Literate Programming as an Aid to Marking Student Assignments

by

A. J. Hurst

Department of Computer Science
Monash University
email: ajh@cs.monash.edu.au

Abstract

In a university climate that sees rising student-staff ratios and increasing workloads for academics, any mechanism that reduces the student marking and assessment process must be welcome. One possible path of improvement is to shift much of the repetitive nature of student assessment away from the academic, freeing up valuable time that can be spent more directly with students.

We argue in this paper that the Literate Programming paradigm (LP for short), first proposed by Knuth [1], can assist substantially in this assessment process. We detail some experience in using LP, firstly as a paradigm for student assignment submission, and secondly as a model to structure the semi-automated marking procedures. An electronic submission procedure is used, in which students' work is handled automatically by scripts which annotate the literate program, and extract key components for testing against model answers. Although initial experience is limited in the style of analysis that can be used, we conclude that the LP paradigm does offer some significant advantages to the mundane activity of student assignment marking.

Introduction

Of all the tasks facing an educator, student assessment seems to be one of the least popular! There is something about the repetitive nature of the task, and the frequently pedestrian and/or confused ideas presented by the student, that leads assignment and examination marking to be viewed with some distaste by university educators. Colleagues frequently complain of the "mind-numbing" effect of long hours of marking, the more so in the computing field it seems, because of the necessity to also scan programming code as part of the assessment process.

We report here some trials in using the Literate Programming (LP) paradigm as an aid to relieving this mental torment. We can identify several aspects for which LP is helpful.

1) Students are required to write their assignments as literate programs.
2) Assignments presented as LPs can be handled completely electronically through mail-based submission systems.
3) The use of document typesetters as the back-end to LP processors means that a common format of assignment presentation can be used.
4) Such a common format allows automatic handling, even marking, of student assignments.
5) The automatic marking can be undertaking by systems themselves written as LPs.

We explore and analyse each of these aspects here.

Literate Programming

While the idea of LP is not new, it does not seem to have developed or spread widely. Knuth's original paper [1] appeared in 1984, yet very few examples have appeared in the public domain, nor has the technique been adopted in many institutions. Knuth's original system was based upon the programming language Pascal and the document typesetting system TEX, and was in fact used to write bootstrapped versions of TEX itself. The WEB system comprised two programs, known as 'tangle' and 'weave', of which more anon.

The interested reader is referred to an extensive bibliography [2] for detailed discussion of literate programming ideas, but a brief introduction in this context is perhaps appropriate.

The concept of literate programming shifts the focus of documentation from "code with comments interspersed" to "documentation with code interspersed". That is to say, the author of the program addresses primarily the human reader, and structures his document to meet the needs of the reader first and the computer second, rather than structuring his program to meet the demands of a compiler first, and the reader as an afterthought.

Literate programs contain both code and documentation, but it is the latter to which the 'program' defaults, rather than the former. Code and documentation fragments make up the literate program, usually in alternation, but by no means necessarily so. For the purposes of compilation, a filter is applied to the program, and this filter writes all the code fragments to one or more files. This filter is known as the 'tangle' filter, after the program written for this purpose in the original WEB system. Code fragments may refer to other code fragments, and a process of macro expansion takes place to ensure that the fragments get 'tangled' together in the order required by the compiler, even if not the order in which they appear in the program.

A separate filter (although sometimes combined with the 'tangle' filter) program scans the literate program document and extracts a file suitable for document processing. This includes the code fragments suitable typeset. Such filters are known as 'weave' programs.

Some literate programming systems 'prettyprint' the code before typesetting, but this requires the 'weaver' to have some knowledge of the code syntax rules, and binds the 'weave' filter to a specific language. Because of this constraint, more recent literate programming systems adopt a 'no prettyprinting' rule, which allows them to be used with arbitrary programming languages. However, the document processing system, being less of a constraint to the use of the system, is usually tightly coupled to the system. TEX and LATEX are the document processing systems most widely used in this context.

The processed document then admits of the full typesetting abilities of the document processing system, allowing tables, mathematics, figures, tables of contents and indices all to be automatically gnerated. These greatly enhance the readability of the literate program. Code fragments also may be indexed and cross referenced, and are usually typeset with fragment number or page references to embedded macro calls.

