AUTOMATIC MARKING OF SHELL PROGRAMS FOR STUDENTS COURSEWORK ASSESSMENT

AYŞE SALMAN

School of Computing and Mathematical Sciences
Oxford Brookes University

This thesis is submitted in partial fulfillments of the requirements for the degree of Doctor of Philosophy of Oxford Brookes University
December 1999

Acknowledgements

I am indebted to Mrs. Nilüfer Gözaçan Borahan, the General Coordinator of Doğuş Institute of Higher Education / Oxford Brookes University for her invaluable support and kind help in completing my thesis.

I am also grateful to my supervisors, Prof. Reza Ziarati, the Academic Director of Doğuş Institute of Higher Education / Oxford Brookes University and Mr. John Nealon, the Head of the School of Computing and Mathematical Sciences of Oxford Brookes University for their invaluable source of encouragement, support and guidance during my career as a research student.

I would also like to thank Ms. Sue Greenwood of the School of Computing and Mathematical Sciences, Oxford Brookes University for her support throughout my studies.

To the memory of my father, Dr. Osman Baskol

and

the twins, Ramiz and Harun

ABSTRACT

The number of students in any programming language course is usually large; more than 100 students is not uncommon in some universities. The member of staff teaching such a course has to mark, perhaps weekly, a very large number of program assignments. Manual marking and assessing is therefore an arduous task.

The aim of this work is to describe a computer system for automatic marking and assessment of students' programs written in Unix Bourne Shell. In this study, a student's program will be assessed by testing its dynamic correctness and its maintainability. For dynamic correctness to be checked the program will be run against sets of input data supplied by the teacher, whereas for maintainability the student's program will be tested statically. The program text will be analysed, and its typographic style and its complexity measured.

The typographic assessment in this system is adaptable to reflect the change of emphasis as a course progresses. This study presents the results generated from the assessment of a typical class of students in a Shell programming course. The experience with the development of the typographic assessment system has been generally positive. The results have shown that it is feasible to automate the assessment of this quality factor, as well as dynamic testing. Realistic grading can be achieved and useful information feedback can be obtained. The system is useful to both the students learning programming in Shell, (Arthur, L. J. and Burns, T., 1996) and the staff who are teaching the course.

Although the work here is focused on the Bourne Shell, (Bourne, S. R., 1987) the study is still valid, with little or no change, to all other shells. The method used can also be applied, with some modification, to other programming languages. Furthermore this method is not limited to university and teaching, it can also be used in other fields for the purposes of software quality assessment.

TABLE OF CONTENTS

CHAPTER I

AIMS, OBJECTIVES AND RATIONALE

The main research question can be summarized as:

'Can students' Bourne Shell programs be assessed typographically, and can they automatically mark their programs?' This question has been transformed into an overall aim:

To provide an automatic assessment system for Bourne Shell programs written by students.

To achieve the stated aim, the following objectives were formulated:

- To select the factors which affect the program typographically; such as commands and indentation.
- To calculate overall typographic mark, establish student performance.
- To consider if complexity can be measured.
- To check whether the program is correct dynamically and to carry out tests for several cases.

1. Computer Science Education

An important goal in the development and improvement of computer science education is to improve the process of evaluation and assessment. Computer Science

Departments and Schools of Computing are continually revising their range of courses and also the assessment systems used so that they can adapt to the ever increasing requirements of technology oriented education. In general, the uses of computer technology in education can be divided into three categories.

- Multimedia and Interactive teaching.
- Collaborative Learning Environment.
- Course Assessment and Management Systems.

It is widely recognised that class sizes on computing courses are increasing year on year. This has some detrimental effects on the students as a direct result of the increased workload on tutors (Benford *et al.*, 1995). For example, due to the additional demands of assessment and marking less of the tutors's time is available for quality feedback to students. This may also lead to inconsistencies in assessment between different groups of students as tutors are forced to deal with the inherent problems in different ways (Benford *et al.*, 1995).

As stated, the overall aim of this work is to provide an automatic assessment system for Bourne Shell programs written by students. Providing a system for automating the marking of this work is much sought after by university staff. Programming languages are usually taught to classes with large numbers of students. Without computer assistance, marking students' programs and assessing their progress is a very time consuming process. The use of computer automation has the additional advantage of maintaining a consistent standard in the marking process (Zin and Foxley, 1992).

A student's program can be assessed by measuring its quality (Benford *et al.*, 1993). There are many factors that can be attributed to program quality such as correctness, efficiency, and maintainability. As a student's program is likely to be revised several times for the purposes of either correction or improvement, it must be maintainable. For maintainability to be checked, the student's program text should be analysed for the aspects of quality that assess its maintainability (Sommerville, 1996). It therefore seemed appropriate to apply a "hands-on" approach to assess quality and develop a system which encourages students to be attentive to the style and accuracy of their software.

There has been a wealth of research since the 1980's on the use of computer technology in education. Also much work has been done on typographic style in particular (Baecher and Marcus, 1998; Berry and Meekings, 1985; Leinbaugh, 1980; Miara *et al.*, 1983; Oman and Cook, 1990; Oman and Thomas, 1990; Shneiderman, 1980). There is an immense diversity in these various works on assessment that make them hard to categorise. Nevertheless, the following three main categories can be differentiated, though much of the work reviewed combines more than one category. Chapter III covers these in more detail.

- On-line interactive course delivery and learning.
- Examinations and test assessment.
- Courseware management and assessment.

The work done in this thesis belongs to the final category mentioned above. Two important technological developments were heavily exploited in many of these works. These were the use of networks - primarily the Internet, and the introduction of dynamic paradigms or animation for teaching/learning (Herbert and Brumund, 1998; Marshall, 1997; Naps and Biessler, 1998; Pierson and Roger, 1997; Stasco, 1997).

2. Proposed Assessment System

The system described in this thesis has been developed to measure the quality of Unix Shell programs. The core of the system is an automatic assessment facility which tests students' computer programs for specific aspects of quality. Here, only two attributes of a program are being measured: its maintainability, and its dynamic correctness. Firstly, for students to achieve any experience in programming they should produce dynamically correct programs that both run effectively and give correct solutions for the problems being tested. That is the reason for choosing dynamic correctness. In order to check dynamic correctness, a student's program should be run against sets of input data supplied by the teacher, whereas for maintainability the program is tested statically, analysing a student's program text for certain aspects of quality. The following statements explain why maintainability is selected. Quality assurance is an integral part of the software engineering process (Benford *et al.*, 1995). Students should repeatedly assess the quality of their programs in order to make improvements. By doing this they also get practice in time and work management, as they must meet the deadline set for the exercise. Here two aspects of the quality of a program will be assessed: its typographic style and its complexity.

This system effectively promotes awareness of quality control issues. Students check and correct the style of their programs and focus on obtaining better results. This is the reason for choosing maintainability as an attribute to measure in addition to dynamic correctness. It is essential that students learning how to write different programming languages are taught to style their programs typographically and improve their presentation (Kernighan and Plauger, 1978; McConnel, 1993). The style and presentation of students' programs thus improved following the use of this assessment system.

The automated typographic assessment of Shell programs has provided satisfactory results (Kernighan and Plauger, 1978; McConnel, 1993). As students became more experienced, they obtained better typographic results, and reached towards optimal values. tutors and lecturers may change the assessment weighting according to the requirements of the exercise. As students completed more exercises, there was a demonstrable improvement in the typographic style of their programming. A more detailed discussion can be found in Chapter VII.

3. Thesis Structure

The remaining structure of the thesis is as follows.

Chapter II introduces the technology used in computer science education, software quality, software metrics, and program quality assessment.

Chapter III is a review of work relevant to the use of computer technology in education. Much of the work reviewed can be categorized as:

- On line interactive course delivery and learning.
- Examination and test assessment.

- Courseware management and assessment.

Chapter IV aims to describe the automatic grading method that is based on the typographical analysis of Shell programs, assuming that they are structurally and syntactically correct.

In Chapter V complexity, which is another measure for the assessment of the quality of Shell programs is considered. There are different categories of complexity measures. These sets of measures cover the properties that are usually considered when teaching students good programming practice.

In Chapter VI an automatic system for dynamic testing and subsequent grading of the correctness of Shell scripts is discussed. In this scenario, the teacher will provide the system with input test data in addition to output data. Using the same input data, the student's program is graded by checking its output data against the teacher's.

Chapter VII outlines the assessment system, explains how the assessment is done, and discusses the results of the typographic assessment and interpretation of the results.

Chapter VIII presents the conclusion. The main achievements of this thesis are examined, focusing on the development of a system which combines of three methods for assessment:

1. typographic analysis,
2. complexity measurement and,
3. testing of dynamic correctness.

The aim is to produce an effective assessment of students' programs. Any results obtained in the application of these methods are further discussed. Issues not covered by this thesis and suggestions for future development are addressed in this chapter. It should be noted that this thesis does not attempt to cover two otherwise important issues. The first is the evaluation of the system in a real education environment. The second is the availability of a user-friendly interface for the system.

CHAPTER II

INTRODUCTION

In order to improve Computer Science education, different systems are being designed and implemented. The World Wide Web is used as a platform to support course material; the integration of both internet and multimedia tools are being widely developed to support teaching. However, the other important issue is assessment. As class sizes become large in the teaching of computer science courses, assessment becomes more arduous. In order to assess a student's programming a measure of quality is made. Here, the quality of software programmed by students is being examined. Some of the aspects of software quality factors are correctness, efficiency, usability, and maintainability. Software metrics is a means of measuring software quality. The quality attributes which are being checked in this case are correctness and maintainability. Program quality assessment tools help students improve their own software quality, provide feedback to both the student and the teacher, and finally introduce time-saving benefits for both parties.

1. Technology in Computer Science Education

Research within Computer Science education is aimed towards the study, development and improvement of education in Computer Science through the use of scientific methods and technology. The improvement of the process of evaluation and assessment in particular, through the integration of computer technology, is one important objective.

Computer courses and education are developing and expanding in all directions. But studying and teaching computer courses did not imply the use of computer technology to support the computer education itself. Robotics, image processing, internet technologies, programming languages are a few examples.

Only recently, in the past ten to fifteen years, has the potential of using computer technology to support education, and in particular in computer science education, been realised (Benford *et al.*, 1995). There have been changes in higher education around the world which have prompted such use. Notable among these changes are:

- increases in student numbers and class sizes,
- the diversity of students' academic background.

