§11. Excerpts from class, October 21 [notes by PMR]

Don opened class with the good news that Mary-Claire van Leunen has agreed to help read the term papers and drafts thereof, despite the fact that her name was incorrectly capitalized in last week's notes.

Returning to the subject of "Literate Programming," Don said that it takes a while to find a new style to suit a new system like `WEB`. When he was trying to write the `WEB` program in its own language he tore up his first 25 pages of code and started again, having finally found a comfortable style. He digressed to talk about the vicious circle involved in writing a program in its own language. To break it, he hand-simulated the program on itself to produce a Pascal program that could then be used to compile `WEB` programs. The task was eased because there is obviously no need for error-handling routines when dealing with code that you have to debug anyway. But there is also another kind of bootstrapping going on; you can evolve a style to write these programs only by sitting down and writing programs. Don told us that he wrote `WEB` in just two months, as it was never intended to be a polished product like TEX.

We spent the rest of the class looking at `WEB` programs that had been written by undergraduates doing independent research with Don during the Spring. We saw how they had (or had not) adapted to its style. Don said that he had got a lot of feedback and sometimes found it hard to be dispassionate about stylistic questions, but that some things were clearly wrong. He showed us an example that looked for all the world just like a Pascal program; the student had obviously not changed his ways of thinking or writing at all, and so had failed to make any use of the features of the system. The English in his introductory paragraph also left a lot to be desired.

Don showed us his thick book *TEX: The Program*—a listing of the code for TEX, written in `WEB`. It consists of almost 1400 modules. The guiding principle behind `WEB` is that each module is introduced at the psychologically right moment. This means that the program can be written in such a way as to motivate the reader, leaving `TANGLE` to sort everything out later on. [The `TANGLE` processor converts `WEB` programs to Pascal programs.] After all, we don't need to worry about motivating the compiler. (Don added the aside that contrary to superstition, the machine doesn't spend most of its time executing those parts of the code that took us the longest to write.) It seems to be true that the best way in which to present program constructs to the reader is to use the same order in which the creator of the program found himself making decisions about them. Don himself always felt it was quite clear what had to be presented next, throughout the entire composition of this huge program. There was at all points a natural order of exposition, and it seems that the natural orderings for reading and writing are very much the same.

The first student hadn't used this new flexibility at all; he had essentially just used `WEB` to throw in comments here and there.

A general problem of exposition arose: How are we to describe the behavior of a computer program? Do we see the program as essentially autonomous, "running itself," or are we participants in the action? Our attitude to this determines whether we are going to say 'we insert the element in the heap' or 'it inserts the element ... '. Don favours 'we'; at any rate one should be consistent.

Students used descriptors and imperatives for the names of their modules; Don said he favours the latter, as in ⟨ Store the word in the dictionary ⟩, which works much better than ⟨ Stores the word in the dictionary ⟩. On the other hand, where a module is essentially a piece of text with a declarative function—a list of declarations, say—we should use a descriptor to name it: ⟨ Procedures for sorting ⟩.

Incidentally, it is natural to capitalize the first letter of a module name.

One student used the identifier '*FindInNewWords*'. This looks comparatively bad in print: Uppercase letters were not designed to appear immediately following lowercase ones. Since the use of compound nouns is almost inevitable, `WEB` provides a neat solution. It allows a short underscore to be used to conjoin words like *get_word*. (Since the Pascal compiler will not accept identifiers like this, `TANGLE` quietly removes the underscore.) Don told us that Jim Dunlap of Digitek, who made some of the best early compilers, invariably used identifiers forty-or-so characters long. The meaning was always quite clear although no comments appeared.

Each module should contain an informal but clear description of what it actually does. A play-by-play account of an algorithm, a simple stepping through of the process, does not qualify. We are trying to convey an intuition of what is going on, so a high-level account is much more helpful.