Figure 1 shows a fragment from one of the marking scripts referred to later in this paper. The literate programming system used is `nuweb`. Macro calls are shown enclosed in angle brackets, together with the page number on which they are defined. Lowercase letters are used to resolve multiple definitions on a single page, and `?` question marks are used to indicate macros not defined (since this is only an excerpt of a larger document).

Automatic Submission

In order to automatically mark student assignments, some form of automatic submission scheme is required. This allows a student to lodge his code with a program that can do one or more of the following things:

- vet the code for basic attributes;
- record date, time, subject name, assignment name and other administrative details;
- mail the code to a remote system;
- compile the code or perform other appropriate processing;
- execute the code against some appropriately chosen test data, preferably in a secure environment; and
- evaluate the test results.

The first two of these are required for the purposes of recording the submission. Essential to this project are the last three, to which we must add two additional requirements:

- 'tangle' and 'weave' the literate program to extract the source code for compilation and the documentation for typesetting.
- typeset the documentation.

Let us consider each of these in turn.

It is essential to vet the submission at the outset. We are dealing with students who are still at various stages of the learning process, and they may not have adequately prepared themselves to use the submission system. They may have misunderstood instructions, and key wrong values into the system. The automatic submission system must give as much feedback to the student as possible at this stage in the submission process, so that the student is fully aware of just what has happened to his code. Particularly where marks are involved, some students may become quite upset and/or panicky if thet feel that their work has not been properly processed. Those who do not have confidence in the system will repeat their attempts, which may prove disastrous in the situation where the initial problem was caused by lack of disk space!

Some students may behave maliciously towards the system, in order to break it or otherwise upset the automatic process as much as possible. Their attempts must be monitored and rejected as appropriate, and audit trails left where possible, so that in the case something does go wrong, sufficient information is available to improve the robustness of the system.

The submission process must record as much (relevant) information as possible, so that issues of deadlines, number of submissions, identity of the submitter, etcetera, are available not only for the marking process, but also to investigate complaints as and when they arise. One cannot rely upon the honesty of students!

The need to mail the submission arises not only because the systems responsible for developing the code (the student system) and processing the code (the marking system) are generally part of some form of local area network, but also because of security concerns. Separation of systems in this way decreases the risk of breakins.

If the student code is to be evaluated, some mechanism for compiling and executing the received submission is necessary. This would not normally be a problem, except that a) the student may be using a different system to develop and test his code, and subtle system dependencies may surface at this stage, and b) one must be careful to execute student programs in a secure environment, where accidental or malicious program misbehaviour may compromise system security.

To this end, we have developed a `submitroot` program, which allows execution of a student program in a (unix) context where the only file systems accessible are the directory in which the executable code resides, together with a subdirectory containing any required libraries or other executables.

The behaviour of the submitted code under compilation and execution may then be assessed by running it against a number of test scripts, and examining the output to see if it conforms to the assignment specification. One advantage of such an environment is that the student will be quite unaware of the nature of the test data, and this greatly enhances the quality of test, to say nothing of the need to be accurate in one's exercise design! For example, test data can be chosen randomly, or generated on the spot, thereby forcing accurate adherence to the assignment specifications.

We turn our attention to the need for the two additional requirements in the next section.

Assignments as Literate Programs

It would be fair to say that documentation of assignment programs by students is one area that receives scant attention. Not only is the process of program documentation poorly understood at the best of times, but also the task is compounded by the fact that the student is struggling to realize his own understanding of just how the program should be designed and coded. Students often get no tuition in how to document programs, or, if they do, it is overlooked as soon as attention is directed to some other area of programming. In these circumstances, it is little wonder that documentation becomes something that is done at the last minute, and is done scrappily at that. This is not a good model to be giving students!

With these factors in mind, and following the challenge offered by Hamer [3], it was decided to introduce literate programs as first the 'preferred', and then the 'mandatory' form of student submission for assignments in two third year courses taught by the author. The fact that one of these subjects, a course in Compiler Construction, used the Eli system developed by Waite et al. [4], which allows compiler specifications to be written in FunnelWeb [5], a literate programming system was an added impetus. As Hamer states: "... there is no 'obvious' course in which to include a topic on literate programming." [3],p286. Eli gave just the opportunity to respond to that challenge.

Experience to date has shown that the general quality of student submissions has risen as a result of using literate programming. One dimension of this quality is simply that resulting from the use of a document processing system being forced upon the students. Previously, the author had tried to move in this direction by asking students to submit work that had been typeset by any document processing system, WYSIWYG or markup, or at the very least typewritten; but the variability between layouts and print quality, together with the variety of skill levels demonstrated by students, meant that the overall quality of program documentation was just as wide (with just as extreme end points) as it had ever been.