Information Technology, especially, addresses many of the issues raised by these and other changes in education. As a consequence, Computer Science Departments, and Schools of Computing are revising their availability of courses and also assessment systems so that they can adapt to technology-oriented education. This includes, for example: modern delivery methods that allow teachers and students to schedule the place and time of learning; support for teaching strategies such as collaborative learning and assessment; and visualisation techniques that enable teachers to make explanations in a more dynamic fashion.

In general the uses of computer technology in education can be divided into three categories:

- Multimedia and Interactive teaching/learning.
- Collaborative Learning Environment.
- Course Assessment & Management Systems.

In order to improve Computer Science education, different systems are being designed and implemented. Some of the examples use the World Wide Web as a platform to provide course material, satisfy the module syllabus and provide a means of measuring a student's level of understanding, and tutorial interaction. Universities are developing software to allow students to add content to the Web, with no knowledge of HTML, or the use of web page generation tools (Inman *et al.*, 1996). This is being developed to encourage students to utilise the recommended course material as much as possible. Students select a reference to review; their tutor helps in the decision process. This has been useful for students who are taking a course and who can then continue to study a subject in more depth; it is also useful as an inspirational tool for new students.

Nowadays, the provision of courseware has taken a different direction. Courseware does not exist as a single discreet package; by the use of the internet and accompanying browsers, there are multimedia tools available which can be integrated into a www browser. For instance, at the University of Cardiff, courseware has been developed to support the teaching of C programming, X windows, Parallel Processing and so on by using the www (Allum, 1996).

Assessment is an important issue for universities. As class sizes become large, manual assessment becomes extremely ineffective and time consuming. If what is required is a high quality assessment system, it should also be one capable of providing feedback. This feedback should not only be sufficiently appropriate and detailed, but its results should also be quickly available so that students can make their revisions and corrections as soon as possible. In addition, the system should have a grading facility, which provides an accurate overall grade as well as information that identifies a student's weak areas. As a result of the dramatic increases in student to staff ratios in universities, the provision of such a system has become even more difficult to achieve without computer assistance. This is particularly important when it comes to the assessment of coursework for students in programming language courses. Since the number of students in any programming language course is usually large, the staff member responsible for teaching such a course has to mark a large number of program assignments, which on a weekly basis is very time consuming.

The quality of a student's program is the key to its assessment. Before such quality is measured, a clear understanding of what is meant by program (or software) quality is required.

2. Software Quality

Software quality is the study of the numerical measures that can be applied to the products and processes of software development. Although the quality applies to both the software process and the software product, interest here is reserved for the *software product quality*. Hetzel (1998) says that the elements of quality are not constant. The quality of a product in one environment may be different from, or unsuitable for, another. "Meeting requirements" seems to be more meaningful definition of quality according to Hetzel (1998). The "meeting of requirements" depends on the criteria selected. Quality is formed from a set of factors that guide the expectations of software.

Software quality can be defined using the following criteria:

- Software requirements are the base from which quality is to be measured.
- There are certain specified standards for development
- There are implicit requirements like maintainability

Cavano and McCall (1978) state that software quality factors can be categorised as three groups. Each group focuses on a certain aspect of the product. (in this case it is a student's program). These are:

- Ability to make changes. This includes factors such as maintainability, flexibility and testability.
- Adaptability to new environments. This group covers portability, reusability and interoperability.
- Operational characteristics. This group includes factors such as correctness, reliability, usability, integrity and efficiency.

These are a set of qualitative factors for the measurement of software quality. Some of them can be measured directly, some cannot. Cavano and McCall (1978) state that such measures are subjective and therefore for objective analysis, quantitative measurement of software quality is needed. For instance, measures such as reliability, correctness, usability and maintainability are not quantitative measurements; they cannot be measured directly. Ideally these need to be quantified through the use of quantitative measures. This can be achieved indirectly; for example, in order to predict maintainability, which is a qualitative measure, complexity measures can be used. The relation between quantitative and qualitative measures is the relationship that exists between what can be measured and the information which is sought. Thus, if information about maintainability is needed, its complexity can be measured.

Nevertheless, a common criticism of software quality is that it's collection it is too hard, too time consuming and also that metrics values might be difficult to interpret. To a certain extent, some of these complaints are justified, not least because there is no universal quality. However, by using a combination of a variety of metrics some form of measure, rather than simple estimate, to judge quality can be made.

The factors proposed by McCall as mentioned above are comprehensive. However the relative importance of different factors may vary in different environments and some might not be appropriate to individual stages in the software life cycle. In this present study of student program quality the only relevant factors are correctness and maintainability. Correctness is of course very important because the main aim is to produce a program that can run and produce the correct results. Maintainability is also very important because in an educational environment a student's program will change and be modified several times before being submitted in its final form. The cost of maintenance activities in developing a software engineering project currently forms a very high proportion of its total cost.

3. Software Metrics

A software metric (or software measure) is any type of measurement of the software quality. Hence it measures a software system, process, or related documentation. Examples are the measures of the size of a program in lines of code, the number of reported faults in a software product and the number of person-days required to develop a system component.

Metrics fall into two classes, namely control metrics and predictor metrics (Sommerville, 1998). Control metrics are those used by management to control the software process. Examples of these metrics are effort expended, time elapsed, and disk usage. Estimates and measurements of these metrics can be used in the refinement of a software project planning process. Control metrics can provide information about process quality and are therefore related to software product quality and process improvement.

Predictor metrics, on the other hand, are measurements of a product attribute that can be used to predict an associated product quality such as maintainability. For example, it has been suggested that the readability of a product manual may be predicted by estimating its Fog index, or the ease of maintenance of a software component may be predicted by measuring its cyclomatic complexity (Sommerville, 1998). Whether valid quality predictions can be made from such measurements is open to heated debate. Ideally, a predictor metric would allow prediction of the value of some external attribute of the software (such as maintainability) by measuring an internal attribute (cyclomatic complexity, for example). An external attribute is something which can only be discovered after the software has been put into use. An internal attribute is an attribute of the software which can be measured directly from the software itself. External attributes cannot usually be measured directly so it must be assumed that a relationship exists between what can be measured and what information is required.

Twenty years ago metrics were used relatively little in the software world. Recently, an increasing awareness that they have an important role in software quality improvement has been witnessed. Most of the work in software quality, however, is related to industry and little work has been done in education. This thesis shows that they also have a role in education: its evaluation and assessment.

The particular software whose quality is being examined here is a student's program. Software metrics are being used to determine student program quality attributes. There are a number of possible important quality attributes, but the key attributes, as mentioned above, are correctness and maintainability. These are discussed further in the next section.

4. Program Quality Assessment

Quality assessment is of fundamental importance in many areas of software quality control. Tools for assessing quality are useful in two distinct areas: assisting people in the software profession to improve the quality of their work in a systematic way, and helping a manager to maintain quality controls and uniform standards for projects teams. These tools are also useful in two comparable roles in education (Zin and Foxley, 1992). Since universities are anxious to teach the concepts of software quality control, the availability of a suitable tool would help students improve their own software quality and would give them the experience of working in a quality control environment. This corresponds to the use of such tools by software professionals described above.

These tools can be of use to managers, in this case the teacher, but now in two distinct ways. Firstly, they can provide valuable feedback to the teacher about the strengths and weaknesses of the class as a whole, and thus indicate any emphasis that should be made in the classroom. Secondly, these tools can assist in marking students' work, otherwise a time consuming task performed unsatisfactorily by hand. Throughout a course, students may be given one or two assignments every week. A printed form of the students' work is normally marked together perhaps, in the case of programs, with the results obtained from running them. These papers are then handed to various graders who assess them. However, this type of arrangement is unsatisfactory for several reasons (Zin and Foxley, 1991)

a. With large classes, the volume of papers to be handled is inconveniently large.
b. If several people are involved in the marking process, it is difficult to maintain a uniform marking standard.
c. Recording the marks, collecting statistics about them, and analysing these statistics is extremely limited.
d. Programs submitted by students are not thoroughly tested by the teacher, so their effectiveness and accuracy cannot be fully assured.

CHAPTER III

REVIEW: COMPUTER TECHNOLOGY IN EDUCATION

There has been a wealth of research since the 1980's on the use of computer technology in education. There is an immense diversity in that work that makes it hard to categorise. Nevertheless, the following three main categories can be identified - it should be noted that much of the work reviewed can be placed in more than one category.

- On-line interactive course delivery and learning.
- Examination and Test Assessment.
- Courseware management and assessment.

Two important technological developments were heavily exploited in much of this work. These were the use of networks mainly the www on the Internet, and introducing dynamic paradigms or animations for teaching/learning. Work done in the context of these two developments has been examined first.

1. Using the Web for Learning and Teaching

Hypertext and the www appear to have a great positive impact on learning nowadays. It has been realised that the www enhances the role of the student as a learner. Behaviourism and constructivism are considered as two opposite views in learning. Behaviourism is related to student behaviour in conventional lectures whereas constructivism is related to coursework, project work and so on. The use of the www falls into the second category. Some researchers identify four web models of learning; the Web as "Source of Information", the Web as an "Electronic Book", the Web as a "Teacher" and the Web as a "Communication Medium between Teacher and Students".

The web as a Source of Information.

A typical university course supplies students with information and assignments. Students frequently use the Web to obtain information for their assignments or projects from various web sites.

The web as an Electronic Book.

In this case, information is presented in a structured way. Students learn through on-screen instructions. There is no interaction between the teacher and the students through the Web.

The web as a Teacher.

During the study of some of web based courses, there exists communication between the teacher and the student and also between students themselves. However, in this context the Web is still not regarded as a very good medium for inter-personal communication.

The web as a Communication Medium between Teacher and Students.

Students may learn from the teacher through the Web. This is a more satisfactory model as a communicator between the student and the teacher. Such a model also provides the communication medium. However, the Web is not always suitable, for instance in the teaching of professional acting, as the teacher and student may need to see each other.

An example of the implementation of the Web as a Communication Medium between teacher and students is explained in Astruchan and Rodger (1998). In a typical lecture session the students can view different windows. In the top window, they see and hear their lecturer explaining the material in real time; in another window students can make comments, or ask questions using text entry. Other students can see these comments as well. Such a method could be very useful for individual student/teacher consultation and formal lecturing. Text, graphics and pre-recorded audio and video clips may be used. Such an implementation could be very useful in distance education. It is also useful in that students can review lectures later on and they can do this whenever they wish.

