32 indepndent processes (Abbott's simile was a collection of 32 music boxes that the user could start at any time). The guiding idea seems to have been that a performer could play a keyboard with 32 eys on it, but each key, rather than being restricted to playing a single note, could in turn set off a whole sequence of actions. This would seem to greatly magnify what the keyboard player could do.

It seems also to have been on people's minds that the playing of sequences could usefully be unified with the business of scheduling breakpoints to a pitch or amplitude envelope consisting of many segments. Both the sequencing of collections of notes and the sequencing of emvelope breakpoints led many computer music researchers to think that a process model, as would appear a time-sharing operating system, was a perfect metaphor to reuse in the design of real-time computer music control systems.

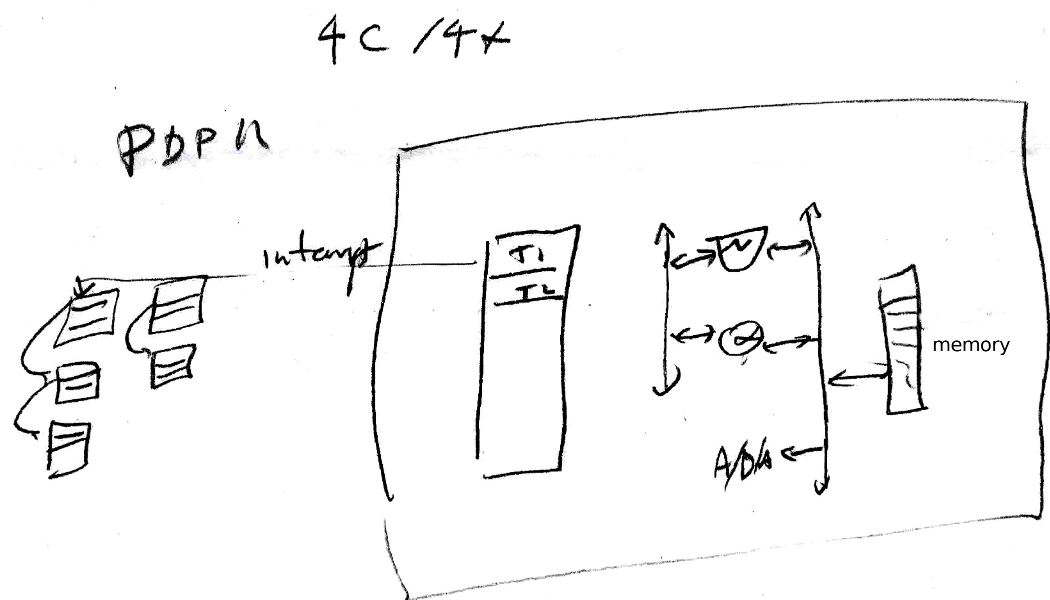

Figure 6: The 4C and the 4CED environment

The `dum` language and interpreter

For what follows we will first develop a small, malleable programming environment named `dum`[1] that we can then implement in a variety of ways to test different possible ways to deal with the scheduling problems that come up, particularly when we wish to interleave several real-time tesks that may want to intercommunicate. First we introduce the `dum` language itself and then we go into the different possible ways of implementing `dum`.

The intention is not to propose `dum` as a potentially viable real-time environment; it is kept simple in order to lay bare the underlying problems and approaches to solving them that one hits in a much more complicated way in more "real" situations.

Definition of the language

In the 1980s the `forth` computer language was widely adopted by music hackers working on personal computers. Here we describe a new environment based very loosely on `forth`, called `dum`. I regard this activity not so much as a demonstration of a computer science principle, but more as a way to smooth out any irregularities `forth` might have.

An `dum` program is a sequence of *words* (with no more structure than that implied by "sequence"). Each word can be either:

- a number;
- the name of a built-in function; or
- an open or close parenthesis.

A sequence of words is called a *list*.

A `dum` program can access data in the form of a stack and a collection of memory locations. The stack and each memory location contain lists, exactly like the program, but unlike the program (which is simply read and executed sequentially), the stack and memory can be both read and written.

The "hello, world" program in `dum` is as shown:

[1] `dum` is an acronym for "Don't use me".

```
( 104 101 108 108 111 44 32 119 111 114 108 100 10 ) print
```

```
--->
104 101 108 108 111 44 32 119 111 114 108 100 10
```

The dum interpreter executes this (or any) program by sequentially reading items from beginning to end. The items are lists of one or more *words*. In the above example there are two items: a list of 13 numbers, and the function print.

Each item the dum interpreter encounters is pushed onto a stack, except functions; these are executed. Functions may pop (take off) arguments from the stack, and may return one or more values by pushing the return values back on the stack. So in the above example, the list of numbers is pushed onto the stack and the function print, which takes one argument, gets it by popping the list back off the stack and prints it to the output. At this point we have reached the end of the program and execution stops.

The print function is special in that it might have some effect on the real world outside of the dum environment itself. Depending on implementation this could cause a teletype machine to print characters onto a roll pf paper or a Pd message to appear. Functions that aren't considered *system* functions may be described as *pure*.