A 'skeleton' literate program is made available, which is used by students as a starting point for constructing their solutions. The advantage of this is that it can be as sketchy or as detailed as one likes. The framework allows hints to be given, and may be used to structure the design to be used by the student. Figure 2 shows one such example.

One emphasis presented to students is that they are encouraged to 'make their literate programs tell a story to the reader'. This paradigm seems to strike a chord with some students, who relish the opportunity to narrate their way through a program.

It must be admitted that there are students at the other extreme, who submit literate programs that have just one macro (the entire program code) and little or no documentation. Addressing this failing is the topic of the next section.

Marking the Literate Programs

The use of literate programs as the vehicle for student submissions not only improves the pedagogic process from the student's perspective, but it can greatly assist that process from the lecturer's point of view as well.

At a basic logistic level, the use of literate programming means that everything the student needs to submit as part of his assignment solution can be present in the one file. All code, all documentation, including the ability to typeset tables, mathematics, even include figures, is available through various mechanisms.

Of course, some of these tools require additional investment on the part of the student, so these mechanisms must be used with care. However, students as a whole seem keen to learn their use, and this is appropriate for budding computer professionals. Knowledge of such tools as TEX, LATEX, `xfig`, Postscript, etc., or their equivalents, is seen by many computer science educators in a somewhat equivocal light. On the one hand, they are regarded as essential components of a computing professional's knowledge, but on the other hand, educators may be reluctant to devote curriculum time to teaching them. Using them as part of the exercise submission cycle can resolve both of these points of view.

Because the material is submitted entirely electronically, there is no paper to handle. Experience has shown that paper submissions often go astray, and the department has had to institute various auditing processes to ensure that the quality of service to the student does not degrade as the number of separate assignment scripts handled per academic rises. With electronic submission, such procedures are reduced.

So how is the marking handled? The system developed so far uses shell scripts to perform some basic tasks, such as compilation and execution against test scripts. Where the style of the exercises admits, marks can be automatically accumulated for each test passed. Upon completion of the submission processing, a summary is mailed back to the student, together with the mark realized. Depending upon the tests failed, the script can also generate various hints as to what the student has done wrong.

Students appreciate this immediate feedback, and are allowed to rework and resubmit material until they are satisified with their efforts, or they run out of time. Both of these points encourage students to submit earlier in the assignment cycle, rather than the last possible moment. On the one hand they can see immediately where they are failing, rather than having to wait until the deadline has passed, the lecturer has marked the material, and it has been returned to the student. By then, the immediacy of the problem has gone, and the student is often no longer interested in just what issue caused him to lose marks.

On the other hand, submitting earlier maximises the amount of time they have to perfect their work, and the use of automatic testing and hint generation leads the student to focus directly upon her error.

The literate programs are 'tangled' and 'woven', and the processed material stored until the lecturer is able to view it. I use scripts to automatically cycle through the collection of submitted material, so that as each student's literate program is displayed, another window shows the marks recorded automatically, and the lecturer can satisfy himself that the marking procedure is behaving appropriately.

Alternatively, some things in an assignment may not lend themselves well to automatic marking. For example, in one of the subjects taught by the author, students are required to write Z schemas. These are difficult to mark automatically, since there are issues of layout, style, and a variety of formulations to contend with. The approach used here is to assess the students work by visually scanning the material, but here again, the fact that the material is on-line means that scripts can be created to cycle through the collection of student submissions, together with a suitable script that prompts for each mark. As marks are entered according to how the literate program reads, the script prompts for the next marking point, which facilitates the marking process.

One approach tried in this respect is to scan and annotate the student's literate program, adding highlight mechanisms that direct the eye to each salient point. This process is aided by making available a literate programming template, which the student can edit to reflect his work. Appropriate headings or text markups can be used to direct the script to annotate the relevant section of text. In Figure 1, the macro ⟨Annotate the student's literate program 3b⟩ does this task, using a perl script to perform the pattern matching and annotation. It relies upon key text fragments in the skeleton (such as **2.2 Explanation of *stringToInt* Schema** in Figure 2) to add appropriate annotation. Figure 3 shows the result of annotating part of Figure 2, as applied to an actual student submission, as evidenced by the errors in the text! Marginal notes are used to draw the marker's attention to the key points. The leading number indicates how many marks the highlighted fragment carries.