Although using the www for course delivery is becoming very popular, it is not without drawbacks. Most novices, when they are using some kind of network facilities, click links that are irrelevant. They get carried away, and when they realise that they have to do their coursework, or carry on their learning, they have already wasted a lot of time and effort browsing irrelevant things. There should be some restrictions on their use of the www for the study of their courses.

Another example is GA (Genetic Algorithm) Web Tutor (Cole *et al.*, 1998) that is used to provide educational software through the www. This uses the Web as an electronic book. The growth of the www and the development of the Java Programming language offer a lot of opportunities in education. There are platform compatibility problems in using Java applets. Through the implementation of a front-end GUI to an application, such problems are decreased. Educational tools can be built in Java language, placed on a web server, and used with a Java enabled browser, this requires a browser enabling Java and ported to the user's platforms with its own Java interpreter. Cross platform compatibility then disappears. An interactive genetic algorithm tutorial package is used with the help of Java applets. The tutorials are made available to anyone on the www.

2. Dynamic vs. Static Paradigm

A shift away from a relatively static paradigm of teaching and learning towards a dynamic paradigm is necessary in computer science education. Nowadays, in addition to web-based lectures, it is possible to provide dynamic visualisation (or animation) of computing concepts (Herbert and Brumund, 1998; Naps, 1998). Animation is actually recommended by many researchers (Stasko, 1998) as an effective learning tool. World

Wide Web and Java applets opened a new era for algorithm animations, permitting remote interactive access to such animations with platform independence and an ability to link animations to text, sound and video. In addition, programs that use animations or visualisations attract student interest and offer feedback that can enhance different learning styles as students work to master programming and problem solving. In short, animation enhances students' learning and understanding and promotes their interest in, and enthusiasm for, the subject under study. The main animations explored (Boroni *et al.*, 1998) are: *program animation*, and *general concept animation.*

Program animation is the interactive presentation of the program on the monitor so that it can be studied and experimented with. Animator software has been developed by the research group of Boroni *et al.* (1998). The software comes with an editor and a compiler so that it can be used by instructors for editing groups of programs. Students can experiment with these programs by rewriting and developing them.

Algorithm animation refers to the virtual, graphical representation of algorithms on the computer monitor. For instance, a sorting algorithm can be demonstrated using bars and after sorting they are rearranged in descending order. Also insertion can take place.

Concept animation covers the rest. For instance, to animate the parsing process, Boroni *et al.* (1998) animated the theory of computing. It would appear that in the future the ideas of program animation, algorithm animation and concept animation will be coalesced onto a CD or DVD, and that new editions could be released every semester if required.

However, there are some hurdles to mention as well. Java applets often run too slowly. File operation problems exist. Platform independence has not been solved between Netscape, Microsoft and Sun. It is still a matter of discussion whether the Web dynamic paradigm is better then the current available paradigms. There are definite advantages and disadvantages. Due to these hurdles, there are limitations in such a paradigm. At the same time, students are more attracted and motivated, and the explanation of certain difficult parts of the course is made easier for lecturers.

Animation tools for computer science education were developed in each of the above three categories. Below some examples that are typical of the work are examined.

Algorithm animation was demonstrated in Boroni *et al.* (1998), Herbert and Brumond (1998), Naps (1998) and Stasco (1990). A tool was prepared which uses sort algorithm animation, in the form of an applet, so that it can be used on the World Wide Web (Herbert and Brumond, 1998). The tool was introduced for teaching and learning sort algorithms. An animation technique is added which enhances the occurrence of recursion in sorting algorithms. Sort animation and code animation are both shown in split windows. When the code is being executed, it is possible to view vertical bars which represent the elements to be sorted. The user has several controls over the animation (e.g. colour, number of items to be sorted, and the arrangement of the sort in ascending/descending order). In addition, the speed of the animation can be altered. There are several buttons at the bottom of the window, where the sort button starts the sort, and stop stops it. When the code is activated, the line being executed is highlighted. Recursive sort algorithms can also be viewed with the sort animator. In this case, horizontal bars at the top of the window show the level of recursion.

Sort animator and sort animation builder are thus useful tools for enhancing the teaching of sorting. Sort animation builder is used for both the instructor and the student to be able to build their own algorithm. In the lab, the instructor may show the animation of the vertical bars to the student, ask which sort algorithm it is and code it. When students build their own animations they tend to experiment better.

In addition to using animations for teaching sort algorithms as in Herbert and Brumond (1998), compiler algorithms such as parsing can be integrated into full animations. In full animation, the input string being parsed, the corresponding actions that take place in the stack, and the building of the parse tree are all simultaneously animated on the same screen. This enables the user to get a full appreciation of all the intricate details that occur during parsing (Khuri and Sugano, 1997). XTANGO (Xwindow Transition- Based Animation Generation Package) (Naps, 1998; Khuri and Sugano, 1997) was developed to demonstrate the animation of the whole compiling process. It is used to help students gain an understanding of the compiling process.

As discussed in Herbert and Brumond (1998) and also in Boroni *et al.* (1998) to use XTANGO package, which is claimed to be user friendly, it is only necessary for the user to know a few control commands. This is necessary to drive the file to start animation. Four objects are considered in the animation table: a stack, input string, parsing tree, and an action box. The action box shows each action taken. The parsing tree is updated each time. One pointer points to the top of the stack, and another pointer points to the current character in the input string. Two windows can be examined; one for XTANGO and another for the user console. Animations can be viewed in the XTANGO window. Such a package is believed to introduce the two parsing techniques to students effectively. Students can also use it to create their own parse table and then test it with an input string to see whether it works. When students take a compiler course, a programming project is given. According to the weight of the project they are required to form a parser, and possibly an additional lexical analyser which feeds the tokens to the parser.

The novelty of the package for the user is that the user can view all the steps in the parsing process; the items being popped and pushed out and into the stack, and the input string; the whole parsing tree can be viewed on the XTANGO window.

Graph Theory and its algorithmic aspect are known to be a difficult part of Computer Science study. Software called DIDAGRAPH (Culwin, 1998) was developed for the teaching of graph algorithms. The difficulties for students in graph theory are of two kinds:

- students perceive how an algorithm works, but they have problems in understanding its details
- they have difficulty in understanding and interpreting the intermediate results

DIDAGRAPH is a kind of visual system, used for didactic purposes. It is recommended to help students overcome the above mentioned difficulties. The implementation of such a system is as follows. A student can choose an algorithm and by choosing a solution he can see the execution of the whole algorithm and can describe the next step by manipulating the vertices or edges of the graph. However, the software is still under construction. For example, one of the problems the authors of the system in Culwin (1998) are trying to implement is the design of a graph-oriented calculator.

Animation of Data Structures is demonstrated in Pierson and Rodger (1997). It is suggested that using animation to study data structures as well as for debugging is more helpful to students than plain text. Students can thus understand data structures better and the use of animations to debug programs can help increase speed in finding errors by visualising incorrect movement or pieces. A system called JAWAA (Java and Web Based Algorithm Animation) was developed for this purpose (Pierson and Rodger, 1997). It is a command language written in Java and runs on the Web. It can be used either in the classroom for demonstration purposes, or for the students to generate their own animations.

JAWAA provides an interface through which users can write, run, and display animations over the Web using a web browser that supports Java. It is not necessary to know Java in order to use JAWAA. One-line commands are used to create and display data structures, then another set of one-line commands perform operations on the data structure. Arrays, stacks, queues, graphs and trees can be created. For example, the node command

```
Node n1 30 20 10 black light grey
```

creates a node named n1 located at position (30, 20) with a diameter of 10. The outer part of the node is black, whereas the interior is grey.

Animation of standard C/C++ is done in Sangwan *et al.* (1998). When students create their own animations, they can identify the parts with which they have difficulty, such as parameter passing. For instance, when demonstrating a program with structures and a function using graphics and window interface, separate windows are used to display all global and local variables during the execution of the functions, function calls, the code of the main, and the code of the function.

WebGAIGS is a multi-window environment for simultaneous visualisation of algorithms on the www (Naps and Bressler, 1998). Visualisation shows the execution of an algorithm as a sequence of graphical snapshots of the algorithm's state over time, from the beginning of its execution to its termination. However there are still some visual shortcomings, such as lack of ability to simultaneously see several algorithmic states and lack of ability to see side-by-side comparisons of algorithms.

SQL-Tutor, which is an Intelligent Teaching System, is designed to solve the difficulties of students learning SQL (Naps and Bressler, 1998). SQL is a database language that contains data definition and manipulation statements. The difference in this system from other similar systems is that it focuses on individual students checking their areas of weakness. Students who have difficulty in learning SQL, particularly RDBMS (relational data base management systems) are not able to deal with semantic errors. SQL-tutor gives better feedback to students. It is an Intelligent Teaching System (ITS) developed for guiding students in learning SQL (Naps and Bressler, 1998).

3. On-line Interactive Course Delivery and Learning

This is the use of computers' interactive capabilities and networks for the delivery of courses. Many of the systems mentioned above are used for some form of course delivery. The following systems are a further review of typical examples of such development.

The Tutorial Generation Toolkit (TGT) (Barnet *et al.*, 1998) framework has been developed to provide on-line, highly interactive, tutorial material that can be applied to the delivery of introductory computer courses, which can be useful for a variety of hardware software / software platforms. In this case Java is selected as the platform. The software is formed of interactive hyper-linked slide presentations. They contain text, images, animation and sound. Sequencing of the slides is organised by keeping a track of the ID (identity) of the current slide, its link and the destination slide. As a result of measuring the impact of these tutorials on students, a particular method of testing is done. Results are logged into a database using Java features. Interactivity of the tutorials is provided by selecting and manipulating buttons; there are certain text input areas and animation is used in some of the tutorials. For instance, when explaining fetch-execute cycle, the signals going from CPU to memory are animated.

The General Course Interface (GCI) system is used, along with UNIX, to overcome problems in teaching computing to large classes (Canup and Shackelford, 1998). Students login and off to the GCI system, PATH, and some of the environment variables are changed every time they access the system. Programs on the course are made available to the students. In the UNIX system, transferring binary files is complex, so `uudecode` and `tar` commands need to be used. However, such commands, as well as `ftp`, seem difficult for novices. So a command line interface allows them to get and view the programs in their accounts.