The stack

Consider the following dum program:

```
2 3 + 4 * print
```

```
--->
20
```

The items (three numbers and two functions) are evaluated in order; first the numbers 2 and 3 are pushed onto the stack. If you want to see stack at this point you could use the dump function which shows the entire state of the interpreter (on whatever device debugging output should go to):

```
2 3 dump
```

```
--->
stack: 2 3
thank you.
```

The + function is next. It pops the top number off the stack (3) and then the top one left after that (2). Althought the stack is now empty there could easily have been other items on it below those two and if so they would still be there. The + function then adds the two numbers and pushes the result, 5, on the stack. The next two items in the program, 4 and *, then push the number 4, then pop the numbers 4 and 5 and multiply them, giving 20. This goes back onto the stack and is then popped again and output by `print`.

Memory

Memory is addressed by numbers starting at zero. Each memory location may contain zero or more items in a list. Often a memory location will contain a single number. Memory may be written and read by the functions `w` and `r`, respectively:

```
1 0 w   ( 1 2 3 4 ) 1 w  dump
```

```
--->
stack:
0: 1
1: 1 2 3 4
thank you.
```

Here the `w` function takes the top item on the stack (its first argument but the one that appears second in the program, i.e., 0 and 1 for the two invocations of `w` here) as an address and the next item is written to memory at that address. The `r` function takes a single numeric argument as an address. So to write something to memory, read it back, and then print the result we could write:

```
5 3 w  3 r  print
```

```
--->
5
```

At this point the stack is empty but memory location 3 would still have the value 5.

Because memory locations can hold lists they may be exeuted as programs; this is the `dum` way to define a subroutine.

Program flow: `if` and `while`

Program flow is made possible by two functions, `if` and `while`. The `if` function tests its first argument and, if nonzero, executes the second argument:

```
( ( 1 2 3 4 ) print ) 1 if
```

```
--->
1 2 3 4
```

```
( ( 1 2 3 4 ) print ) 0 if
```

```
--->
```

In both examples the program to be executed is as folows:

```
( 1 2 3 4 ) print
```

which puts a list of four elements on the stack (as a single object) which the `print` function then pops back off and prints. Then either 1 or 0 is pushed on top. To see what the state of the interpreter is at this point we can type:

```
( ( 1 2 3 4 ) print )   1   dump
```

```
--->
stack: ( ( 1 2 3 4 ) print ) 1
thank you.
```

Now the `if` function is called. It pops the top object from the stack, 0 or 1,, and then the next object the list, `(1 2 3 4 ) print`. (The enclosing parentheses aren't actually part of the object, they merely frame it syntactically.) The value is tested, an if it is found to be nonzero, the list is executed.

(By the way, `dum` has no need for either a "pop" or an "eval" function; these are accomplished by writing `0 if` and `1 if`, respectively.)

The `while` function executes loops. Unlike `if` whose top argument must be a number, the two arguments to `while` are both arbitrary lists. The top argument is evaluated (passed to the interpreter). It must return a number by leaving it on the top of the stack. This number is then popped and tested. If it is nonzero the second argumnet is evaluated (this is the body of the loop). Then the test (the top argument) is re-evaluated and tested again, and so on. The second argument is not evaluated even once if the test fails on the first try.

[note to self - hmm, you don't really have to pop the two lists off the stack if you just protect the stack against the lists popping themselves while they're running - this will make context switching simpler later.]

So, for example, we can now count to ten:

```
0 0 w
    ( 0 r print   0 r 1 + 0 w )
    ( 0 r 10 < )
while
```

```
--->
0
1
2
3
4
5
6
7
8
9
```

Here a loop is initialized by writing 0 to memory location 0 (the first line). The second line is the body of the loop. It reads and prints the contents of location 0 in three words: `0 r print`. Then it increments the value (reads it, adds, one,

and writes the result back): `0 r 1 + 0 w`. The next line is the test: we read the value again and push 10. Then the logical operator `<` returns 1 if the value was indeed less than 10, and zero otherwise. These two lists (lines two and three) are the arguments to the function `while`.