The use of shell scripts, perl scripts, test files, etc., tends to lead to a proliferation of files in creating a suitable marking environment. What better way to manage all these than to write the marking script as a literate program itself. This has the advantage that all the scripts are kept in one coherent whole, together with appropriate documentation. The marking script then becomes easier to debug and maintain.

One reason for this is that a consistent environment may be reestablished by 'tangling' the literate marking program: this rewrites all files to ensure that they are in a consistent form. Design decisions made in the development of the marking script are recorded for posterity: this is particularly useful when the program is redesigned to handle the next exercise, whether it be next week, next month or next year.

Towards an Automatic Marker

The system as described above is not automatic. The lecturer still has to construct the scripts, which, while having some commonality with each other, are still highly tailored towards the specifics of a particular exercise. We would like to move a step beyond this, and have much of the tedium of such script writing removed from the onus of the lecturer.

A major impediment to the more widespread use of marking scripts is this need to redevelop the marking scripts for each exercise. Shell scripts (or their equivalent) require a certain level of programming skill that some users would not wish to acquire, and if we are to realize the goal of automatic marking in general, some automation of script generation is essential.

To do this requires some notational mechanism in which to describe the marking task. To date little work has been done to develop this aspect, but several ideas present themselves. One approach would be to develop a syntactical description of an exercise outline, and use a translator tool such as Eli to firstly ensure that the submission is syntactically correct, and secondly to perform semantic transformations on the submission so that it can be handled by more primitive marking scripts.

Another alternative might be to develop a notation that abstracts over the essential tasks in building such scripts: shell tasks such as compilation and execution; maintenance, selection, and automatic generation of test files, together with appropriate outputs; comparison and evaluation of test results; etc.. 'Marking' would then become a process of building upon such tasks, perhaps by using some form of programming notation to define how these tasks are composed and ordered.

Conclusions

We have described a system which is in use in the Department of Computer Science at Monash University. The system allows automatic submission of student exercise solutions, and to some extent automates their marking.

Central to the philosophy of this process is the use of literate programming, which enforces an appropriate paradigm suitable for automatic marking upon the students. The design and maintenance of the automatic marking scripts themselves is enhanced and supported by the use of literate programming to construct them as well.

Experience in the use of this system indicates that there is a fairly high overhead in creating and setting up the various scripts. For single courses, or small numbers of students, or marking requiring substantial interaction, this effort makes the process of marginal benefit. However, for large or repeated courses, or where there is a high degree of mechanisation to marking, the effort certainly facilitates the learning process.

As yet, we do not have mechanisms to support the automatic generation of the marking scripts. We believe that with further research, some progress might be made towards this goal, and thereby make the overall process of student exercise marking more tractable.

References

1. Knuth,D.L., Literate Programming, *The Computer Journal*, 27(2):97-111 (1984).
2. Beebe, N., Literate Programming Bibliography, available from URL: `ftp://ftp.math.utah.edu/pub/tex/bib/litprog.{bib,ltx,twx}`.
3. Hamer, J., Literate Programming: A Software Engineering Perspective, *Software Education Conference (SRIG-ET'94)*, University of Otago, New Zealand, pp282-288, November 1994. (published by IEEE Computer Society Press, ISBN 0-8186-5870-3)
4. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. and Waite, W. M., Eli: A Complete, Flexible Compiler Construction System, *Communications of the ACM*, 35(2): 121-131 (February 1992).
5. Williams, R., The FunnelWeb User's Manual. Available for anonymous ftp from URL: `ftp://ftp.adelaide.edu.au/pub/funnel-web`, or any Comprehensive TEX Archive Network (CTAN) site, May 1992.

The Submission Script

This uses the `submit` program written in the Department of Computer Science at Monash. The `submitr` program used to register a submission relies upon a script embedded in the database file `submitrc`. That script executed is this program, `submitscript`. This script gets executed in the context of the student's submit directory, hence we must be careful to copy in all necessary files for the mark script, and to clean up afterwards.