A GCI program called 'get-job' allows students and teaching assistants to retrieve coursework; a marking scheme is distributed to the teaching assistants via the same program. In order to prevent errors caused by using `tar` and `uudecode`, a program called `turnin` allows students to return any coursework or assignment easily. It may be troublesome for novices to cope with both UNIX and their assignments at the same time. They may resubmit their assignments until they are confident about the right solution. Each time the assignment will be overwritten by the new submission. The program `turnin` logs every error which appears in a student's program (e.g. format errors, runtime errors and so on). Log files are kept for every access of the program. Deadlines are also enforced by the same program. In order to save time in marking, work is required to be written in a specific format. Feedback is done by an on-line survey of the students. However, the new version of the GCI will be supporting the www. The 'Getgrades' program organises the curves, drops the lowest quiz etc., and students can view a summary of their marks. A student's program won't be accepted by the GCI if it does not log to a file. Information such as user name, program title and the time it is run is contained in this file.

<u>**Courseware**</u> has been developed at the University of Wales, using the www, for delivering courses such as: C Programming, Xwindows, Parallel Processing,

Computer and Vision and Image Processing, Artificial Intelligence and Computer Graphics (Mason and Woit, 1998). Software tools are integrated with the HTML language. User interaction with the material is supplied by the use of HTML; an HTML based user interface is written for the existing Ceilidh system. Java language seems to be common in this sort of application, and was used by the University of Wales to demonstrate linked lists.

4. Examinations and Tests Assessment

This is the use of computer technology to conduct on-line exams and tests. The advantages of on-line programming tests and examinations have been addressed by Mason and Woit (1998). They pointed out that with classes of large numbers (e.g. one hundred students or more) it is always troublesome to teach and assess, particularly in the case of first year computer science students. Some students may not yet have sufficient ability and therefore attempt to copy or cheat when examined or when preparing assignments. While they would appear to have received a good mark, these students would not have gained knowledge and experience required as a foundation for further study. Using computer technology in examining such classes brings great benefits to everybody. Three types of benefits can be realised. For the teacher: preparation, marking, feedback statistics, etc. For the student: simultaneous availability of the exam, and all documents needed. For the exam procedure: security and a reduction of plagiarism.

Mason and Woit (1998) conducted a study on the use of computer testing. Students were divided into two groups. The first group was concerned with the principles of computer science where functional programming is introduced and the second group concerned C and Unix Shell programming. In the former, there were both kinds of tests; conventional and (partially) on-line. The latter group included solely on-line tests, since the best way to learn programming is by practise. The most important goal in running such tests with a large classroom is security. By using a Unix facility, `chroot`, a directory is created for each student above the root, so everything becomes private to that student. All communication facilities (e-mail, telnet) were disabled and passwords for the test accounts were set and distributed during exam time. Students used commands with which they were familiar; on-line documentation of these commands was available. Sometimes students asked for scrap paper to use for working out problems. Even though Unix `emacs` editor provides multiple windows, students still preferred to use scrap paper as the initial problem solving editor! Also they required hard copy of the tests. It is difficult to explain why; maybe they felt more comfortable to see the question on paper first, and present it on-line afterwards.

A drawback to the system is that when students make avoidable mistakes and they are not able to compile their program, their confidence decreases and they do not feel like doing the rest of the question. This situation may affect students of all abilities.

A further drawback is the incorrect use of the naming convention for the student file. Even though students are told to name their files in a certain way, some of them may not do so, forcing their lecturer to waste time finding the file in the system.

Students find on-line tests more challenging and they tend to be seen as a threat to students who have a tendency to cheat or copy during practical sessions. They need to do enough practice to bring their practical skills to a certain level. Marking some parts of the course was fully automatic, but a part of the programming test was marked conventionally.

It is debatable whether students should take on-line tests for programming courses or whether they should work for assessed assignments. The marking of assignments then can be carried out automatically as is done in this thesis. When preparing their assignments, students have enough time to think and they can detect their compilation errors without panicking.

The Computer Aided Assessment (CAA) software, Question Mark, was developed for the assessment of large groups of students in introductory systems analysis and design (King, 1996). The essential feature of this model is that a test is available on the computer network. Students have a certain set period, usually two weeks, to do the test. After the deadline the test is normally removed immediately. With back-to-back testing one test is removed then the next is made available. After the test completion deadline, feedback is given to each student on her/his individual performance. Although the tests are open book, the model generates the atmosphere of an in-class test. Tests contain multiple-choice questions. Four tests were initially delivered back-to-back each semester. Later the number of tests was reduced to three due to the following facts:

- preparation of separate tasks such as the control of the release of a test and its removal.

- issuing feedback reports to students, informing them about dates and deadlines and so on.

Students were initially given a tutorial, but in the final year this was dropped; they were asked to familiarise themselves with Question Mark by trying the example tests provided. This was successful. As a kind of feedback, Question Mark tests had a corresponding revision test, which provided question-by-question feedback. 90% of the students reported these revision tests to be useful and two-thirds used the tests more than once, increasing their scores.

The most difficult questions were discussed in the classes. Feedback to students on their own test scores and individual question responses varied over the course of the study. In the first two years scores were posted to students at the end of each test period. Feedback was enhanced by exception reports, which covered those questions that were answered incorrectly. Later on these were cancelled and the students in the second year relied on revision tests as a kind of feedback. There was an improvement when the marks were given at the end of each test in the final year, but individual question-by-question feedback was still a problem. The model relied heavily on exception reports as a form of feedback. These must be a basic requirement for any software used for continuous Computer Aided Assessment (Stasco, 1990).

Regular individual feedback was not successful, and this reduced the usefulness of the test to students and staff. Many of the restrictions were due to making the tests in class more formal, and also reducing collusion between students when carrying out such tests.

Later tests were made available as soon as the topics were covered and left on the system until that course unit was covered. This was done in order not to make the tests appear as a substitute for in-class tests.

Question Designer (McCabe and Troise, 1996), Another piece of software, is a general-purpose product but it has its limitations, especially for specialist subject areas. It is complemented by external specialist tests. It has benefits, however, such as the ability to exploit alternative question types, test structures, and interactions developed within specialist tests.

The ExCon project (Nulden, 1998) is an intervention with the premise that examination in higher education must focus more on process and less on product, and this can be supported by mobile computing (Nulden, 1998). In the early 1990's PDAs (personal digital assistants) were introduced. There were still drawbacks in using them but, after solving early problems, the latest versions of PDAs and mobile computers are used successfully nowadays. The ExCon project and tracker is an attempt to use mobile computing and the www as an alternative method of assessment.

5. Courseware Management and Assessment

The objective of this section is to review the work carried out in the use of computer technology in courseware management and assessment. The research in assessment has been undertaken in several areas. However, here the assessment of programming courseware, as it is relevant to this thesis, is reviewed.

There are several computer assessment systems. The most important research can be found in the development of the four systems: Pest (Oliver, 1996), PASS (Thorburn and Rowe, 1996), SPROUT (Rimmer, 1997), and Ceilidh (Benford *et al.*, 1996) which are used for assessing programming language courses.

PEST (Program Exercise Solution Tester) is a UNIX based system used for assessment of program correctness. It was developed by R G Oliver at the University of Hertfordshire Computer Science Division. (Oliver, 1996). He argues that it is easier to automate the marking of programming assignments for correctness than for style. Style can be marked by hand, whereas marking correctness by hand is tedious and difficult. He says that automatic assessment of style is hard since many factors have to be taken into consideration and balanced against each other. However, checking each student's program by hand for style, particularly when class sizes are big is arduous, even though Oliver argues that it is easier. Factors that affect style such as indentation, depth of indentation, lengths of variable names, comments, and so on have to be counted and marked for each program. This is very time consuming and requires extra staff. This issue is addressed in Chapter IV.

PEST is used for the automatic assessment of correctness of students' programs. It attempts to compile/link specified student programs and, if successful, subject them to sets of test data. It is written in C and, by the writing of UNIX shell scripts, it is customised to do compilation/linking for particular languages. When assessing programs, PEST gets the names of the files from the directory of the students and executes a Shell script. During the execution of the Shell script each student file is given a full pathname, including the exercise name. The script reports back whether a student's program is found, readable and so on. When control returns to PEST if program marking is selected, it checks through a specified directory for the test data. PEST executes each student program with a set of test data. If a program terminates abnormally, certain symbols are displayed (e.g. "crashed", "killed" etc.) in the output obtained. These symbols are considered later in marking.

It has been observed that more care is needed in the preparation of student exercises. For example, it is necessary to be more prescriptive in what is to be included in the output - what form, what order - so the writing of the marking script becomes more manageable.

Teaching staff did not face the burden of the marking process with such a system. Rapid feedback encouraged the students to take a more active approach to the course. PEST is also set for an interpreted language, Miranda. However, using PEST requires, in general, a translation script.

PASS (Program Assessment using Specified Solutions) is another assessment system, which used to act as an environment to aid the assessment of programs. It analyses a student's program and quickly provides information about the design of the program. The course tutor provides a solution plan to be analysed by the PASS system. The solution plan contains descriptions of the functions/procedures and their names. Their implementation is not specified. This is called abstract description of the function/procedure. The tutor's and student's solutions are compared regardless of their implementations and equal functions/procedures are extracted. So the problem of analysing solutions using different implementations is solved by comparing only what the function/procedure is doing, not how it is doing it.

Such a system is intended to guide the students in solving the programming exercise by using a valid solution plan. It is possible to solve the question requirements by using ill-structured solution plans. But the aim is to learn programming using a good program design which has a valid solution plan.

If a student's program uses an entirely novel approach and still produces the same result, PASS will recognise this and will inform the student. Also PASS will recognise if a program does not produce the correct overall answer but uses a partially correct plan.

Only the solutions to a small subset of C language can be assessed in this system. These features include the four basic data types of all operators and arrays. However, more advanced C features such as user defined variables and pointers cannot be assessed. The system was successfully able to assess programs which have too have many functions. Later on, after the evaluation of this system, limitations to assessing only simple features of C programming were removed and now it is being revised to assess more advanced problems.