We saw several modules that were much too long. Don thinks that a dozen lines of code is about the right length for a module. Often he simply recommended that the students cut the offending specimens into several pieces, each of more manageable size. The whole philosophy behind `WEB` is to break a complex thing into tractable parts, so the code should reflect this. Once you get the idea, you begin writing code this way, and it's easier.

We saw an example in which the student had slipped into "engineerese" in his descriptive text—all conjunctions and no punctuation. This worked for James Joyce, but it doesn't make for good documentation. One student had apparently managed to break `WEB`—the formatting of **begin**s and **end**s came out all wrong. Heaven knows what he did.

One student put comments after each **end** to show what was being ended, as **end** {**while**}. This is a good idea when writing Pascal, but it's unnecessary in `WEB`. Thus it's a good example of a convention that is no longer appropriate to the new style; when you change style you needn't carry excess baggage along.

Don had more to say about the anthropomorphization of computer systems. Why prompt the user with '`Name of file to process?`' when we can have the computer say '`What file should I process?`'? Don generally likes the use of '`I`' by the computer when referring to itself, and thinks this makes it easier for users to conceptualize what is going on. Perhaps humans can think of complex processes best in terms of demons in boxes, so why not acknowledge this? Eliza, the AI program that simulates a certain type of psychiatrist, managed to fool virtually everyone by an extension of this approach. Eliza may or may not be a recommendation for anthropomorphisms, or for psychiatry. There are those, such as Dijkstra, who think such use of 'I' to be a bad thing.

As in the case of maths, don't start a sentence with a symbol. So don't say '*data* assumes that . . . '—it can easily be rewritten.

We saw several programs by one student who had developed a very distinctive and (Don thought) colourful style. His prose is littered with phrases like "Oooops! How can we fix this?" and "Now to get down to the nitty-gritty." This stream-of-consciousness style really does seem to motivate reading, and helps infect the reader with the author's obvious enthusiasm. There were a few small nits to pick with this guy though: His descriptions could often be more descriptive. Why not call a variable *caps_range* instead of just *range*? Don also had to point out to him that 'complement' and 'component' are in fact two different words.

In `WEB` you can declare your variables at any point in the program. Don thinks it is always a good idea to add some comment when you do so, even if only a very cursory explanation is needed.

A note about asterisks: Be warned that typeset asterisks tend to appear higher above the line than typewritten ones, so your multiplication formulæ may come out looking strange. Better to use $\times$ for multiplication, and to use a typewriter-style font with body-centered '`*`' symbols instead of the '*' in normal typographic fonts.

Another freshman was digitizing the Mona Lisa for reasons best known to insiders of Don's research project. Don pointed out that since the program uses a somewhat specialized data structure (the heap) that might be unknown to the readers, the author should keep all the heap routines together in the text so that they can be read as a group while fresh in the reader's mind. In `WEB` we are not constrained by top-down, bottom-up, or any other order.

This student capitalized the first letter of every word in titles of modules, even 'And' and the like. This looks rather unnatural—it is better to follow the newspaper-headlines convention by leaving such words entirely in lowercase, and even better to capitalize only the first word.

Don thought it a good idea to use typewriter type for hexadecimal numbers, for instance when saying '`3F` represents 63'. But leave the '63' in normal type. This convention looks appropriate and provides a kind of subliminal type-checking.

The words used in the documentation should match the words used in the formal program—you will only confuse the reader by using two different terms for the same thing.

It's a good idea to develop the habit of putting your **begin**s and **end**s inside the called modules, not putting them in the calling module. That is, do it like this:

```
if down = 4 then ⟨ Punt ⟩;
    ⋮
⟨ Punt ⟩ =
    begin snap;
    place;
    kick;
    end
```

Not like this:

```
if down = 4 then
    begin ⟨ Punt ⟩;
    end
```

```
    ⋮
⟨ Punt ⟩ =
    snap;
    place;
    kick
```

Incidentally, appalling bugs will occur if we mix the two conventions!