This script is called with three mandatory parameters: the nuweb file name submitted, the student ID number, and the student's user name.

```
"submitscript" 3a ≡#! /bin/sh
   ex1dir=$HOME/Submit/csc3080/exercise1
   latex=/usr/local/bin/latex
   nuwebfile=$1; studID=$2; studlogin=$3
   rm ex1-output >/dev/null 2>&1
   ⟨Annotate the student's literate program 3b⟩
   cat ex1-errors
   ⟨compile the student's code ?⟩
   ⟨test the student's code ?⟩
   ◇
```

Annotate the Student's Literate Program

This script does not do much marking. The human marker is still require to scan the submitted text and assess the result. To assist in this process, we annotate the literate program with some marking comments, using the `\marginpar` mechanism of LaTeX. A perl script is used to do the annotation, and the resultant markup is processed through `nuweb` and `latex`.

```
⟨Annotate the student's literate program 3b⟩ ≡
   perl perl-file <$nuwebfile >ex1-annotate.w
   nuweb ex1-annotate.w >>ex1-output 2>&1
   $latex ex1-annotate.tex </dev/null >>ex1-output 2>&1
   ◇
```

Macro referenced in scrap 3a.

The perl script breaks into two parts: those annotations which must appear before the matching input line, and those which appear after the line.

```
"perl-file" 3c ≡#! /usr/monash/gnu/bin/perl
   ⟨initialize perl script ?⟩
   while (<STDIN>) {
      ⟨annotate question 1 ?⟩
      ⟨annotate question 2 ?⟩
      ⟨annotate Literate Programming ?⟩
      print $_;
      ⟨annotate question 1.1 ?⟩
      ⟨other stuff not shown ?⟩
   }
   ⟨Check and report on missing sections ?⟩
   ◇
```

Figure 1: Sample literate program excerpt.

2.1 *TEAM* versus `Team`

Include your explanation here.

2.2 Explanation of *stringToInt* Schema

$$\begin{array}{ll} \mathit{stringToInt} : \mathrm{seq}\, \mathit{CHARACTER} \rightarrow \mathbb{N} & [1] \\ \hline \forall\, s : \mathrm{dom}\, \mathit{stringToInt} \mid & [2] \\ \quad (\mathrm{ran}\, s \subseteq \{\text{`0'} \mathrel{..} \text{`9'}\}) \wedge (k = \#s) \bullet & [3] \\ \qquad \mathit{stringToInt}(s) = \sum_{i=1}^{k} (s(i) - \text{`0'}) * 10^{k-i} & [4] \end{array}$$

Complete the following lines.

1) This line says ...
2) This line says ...
3) This line says ...
4) This line says ...

3 Exercise 1.3

Include your discussion about the *Ladder* class here.

⟨Ladder class definition 6⟩ ≡

```
class Ladder {
public:
  ? teams;
  ? initialize(?);
  ? sort(?);
};◇
```

Macro never referenced.

Figure 2: Skeleton Literate Program issued to students (excerpt)

2.1 *TEAM* versus `Team`

1 Team v Team

The [TEAM] set is a data type containing irrelevent variables and data structures (e.g. player's names, pay roll, what ever) where as the class Team is just concerned with the stats that affect it position on the ladder (e.g. wins, loses, etc).

2.2 Explanation of *stringToInt* Schema

2 Explain *stringToInt*

$$\begin{array}{ll} \mathit{stringToInt} : \mathrm{seq}\, \mathit{CHARACTER} \rightarrow \mathbb{N} & [1] \\ \hline \forall\, s : \mathrm{dom}\, \mathit{stringToInt} \mid & [2] \\ \quad (\mathrm{ran}\, s \subseteq \{\text{`0'} \mathrel{..} \text{`9'}\}) \wedge (k = \#s) \bullet & [3] \\ \qquad \mathit{stringToInt}(s) = \sum_{i=1}^{k} (s(i) - \text{`0'}) * 10^{k-i} & [4] \end{array}$$

1) This line says that string to int turns CHARACTER to natural numbers
2) This line says that for all s the domain of string to int is
3) This line says that the range of s must be between '0' and '9' and that k = number of digits in s (e.g. if s = 564 k = 3)
4) This line does the working out translated to C : for (i = 1; i ¡= k; i++) number = number + (s(i) - '0') * pow(10, k-i); simple huh - it takes away the acsii value for 0 which gives a number then the place the number is in is worked out e.g. 1 or 10 or 100 or 1000 etc.

Figure 3: Result of Annotating Figure 2 (excerpt)

(The effective page width has been shrunk to show the marginal annotations.)