The PASS system was evaluated by Thorburn and Rowe (1996). Particularly, the assessment of program design when programs and solutions become more complex, were observed. One of the problems in dealing with more complex problems is that valid solutions may not follow the same functional layout as that of the tutor. This is solved by allowing multiple solutions for each problem. The system will compare the student's program with each of the solutions and assess it according to the closest tutor solution. The other possible development to this system is to turn it into a tool which students themselves can use to assess their solutions.

Thorburn and Rowe (1996) state that a method of assessment that compares students' results with intended results have some drawbacks. In some cases there is no problem. For instance, in an exercise that asks the student to sort a data file using certain flags and conditions there is no alternative output. However, in other cases a student might have produced the correct output, but she/he could lose marks because the output might not be the same as the tutor's. However, this is not the same in the assessment system used for this thesis which assesses Shell programs. Shell programming language is interpretive and each shell programming exercise is a set of commands. For dynamic correctness, the student output is compared with the intended output. However, if the course tutor wants to be more flexible for certain exercises, it is possible to give partial marks to encourage the students. In particular, in typographic assessment the tutor gives certain weights for each factor. This could be different for each exercise. This is discussed in more detail in Chapter III. Also in Chapter VI the code is checked for its complexity.

SPROUT- Simple Programs Routinely Observed Under Test (Rimmer, 1997) is a system that was developed for assessing small Pascal Programs. Students submit the programs during laboratory sessions for part of a first course in programming. The assessment method for this module is coursework only, which comprises only 10 small exercises. Once the student submits her/his work to the system, it is compiled and if the compilation is successful, the program is executed with test data cases. Each execution is monitored for successful completion and the output produced from each execution is tested for correctness. An overall mark, and also feedback, is provided for their mistakes. Also, the system retains a copy of each submitted work so that these programs can be examined at a later date to find out common problems. Incorrect statements, and statements which are acceptable but less desirable, can then be identified. Depending on the assignment, the tutor looks at the following points:

- The program compiles without errors.
- The program executes to normal completion.
- Using one or more standard input cases, the program produces correct output with appropriate titling and spacing.
- The structure of the code matches with the given pseudo-code - this is given as comments in the program.
- Identifiers should be meaningful.
- The program has been correctly formatted. A software formatter is available.

The quality assessment is in three areas:

1. Source code profiling – the student program is profiled using a syntax analyser.
2. Identifier analysis - identifier names are checked against a list of acceptable names.
3. Code formatting - students use an in-house documentation workbench to format their programs.

The knowledge base for each program exercise contains the following:

- Input cases with corresponding output.
- Executable code profiles (symbol or token lists).
- A list of acceptable identifier names.
- A list of unacceptable identifier names.

The knowledge base is password protected and only the tutor can use it.

SPROUT runs student programs with test data and compares the output. A variation of space characters and upper/lower case characters is allowed in the output. This is similar to PEST and dynamic testing done in this work (Chapter V). SPROUT also profiles (divides into tokens) the students' programs by using a standard syntax analysis to see whether students' profiles and SPROUT's match. In the work examined in this thesis, identifier length is checked against good identifier length, which is set by the tutor. SPROUT checks the students' identifiers against a list of acceptable identifiers given by the tutor. This is a good idea, but the tutor could inadvertently miss some acceptable names. Students, particularly novices, get frustrated and lose their motivation when they lose marks for no reason.

ICCASAS (Integrated Clarity, Complexity and Style Assessment System) (McAlpin *et al.*, 1997) is an automated tool used for assessing the style, complexity and clarity of Modula-2 programs. Such a tool calculates a percentage mark for each student's program by comparing it with the lecturer's solution. A marking scheme is derived from the lecturer's solution, since different assignments emphasise different aspects of style and complexity. The measures used for style are similar to the measures used by Rees's STYLE tool (Rees, 1982). A percentage mark is given for the overall program rather than individual procedures. The assessment of the use of algorithms and data structures involves measurement of loops and decision-making structures such as `If` and `Case`. Algorithmic complexity is measured by McCabe's cyclomatic complexity metric. For indentation, the ICCASAS system uses a variation to the indentation measure used for STYLE. In the latter, this measure is simply the number of lines indented. In ICCASAS, the measure used compares observed indentation to expected indentation. This measure is the coefficient of correlation between those indentations mentioned. An evaluation exercise has shown that system to be useful in the teaching, assessment and learning of programming.

CEILIDH (Benford *et al.*, 1993) is an on-line coursework administration and automatic-marking facility, designed to help both students and staff with programming courses. The system acts in a number of ways for students, tutors and teachers and supports a variety of programming language courses such as C and C++. There are facilities for students, such as reading notes and coursework material, looking at examples, developing and marking their own programs. For tutors, the facilities are observing submitted work and marks and checking for plagiarism. For teachers those facilities involve setting up exercises relevant to the course they are teaching and also amending course material.

Once written, the student asks the system to mark his coursework. A summary of the marks is made available to the student. She/he can mark her/his program several times thus working towards a quality target. Once the deadline is passed Ceilidh will allow students to access the teacher's model answer and try to run their solution and the model against the test data. Then they are allowed to send comments to the course developer/teacher concerning that specific exercise. In addition to this staff can view more details of the marks such as weak points in each exercise and late submission. More complicated reports and charts are provided, for example performance of each student for all the exercises in all the units covered.

One advantage of this system is that students can mark their programs and a summary of the marks is made available to them. In addition to the overall mark, they can also view their marks for typographic style, dynamic testing, and complexity.

At the University of Luton, the Ceilidh system has been used for first year programming. 'Two Years of Ceilidh', (Allum, 1996), discusses their experience. They put forward three measures to judge the effectiveness of this system. These measures are:

- a reduction in academic hours spent in teaching, preparation and marking.
- improvement in student satisfaction measured by SPOM1 improvement in drop out rate.
- improvement in student pass rate.

After collecting information about the above measures, they believe that they have enough evidence to justify the continuation of Ceilidh. It has also been recommended to other institutions. It is concluded that the Ceilidh system produces a great improvement to the tutors' quality of life, due to the reduction in marking - they only have to write questions-exercises. The other benefit is the marking consistency. However, these only occur when Ceilidh is used by a lecturing team that is in touch with its own student' needs and is ready to create a course to match these needs.

At Nottingham University, a model has been developed as a tool for Ceilidh to automate the assessment of question/answer natural language exercises (Lou, 1996). Natural Language Processing (NLP) is an area in computer development where computers are intended to understand or assist humans in natural language. The student's work submitted in the English language is assessed even though there are differences in vocabulary, grammar and style between answers. A prototype of the model called Simple Text Automatic Marking System, (STAMS) has been developed. An approach involving fuzzy semantic techniques is adopted. Also an online Roget's Thesaurus is used as a knowledge base. Semantic Processing means the study of words, phrases and sentences. This system compares the tutor's model answer with the student's answer. Both answers are processed separately. The Ceilidh system course developer and student interfaces are used. STAMS asks the teacher to give her/his possible answers for the question/answer exercises, and then it turns these into fuzzy semantic sets. Students submit their answers to the Ceilidh system and there is a deadline for this. They are allowed to mark their answer several times before they submit and sometimes they may resubmit. The system was written in shell programming language and also sed, a UNIX facility, was used. The author, however, suggests that it is faster to run the programs with C or C++.

To make computers understand natural language is a great achievement, of course. But it is still difficult. Grammar rules are not included in this work. Perhaps, the tutor/lecturer can add grammar rules to make the answer more specific.

A methodology for courseware design has been developed at the University of Wales for web-based courses and C programming. Recently a new course, Internet Computing, has been introduced. Automatic feedback and marking have been incorporated into the courseware. Submitted coursework is automatically assessed on-line (Marshall, 1997). As it is not possible to assess the entire course, certain topics are selected such as e-mail, ftp and www programming. For instance, students are asked to do ftp to a certain location as an exercise. Similar to Ceilidh dynamic tests, (Chapter VI) a Unix `diff` command is used in the scripts to compare a student's template with a solution template.

An automated diagram comparison system has been developed at the University of Teesside (Hoggart and Lockyer, 1996). It is used by students to compare a solution diagram against a model answer that is also a similar diagram. They receive feedback according to the comparison of their solutions.

A CASE (Computer Aided Software Engineering) tool which has facilities for drawing diagrams, and a CAL (Computer Aided Learning) system can be combined together as a learning framework (Hoggart and Lockyer, 1996). At the beginning, the student will need some basic information. This can be obtained from CAL material. Several levels of learning are merged together. The first level is obtaining the basic information from a CAL system. It is called 'Exposition Level.' The second level is the time for exercises and assignments. They call it "elicitation level repetitive exercises". Due to the fact that a CAL system has limited diagrams, CAL is embedded within CASE. A CASE tool can provide feedback to the student about his/her diagrams (Hoggart and Lockyer, 1996).

Response shows that it is possible to generate valuable feedback on the system analysis and design diagram development after comparing students' diagrams with the model answer.

Two new courses, Pascal and Modula-2, were developed using the Ceilidh assessment system (Lewis,1997) at the University of Glamorgan. As was mentioned above, the Ceilidh system has course developer's facilities such as the creation and modification of course material, and facilities for setting up exercises for each unit of the course. The similarity of Modula-2 to Pascal enabled some reuse of Ceilidh tools for Pascal. Various exercises have been modified and reused. Existing courses include interactive handouts distributed electronically before lectures, using a combination of common directory areas and/or electronic mail as appropriate. These are intended to encourage students to prepare in advance of lectures, and also to be able to complete the missing sections. With the use of an in-house Modula-2 compiler, dynamic tests could be reused by rewriting the solution in a different language. Those exercises written for Pascal, C, and C++ were reviewed and those which fitted the overall structure of Modula-2 course were selected. Solutions were designed and implemented in Modula-2.

6. Outline of the Proposed Assessment System

The system described in this thesis has been developed to measure the quality of UNIX Shell programs. The objective is to use the values of the measures for grading students' courseware in Shell programming. Only two attributes of a program are measured; its maintainability and its dynamic correctness. These two are the most relevant to a student's learning environment. Firstly, for students to achieve any experience in programming they should produce dynamically correct programs that run and give correct results for the problems solved. Moreover, as their programs are going to be revised many times for the purposes of correction, modification or improvement, they must also be maintainable. For dynamic correctness to be checked the student's program should be run against sets of input data supplied by the teacher, whereas for maintainability the program is tested statically. The latter requires that the text of a student's program is analysed for certain aspects of quality. Two aspects of the quality of a program will be assessed here; these are its typographic style and its complexity. Typographic assessment is examined in Chapter IV, program Complexity is discussed in Chapter V and finally dynamic testing is discussed in Chapter VI. Program structures are explained at the beginning of Appendix 1.