We now have all we need to write computer programs. Behold:

```
2 0 w
(
    2 1 w
    1 2 w
    (
        ( 0 2 w ) 0 r 1 r mod 0 == if
        1 r 1 + 1 w
    )
    ( 1 r 1 r *  0 r  <=  ) while
    ( 0 r print )  2 r if
    0 r 1 + 0 w
)
( 0 r 50 < ) while
```

```
--->
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

List of functions in `dum`

Table 1 shows all the functions in `dum`. In addition to the ones we've used (and the several binary arithmetic functions that work the same way as the

name	args	return	function
`if`	A F	-	if F is nonzero evaluate A
`while`	B A	-	eval A; if nonzero eval B; repeat
`r`	F	list	read and return memory location F
`r1`	F G	number	read Gth item of memory location F
`w`	A F	-	write list A to memory location F
`w1`	H F G	-	write number H to Gth item of memory loc F
`+`, `*`, etc.	F G	number	F+G, etc.
`<`, `==`, etc.	F G	number	logical operations
`min`, `max`	F G	number	minimum and maximum of 2 numbers
`div`, `mod`	F G	number	integer division and modulus (remainder)
`print`	A	-	print a list
`scan`	-	number	scan a number from input
`wait`	F	-	wait F seconds
`dump`	-	-	printout for debugging

Table 1: Functions defined in `dum`. The last four are implementation dependent.

ones we've seen) two interesting functions appear: `scan` and `wait`. These are of course the most important functions, since they are the ones that are relevant to how an `dum`program will function in real time. Their effect will differ depending on what environment we are running `dum` in, and so we defer discussing them until we are in the context of one or another implementation, starting in the next section.

Implemtation of `f` as a command line program

A command line `dum` interpreter, in about 475 lines of c code, is available from `http://crca.ucsd.edu/~msp/syllabi/206.12f/f.c`. On a unix-like system this can be compiled by typing "`cc f.c`" to a command line. When run, `dum` repeatedly reads words from the standard input (the user can type the input of redirect it from a file or other process) and evaluates them.

The code is in three sections: first, data structures and stack manipulation routines; second, the built-in functions (with the system-dependent ones at the end); and third, a parser for text input and the main program.

At the heart of the command-line interpreter is the function `eval`, which scans through the items in a list and executes them. The items can be words or parenthesis-enclosed lists. If an item is either a list or a word that is a number, the item is pushed on the stack. If it is a word that is a built-in function, the function is called. The function can then pop items off the stack to use as

arguments and/or return values by pushing them back on the stack.

The `if` and `while` functions

Of the “pure” (non-system) built-in functions defined in `dum`, the two functions `if` and `while` are special in that they may in turn call `eval` (which, meanwhile, is already in the process of calling the built-in function itself.) This is an example of *recursion*, and any implementation of `dum` will have to be capable of recursion.

Fortunately, in the command-line `dum` interpreter, we can use the built-in recursive function calling capability of the c programming language to provide this. As we will see later, though, in other situations we may not be able to take advantage of this and so it’s worth looking in detail at how recursion is actually working in this implementation.

For the sake of brevity we’ll take the counting program above but only count to one:

```
0 0 w
    ( 0 r print   0 r 1 + 0 w )
    ( 0 r 1 < )
while
```

```
--->
0
```

In evaluating this, the `dum` interpreter ends up calling `eval()` recursively. For instance, at the first encounter of the `<` function on line 3, the state of the `dum` interpreter (as reported by inserting a `dump` function before the `<`) is:

```
stack: ( 0 r print 0 r 1 + 0 w ) ( 0 r 1 dump < ) 0 1
0: 0
```

and the C stack (reported by breaking on `fle` in a debugger) is as follows:

function name	relevant state in c stack frame
`main`	
`eval`	list to eval (`0 0 w ... while`) word being evaluated (`while`)
`f_while`	arg 1 (`0 r 1 dump <`) arg 2 (`0 r print 0 r 1 + 0 w`) arg being evaluating (arg 1) `dum` stack to restore (empty)
`eval`	list to eval (`0 r 1 dump <`) word being evaluated (`<`)

The `dum` stack and the c stack grow and shrink loosely in parallel; the `dum` stack changing constantly every time a word is encountered by the interpreter, and the relevant parts of the c stack more slowly as the texttttf_if(), extttf_while(), and `eval()` functions recursively call each other and return.

The items on the c stack represent whatever context the texttttf_if(), extttf_while(), and `eval()` functions will need to resume execution once control is returned to them after calling any subroutines in turn. The `eval()` function will need to know what list it was executing and the index of the next item it will deal with in the list it is evaluating. The list itself is either the entire program (for the topmost call to `eval()`) or else it is a list that was pushed on the stack for the `if` or `while` functions (`dum` functions, not c functions) to access as arguments.

As to the `while` `dum` function, something slightly impure has already taken place that needs explaining. In the `dum` implementation being described here, the texttttf_while() c function has artificially delayed popping its two list arguments off the stack. Logically those lists themselves should be in the stack frame belonging to the texttttf_while() call. However, we have chosen to simplify the `dum` interpreter slightly by leaving those two lists on the stack and remembering that we should have popped them. For this reason, we need one more thing in the texttttf_while() stack frame, showing what size the stack should be popped down to once we're ready to return.

References

[Abb81] C. Abbott. The 4ced program. *Computer Music Journal*, pages 13–33, 1981.

[Loy] D. Gareth Loy. Notes on the implementation of musbox: A compiler for the systems concepts digital synthesizer. pages 333–349.

[Mat69] Max V. Mathews. *The Technology of Computer Music*. MIT Press, Cambridge, Massachusetts, 1969.

[MCA+79] J.A. Moorer, A. Chauveau, C. Abbott, P. Eastty, and J. Lawson. The 4c machine. *Computer Music Journal*, pages 16–24, 1979.