CHAPTER IV

TYPOGRAPHIC ASSESSMENT OF SHELL SCRIPTS

The aim of this chapter is to describe the automatic grading method that is based on analysing Shell programs typographically assuming that they are structurally and syntactically correct. Properly styled programs are more readable and therefore more comprehensible. Such factors make the programs more maintainable. Marking each factor of the typographic style is achieved using a method similar to that which Rees (1982) has done. In section 2 the method of marking is explained. Section 3 explains the factors affecting the Shell typographic style. Finally, section 4 discusses the results.

1. Introduction

As mentioned in the last chapter, the student's program can be assessed by testing it either dynamically or statically. For dynamic testing the program has to be run against sets of input data, whereas static testing means analysing the program text for various aspects of quality: typographic, structure, etc. This chapter describes a computer-aided assessment, as a part of the system developed for this thesis, which assesses programs by examining them and marking only their typographic style. Teaching students to style their programs typographically and improve their presentation is actually essential in learning programming languages (Allum, 1992 and Anand,1998). Much research has been done on typographic style of programs, for example Arab (1992) and Benford *et al.* (1993). It was found that 'properly' styled programs are more readable; the style increases a program's comprehensibility and in turn makes it more maintainable.

Of course typographic style on its own cannot be considered adequate for grading a student's work. For example, it is possible for code, which has an excellent style, to be logically incorrect. Typographic style assessment must be complemented by assessing other factors of quality such as complexity (LOC, control flow, etc.), and by checking the dynamic correctness of the program. This is investigated in the next two chapters: complexity checking; and automatic testing of shell scripts (Benford *et al.*, 1995).

2. Method of Marking

Marking each factor of the typographic style is done by using a technique similar to the one described in (Redish and Smyth, 1987; Rees, 1982). First, the score for each factor is calculated from the Bourne Shell script. For each score five parameters are provided (Figure 4.1- Figure of Rees). These are L, S, F, H and 'Max'. If the score is less than L or is greater than H, a mark of zero is given. If the score is in the range of S to F, the maximum mark is awarded. For a score lying between L and S or F and H,

a mark is calculated by interpolating according to their position. 'Max' specifies the maximum mark given for each factor.

If the course assessor does not want to mark a certain factor, he or she can achieve this simply by assigning a zero value for the maximum mark, 'Max'. A negative value can also be assigned to 'Max' if the assessor chooses to do so. This makes the system adaptable to changes in the teaching process. As the course progresses the teacher may shift the emphasis on the different factors from one assignment to the next.

Figure 4.1- Chart showing parameters L,S,F,H for each score
Figure taken from Rees (1982)

This technique was first developed by Rees (1982) to be used in marking Pascal programs. It was further developed and used by other researchers such as Berry *et al.* (1985). Essentially the marking process is based on comparing the student's script with a model supplied by the teacher. When assessed according to the teacher-supplied values of L, S, F and H, this model should score the maximum mark, 'Max' for each factor.

3. Factors Affecting Shell Typographic Style

Some of the typographic factors which affect the comprehension and maintainability of programming languages are universal, but others are peculiar to a specific language (Mcconnell, 1993). Factors selected in this work take into consideration the syntax and the features of Shell. For example, the Shell programming language has certain characters for which it reserves special meaning when seen in a program. These characters, called 'metacharacters' and 'special characters', perform a variety of tasks that make using the Shell easier and more powerful, (Arthur J.L. and Burns, T., 1996). Metacharacters, such as * and ?, are a useful shorthand for handling file names and directory names. The Shell special characters, on the other hand, are not involved in forming file or directory names, but instead they instruct the Shell to perform some specified action. Examples include, the characters for redirecting input and output (<, >, >>, etc.), the piping character (|) and the command separator ';'. Because of the cryptic nature of these characters, using them frequently in a shell program affects its readability and makes it difficult to understand.

Other features peculiar to shell are the environment variables and the positional parameters. These do not follow any particular guide of 'good' identifier length. They are some times either too long or too short and terse, and thus it is better to treat them as if they are keywords.

In the following, twelve factors were selected for the typographic assessment of Shell scripts, and each will be discussed in turn. The factors which are considered here cover the main aspects of Shell style and they are in line with what one usually considers while marking manually. Of course, additional factors could have been included but the view was taken that their contribution to the assessment of programs written by students, at the stage of learning programming, is at best negligible. The factors are

- average number of characters per line.
- proportion of lines with a 'good' number of characters.
- percentage of spaces per line.
- proportion of operators 'properly' spaced.
- percentage of blank lines.
- degree of 'good' placement of blank lines.
- percentage of comments.
- proportion of comment errors.
- percentage of indentation.
- proportion of lines with 'good' indentation.
- average identifier length.
- proportion of identifiers with 'good' length.

3.1 Average Number of Characters per Line

Spaces and tabs within a line contribute to the clarity of statements and expressions used, and thus increase its readability. In particular, spaces before and after operators, and command separators, have the effect of making the code readable. More spaces in a line means less characters and vice versa. The average number of characters per line is thus a measure of how dense the characters are in a line. This factor is calculated by first subtracting all spaces and tabs from the total number of characters in a program. Tabs are replaced by their corresponding number of spaces in the calculation.

The result is then divided by the total number of lines, as shown in the following formula.

$$\text{av. no. of characters per line} = \frac{\sum_{1}^{n} ch_i}{n}$$

where ch_i is the number of characters in line i, and n is the total number of lines in the program, excluding blank lines.

A higher value of this measure, therefore, means a less readable program. A very low figure means the characters are scattered in the lines which again makes the program difficult to read. Rees (1982), and Berry and Meekings (1985) have used this factor in their metrics. They have experimented with different values of the above measure and found that a value of 15-30 characters per line gives optimal readability. Although their work was done on Pascal, this particular measure is quite general and their results can apply equally well to any programming language, including Shell. Thus, S and F can be given the values 15 and 30 respectively, L a small value such as 4, and H some larger value such as 45.

3.2 Proportion of Lines with 'Good' Number of Characters

Perhaps not all lines will achieve a value of character density near to the optimal average of 15-30 mentioned above. The proportion which has achieved a 'good' character density is calculated. A 'good' number of characters in a line could be considered as being within, say, 30% of the optimal. In the system presented here, it was considered that any value of a proportion from 60% upwards is awarded the maximum mark and any proportion less than 10% scores 0. Thus the values of L, S, F and H could be .1, .6, 1 and 1, respectively.

3.3 Percentage of Characters per Line

Another measure is to calculate the proportion of characters per line by dividing the total number of characters by the summation of the line lengths as follows:

$$\text{av. proportion of characters per line} = \frac{\sum_{1}^{n} ch_i}{\sum_{1}^{n} l_i}$$

where ch_i is the number of characters in line i, l_i is the length of line i, and n is the total number of (non blank) lines in the program. However, a more accurate formula than the latter, as it reflects the distribution of embedded spaces within the code, is:

$$\text{av. proportion of characters per line} = \frac{\sum_{1}^{n} \frac{ch_i}{l_i}}{n}$$

Hence, L, S, F and H will take fraction values in this case as in section 3.2. It is assumed that below 5% means very little spacing was used and that above 50% the code is sparsely written and thus no marks are awarded in these two cases. This means that L=.05 and H = .5. Maximum will be given for values of the formula between .1 and .3. Thus S = .1 and F = .3. This is actually a very reasonable choice as it was realised that a value of 10%-30% of spacing per line is adequate for writing readable programs in C.

3.4 Proportion of Operators 'Properly' Spaced

Arithmetic operators such as + have to be separated by at least one space from the variable name that precedes or succeeds it. It is an error of syntax if no spaces are used, as this will be considered to be an illegal variable name. This is the same in many languages, including shell. However, no such rule is imposed for the special characters in shell such as the pipe '|' or the redirection operator '<'. Leaving no spaces between any such operator and the command or file name that is placed before or after it, makes it very difficult to read. This is usually experienced when several such operators are written in one statement, which is not uncommon in Shell programs, particularly when the programmer wants to take full advantage of the power of the commands and facilities available for manipulating files and directories in Shell. This is in addition to 'piping' available programs to construct a prototype. In contrast, it should also be noted that too many spaces also make the statement difficult to read. The system described here checks for spaces both before and after the six special characters |, <, >, >>, ; and &, since these are the most frequently used.

However, in a future development of this system, other special characters could also be considered.

For this purpose it was realised that 1 or 2 spaces before and after a special character are usually enough to enhance readability. The proportion of special characters that were 'properly' spaced according to the above rule was then counted. The maximum score was awarded for programs achieving 60% and above of this proportion. No mark was given for scores below 20%. Thus the values of L, S, F and H could be taken as .2, .6, 1 and 1 respectively.

3.5 Percentage of Blank Lines

The use of blank lines is one way to indicate how a program is organised. They can be used to divide groups of related statements into paragraphs, to separate routines from one another, and to highlight comments. Thus, the percentage of blank lines provides information about how well the program structure has been made visible to the reader. A study by Gorla *et al.* (1982), found that the optimal number of blank lines in a program is about 8 to 16 percent. Above 16 percent, the time needed to de-bug is increased dramatically.

The measurement of the percentage of blank lines is calculated by counting all blank lines in the program and dividing this value by the total number of statements of code. Considering Gorla *et al.*'s (1982) study, a maximum score could be given to values of this measure between 8% and 16%. Thus the values of L, S, F and H could be .04, .08, .16 and .3 respectively.

3.6 Degree of 'Good' Placement of Blank Lines

The percentage of blank lines shown in Section 3.3 does not demonstrate how well the blank lines are placed in the program text. The optimal of 8-16% might have been achieved but the blank lines were put in the 'wrong' places.

It is known that the use of blank lines enhances readability and that the positioning of blank lines is more valuable in some places than others. It has also been noted that there are three places of prime importance where blank lines to enhance program readability the most (McConnell, 1993). The first is both before and after comments. Shneiderman (1980) has realised that if comments are to be of any value they should not disrupt the visual scanning of the program code. To achieve this comments were off-set with blank lines. This helps the reader to scan the code without being confused by the comments or, alternatively, to get an overview of the program code by reading the comments only. The second position for blank lines is between a routine header, its data, and a named-constant declaration, and to separate this from the body of the routine. Thirdly, blank lines should be used to separate groups of related statements into paragraphs.

In order to calculate this measure, the places where blank lines are placed are checked to see whether they fit the above criteria. Only the first and second criteria were checked, as it was realised that the third (the separation of groups of related statements into paragraphs) is not only difficult to recognise but also it is subjectively affected by the programmer's taste. The proportion which achieved a 'good' blank lines placement is calculated. A 'good' proportion could be considered within about 30% of the optimal. In this system it was considered that any value of the proportion from 60% and up is awarded the maximum mark and for any proportion less than 10%, the mark is 0. Thus the values of L, S, F and H could be .1, .6, 1 and 1 respectively.

3.7 Percentage of Comments

Research has shown that using comments judiciously (i.e. in the right locations) is the best approach (Woodfield *et al.*, 1981). Specifically, if short comments are inserted just before logical modules this can aid comprehension by briefly describing the function of the module. A logical module can be considered here to be any group of lines of code which the programmer can consider describing as one task. Most work in this area in programming languages, as in Arab (1992), resulted in a guide to where to put the comments. Recommendations are that comments be put at the top of the code, at the beginning of every function and after each structure's ending key word such as **fi**, **else**, **esac**, **done**, etc.

Based on these guides two measures can be found; the percentage of comments as in this section, and the percentage of comment errors as described in the next section. The percentage of comments is calculated by dividing the number of comments (not the lines of comments) by the total number of program constructs. As too many comments obscure the program code, if the number of comments is more than double the constructs no marks are awarded and the values of L, S, F and H as .1, 1, 1.5 and 2, respectively might be taken.

3.8 Proportion of Comment Errors

The above measure does not reflect how appropriately the comments are located in the program, as mentioned above. The measure for the comment errors is calculated as follows. The number of properly placed comments is first calculated and by subtracting this from the *expected number of comments* the error value is given. This division results in the proportion of errors. The expected number of comments is equal to:

1 comment at the beginning of the code + the number of constructs inside the code

+ the number of functions (if any)

An obvious suitable choice for the values of both L and S is zero and for F is .3 allowing for up to 30% of error for the student to get the maximum mark in placing the comments. Whereas H could be taken to equal 8. That is, above 80% of errors a zero is awarded. Alternatively, since the proportion of errors is being marked, a negative value could be given to Max and for L, S, F and H values such as 0, 8, 1 and 1 respectively. This means that marks are deducted according to the proportion of comment errors found and a maximum mark of Max is deducted for errors above 80%.

3.9 Percentage of Indentation

Indentation is used to show the logical structure of a program. As a rule, one should indent statements under the statement to which they are logically subordinate. Indentation has been shown to be correlated with increased programmer comprehension (McConnell, 1993; Miara *et al.*, 1983; Arab, 1992). The article on "Program Indentation and Comprehensibility" by Miara *et al.* (1993) reported that several studies found correlations between indentation and improved comprehension. The same study also found that it was important to neither under-emphasise nor over-emphasise a program's logical structure through indentation. Inappropriate amounts of indentation prevents readability and therefore leads to programmer dissatisfaction. Miara *et al.* investigated the effect of several indentation levels on students' ability to read programs and answer short questions about them. Subjects scored 20% to 30% higher on a test of comprehension when programs had a two-to-four space indentation scheme, than they did when programs had other indentation schemes or no indentation at all. The lowest comprehension scores were achieved on programs that were not indented at all. The second lowest were achieved on programs that used six-space or more indentation. The study concluded that two-four-space indentation was optimal.

On the other hand, studies have been done by Arab (1992) and also by Miara *et al.* (1983) as to where indentation is most effective. Arab (1992) stated in his presentation that "all statements directly belonging to a control statement are right indented an equal amount from the beginning of that control statement". This means, for example, that a control statement such as **if-then-else-fi** takes the form

```
if      [test condition]
then
        command-list1
else
        command-list2
fi
```

Also, keywords, such as **if** and **else** belonging to the same construct must be in the same column, so that nested constructions can be clear. Considering the above discussion, a measure of the amount of indentation can be calculated as the fraction of lines of code that are indented by any amount. The values of L, S, F and H can, respectively, be taken as .1, .3, .6 and .8, giving maximum mark for indenting 30% to 60% of the code.

3.10 Proportion of Lines with 'Good' Indentation

Not obeying the indentation rules mentioned in the previous section leads to indent errors and hence lower marks. The measure in this section is calculated as follows. The number of places in the program where indentation is required according to the mentioned rules is calculated (*n*), and the number of 'good' indentations in those places is also calculated (*m*). Good indentation means indentation that is within the range recommended by Miara *et al.* (1983), i.e. 2, 4 or even 6 spaces. The fraction of indent errors, $\frac{n-m}{n}$, is then calculated. The values of L, S, F and H could then be taken to be 0, 0, .3 and .5, respectively, giving the maximum mark for up to 30% of error in indentation.

3.11 Average Identifier Length

Teasley (1994) stated that the naming style, though usually not important for experienced software professionals, is very important factor in comprehension of programs for novice programmers. McConnell (1993) has presented an extensive discussion on naming conventions, and Mynatt (1990) has discussed a model of program comprehension. Mynatt suggested that poor variable names would affect a programmer's comprehension of functions. Names that are too short do not convey enough meaning. The problem with names like X1 and X2 is that even if the value of X is apparent, it isn't immediately possible to know anything about the relationship between X1 and X2. On the other hand, names that are too long can obscure the visual structure of a program. Goral *et al.* (1990) found that the effort required to debug a COBOL program was minimised when variables had names that averaged 10 to 16 characters. Anand (1998) has presented rules for naming procedures, variables, pointers, etc.

Taking the above discussion in consideration the values suggested for L, S, F and H could respectively be taken as 4, 10, 16 and 20.

3.12 Proportion of Identifiers with 'Good' Length

As with other factors, not all variables may achieve the length of value 10-16 characters. The measure in this section checks the proportion of the program variable names that achieve that length. If the maximum mark is awarded for 60% (and above) of the variables which are within this optimal 'good' length, the values of L, S, F and H could be reasonably taken as .1, .6, 1 and 1, respectively.

4. Results

This part of the system was implemented in C code and was run under UNIX System. For an assignment to be typographically marked by the system the teacher has to supply the values of L, S, F and H, for each of the twelve factors, as input to the system. This will enable the change of emphasis to be reflected as the course progresses.

A typical output from a run of this typographic part of the assessment system developed for this study on a student shell program is shown in the following table (Table 4.1.). Factors that were given a maximum value Max of zero naturally do not appear in this analysis.

Table 4.1

Typographic Analysis

item	score	mark	out of
av. no. chars. per line	7.00	10.00	20
%blank lines	20.00	20.00	20
%spaces per line	15.51	15.00	15
%comments	0.00	0.00	25
%comment errors	100.00	-20.00	-20
%indentation	0.00	0.00	20
% good indentation lines	0.00	0.00	20
av. length of usr created. var.	3.67	8.33	15
%gd len. usr created var.	66.67	4.17	05

total mark is 27.5 out of 120 = 22.92 %

The system was run for a class of 92 students starting a course in Shell programming. The following table, for a given assignment, (Table 4.2.) shows the distribution of students in the five ranges (from L to H) for each factor selected by the course teacher. Some of the factors were dropped by the teacher, as insignificant, for assignments given at the start of the course. This is normally reconsidered by the teacher as the course progresses and emphases change.

Table 4.2

Factors	Number of students in score intervals				
	1	**2**	**3**	**4**	**5**
average number of characters per line	2	13	77	0	0
% spaces per line	92	0	0	0	0
% blank lines	92	0	0	0	0
% comments	3	0	63	21	5
% indentation	0	2	89	1	0
proportion of lines with 'good' indentation	0	32	60	0	0
av. length of user created variables	11	80	1	0	0
% good length of user created variables	84	7	1	0	0
spaces before pipe	0	1	5	0	0
spaces after pipe	0	1	5	0	0
spaces before semicolon	0	74	2	0	0
spaces after semicolon	0	2	73	0	1
spaces before >	0	2	88	2	0
spaces after >	0	0	87	5	0

1 = # < L, **2** = L ≤ # < S, **3** = S ≤ # ≤ F, **4** = F < # ≤ H, **5** = # > H

The table shows that all students scored nil in this assignment with regard to % spaces per line and % blank lines. This particular assignment was set at the beginning of the course and these nil results may be due to the fact that the program was small and spaces or blank lines were not that obviously needed, or that the students did not yet grasp the importance of using blank lines and spacing in their code. It should also be noticed that the total number of students shown in respect of results for the spaces before and after the pipe '|' , and the semicolon, did not add up to 92. This is because not all students used these markers in their program.

5. Conclusion

Experience gained in the development of a typographic assessment system has been generally positive. However, style assessment is not the only measure in marking student programs and it has to be complemented by other assessments; this will be discussed further in the following chapters. A further drawback which has not yet been easily remedied is that the system has no capability to reward and encourage those who, in their style, go beyond the original statement of the problem, or who find a unique and original way to approach it.

CHAPTER V

USING COMPLEXITY MEASURES FOR AUTOMATIC ASSESSMENT

In this chapter the complexity, another important measure for the assessment of the quality of shell programs, is considered. There are different categories of complexity measures, and measures may be used to cover any one of these categories except the computational complexity. Complexity measures are explained in Section 2. Section 3 briefly explains the scales of measurement. In Section 4 the proposed complexity measures are explained. This set of measures covers the properties that are usually considered when teaching students good programming practice.

1. Introduction

This chapter provides another important measure for the assessment of the quality of students' Shell programs. This is the measure of the complexity of the programs. The system uses a variety of complexity metrics to achieve a measure that reflects various important aspects of Shell programs. Complexity can, in fact, be considered an important factor in assessing students' programs and their learning of good programming practice. For example, it is known that there is a correlation between the use of control structures and complexity, where poor use increases, and good use decreases the complexity

(Henry and Kafura, 1981). McCabe (1976) correlated control flow complexity with reliability and frequency of errors. Since the member of staff teaching a course is usually anxious that her/his students learn high quality programming, complexity might be one measure by which she/he can assess their progress. An early attempt to use some form of complexity measurement in the assessment of students' programs may be found in Ceilidh system (Benford *et al.*, 1993 and Zin and Foxley, 1993). In the system described here, a best model program needs to be set by the teacher, with the least value of complexity by the set of measures used. Adopting the view that program quality is the 'inverse' of program complexity (Van Verth, 1985), in relation to a fixed problem, the student's program is then graded by measuring its complexity-ratio to this model.

2. Complexity Measures

There are several categories of software complexity. Ejiogu (1985) recognises five of them: structural, computational, logical, textual and conceptual. Structural complexity concerns the natural expression of the topological relationships of a system's components. Computational complexity is concerned with the relative difficulty in accomplishing arithmetical/logical computations of an algorithm or data. This complexity is an attribute of an algorithm and not an immediate attribute of the software. Logical complexity refers to the relative difficulty of logical decisions, or flows and branches within a system. Textual complexity concerns the static analysis of a program's source code. Finally, conceptual complexity has to do with the psychological perception or the relative difficulty of comprehending, undertaking or completing a system. That is, those characteristics of software which are considered to be synonymous with comprehensibility or maintainability (Curtis, 1979) and which affect programmer performance. The measures which are used in this work could cover any of the above complexities with the exception of computational complexity, as it is not considered in this study.

Furthermore, the overall complexity of software is a function of many factors. In literature many types of measures can be found, such as: process, product, resource, static, descriptive, quality, code, design, data flow, and information flow measures. In general, the measure of complexity should determine the degree of difficulty in analysing, maintaining, testing, designing and modifying software.

Currently, there are a large number of measures in the literature of software complexity and the number is rapidly growing. They are all based on different ideas of complexity. Examples of the best known complexity measures today are the measures of McCabe (1996) (Cyclomatic Number), Lines of Code (LOC) (Conte *et al.*, 1986), and the Measures of Halstead (Length, Volume, Difficulty and Effort) (Halstead, 1977). Other complexity measures include data flow and information flow measures (e.g. the data flow measure of Oviedo (1980), and the information flow measure of Henry and Kafura (1981). There are more than 100 other measures available to describe the complexity of programs.

The measures of McCabe and Halstead are the most discussed and widely accepted software complexity measures. Halstead's is based on the program source code, whereas McCabe's can be computed from both the program text and the flow graph.

The measure of Oviedo deals with control flow complexity and data flow complexity together. However in this work, only control flow complexity of programs is used since it difficult to interpret the results when they are together.

Furthermore, some of the measures overlap and are thus interrelated; some of the properties covered by one measure are also covered by another. As a single measure is not generally enough to cover all properties considered in one application, Howatt and Baker (1989) proposed that individual measures be made components of a vector of measures. This will provide complete information on each of the individual properties. Nevertheless, if a set of individual measures of some software properties are not highly interrelated in the sense mentioned above a weighted average can combine them into a single-value measure.

3. Scales of Measurement

It is not always possible to make arbitrary statistical operations with the values of these measures, such as calculating arithmetic means, variances and percentages, and applying statistical tests. The statistical operations which are possible depend on the type of scale that is connected with the measure, and thus it is important to choose the measure with the right scale for the calculation of a certain statistical operation. The arithmetic mean, which is the most common statistical operation, is one example. It can be shown that its calculation might not be unique for a set of measures if certain conditions for the scale are not met. The question as to what kind of scales exist for software complexity measures arises.

Five different scales of measurement could be adopted in software complexity measurement. These are (from weaker to stronger): nominal, ordinal, interval, ratio, and absolute.

The ***nominal*** scale is the simplest. For example, a complexity measure is defined on a nominal scale, with a value 0 if the program is structured and the value 1 if not. However, this scale is of trivial importance, as it does not contain much information for any meaningful statistical operation. On the other hand, the ***absolute*** scale is too powerful and might not be suitable for the description of many measurements. In the ***ordinal*** scale the complexities of different programs are ranked to provide a rather crude comparison. In the ***interval*** scale, the difference in complexity between two programs is expressed in units, for example, "Program P_1 is 4 units more complex than program P_2.". However, the use of the interval scale related to software complexity measures is usually difficult and intuitively not reasonable (Zuse and Bollman, 1989). In the ***ratio*** scale, the ratio of the complexities of two programs is determined. The ratio scale is flexible and has enough information to allow many statistical operations to be performed on the measure.

It is usually a goal of research in the measurement of software complexity to obtain as strong a scale as possible (Harrison *et al.*, 1982). For example, under some conditions, the Measure of McCabe provides an ordinal scale. Other conditions also allow the use of the Measure of McCabe as a ratio scale. The ratio scale is required in literature by several authors (Harrison *et al.*, 1982), both implicitly and explicitly, and is discussed as a goal of a scale in measurement theory (Harrison and Magel, 1981). Thus, the aim of the literature is to describe the measure as either a ratio scale or stronger, and at least as an ordinal scale (Roberts, 1979; Harrison and Magel, 1981; Elliot *et al.*, 1988). Zuse and others study the criteria for the use of the measure as a particular scale. Measurement theory, in particular, was used by Zuse (1991) to give the necessary and sufficient conditions for the use of this measure as ordinal or ratio scale.

By itself, Measurement theory cannot solve the problem of measuring the complexity of software. However, measurement theory gives hypotheses and conditions for the use of measures and the appropriated and connected scales with the measures. The Extensive Structure is the way to reach the ratio scale (Zuse, 1991). If a measure is an Extensive Structure it can be used as a ratio scale. A further requirement by the Extensive Structure is the weak order, which also means that the measure can be used as an ordinal scale. Having described a complexity measure as a ratio scale, 'reference programs' can be defined and the quotient of the complexity of the 'reference program' and other programs can be calculated. This means that either the ratios of the complexities of programs, or comparison between one version of a program with another, can be compared.

4. The System Measures

In general there is no known method for selecting sets of well-defined complexity measures that is suitable for every application in practice. This is partly due to the fact that there are no accepted criteria for the description of the properties and the scales of the measures (Ejiogu, 1985). Nevertheless, Zuse (1991) recently presented methods for characterising software complexity measures. These methods can be used for selecting, describing and evaluating software complexity measures that are required for analysing the complexity of a single program module.

There are several possibilities for selecting complexity measures from the wealth of measures available. This depends, among other factors, on the goal of the measure, and those properties of the program which are considered to be the most important to measure. Another factor is whether the measure can be used as a ratio scale. Since, it is the intention to measure the complexity of a program P relative to a model program P', the measures used would best be described as a ratio scale. Therefore, it is possible to express the increase or decrease of the complexity in comparison to the reference program.

Zuse (1991) has further recommended a set of measures which he called the *minimal set of standardised software complexity measures*. The set included measures which give basic information (e.g. LOC), measures of program source code (e.g. measures of Halstead), measures which are sensitive to nested structures, measures of flow of control paths (e.g. McCabe's measures), measures which capture loops effect, and measures to capture lack of structure. Many of the measures in this set can be used as a ratio scale, though some were described as nominal, ordinal, or absolute scale.

The set of measures used for this system is a subset of Zuse's set. It comprises 11 measures of complexity (C_1 to C_{11}). They are all described as a ratio scale (and thus can also be used as ordinal scale). These measures (with the exception of Halstead's measures) depend upon constructing the flow graph of the program, and thus their definitions are based on the definition of the flow graph.

A flow graph is the representation of the control flow of a program. It can be described by the quadruple $G = (E, N, s, t)$, that is a directed graph with a finite, nonempty set of nodes N, a finite, nonempty set of edges E, a unique start-node $s \in N$ and a unique exit-node $t \in N$. Additionally, the in-degree of s is $I(s)=0$, and the out-degree of t is $O(t)=0$.

The 11 measures are described as follows:

Basic information

This is the kind of measure that captures the simple properties of flow graph/program.

C1. Lines of code LOC (Conte *et al.*, 1986; Levitin, 1986): This could be defined in many ways dependent on the definition of what constitutes a line of code. For example, the line of code could be considered as the node in the corresponding flow graph. Thus, for a flow graph G, LOC(G) is the number of nodes in G. A more convenient definition is that of Conte *et al.* (1986). They define a line of code as any line of the program that is not a comment or blank, regardless of how many statements or fragments of statements are on the line or their type.

C2. A decision node (predicate) in a flow graph is a node with out-degree >1. The measure DEC, of Zuse (1991), uses the number of decision nodes as a measure of complexity. Let D be the set of decision nodes in the flow graph G, i.e. $D=\{d | d \in N \wedge O(d)>1\}$. Thus $DEC(G)= |D|$; the number of decision nodes.

Predicate or decision nodes in a flow graph are those nodes that correspond to selection predicate and loop-control in a real program.

Measures of Halstead

The measures of Halstead are appropriate to give information about the program source code. They are usually used in connection with other measures based on flow graphs. Let:

n_1: number of distinct operators,

n_2: number of distinct operands,

N_1: total number of operators &

N_2: total number of operands.

C3. Measure LENGTH, (Halstead, 1977): $N= N_1 + N_2$, the total number of operators and operands.

C4. Measure VOCABULARY, (Halstead, 1977): $VC= n_1 + n_2$, the total number of distinct operators and distinct operands.

A measure sensitive to nested structures: Predicate Execution Number

Nesting is widely considered as an aspect of programs which contributes to program complexity (Weyuker, 1988; Piwowarski, 1982; Evangelisti, 1984). Belady *et al.* (1980) argued that it is more difficult to construct, understand or maintain a program whose nodes are imbedded into multiple environments, than a program with less nesting. Thus, nested control structures are more complex than sequential control structures. The measure PEN (Howatt and Baker, 1985;

Howatt and Baker, 1989) called the Predicate Execution Number, is sensitive to nested structures. To define PEN the 'range of a node' first needs to be defined as follows.

Let IS(p)= $\{m|(p, m)\in E\}$ be the set of immediate successors of node p. Also let MP(n,m)= $\{v|\exists P[P\in FOP(v, m) \wedge v\in P]\}$, where P is a path in G, v is a node in P. FOP(n,m) is the First Occurrence Path from n to m which is the set of all paths from n to m such that node m occurs exactly once on each path. Thus MP(n,m) is the set of nodes that are on any path in FOP(n,m). Let LB(p), called the set of lower bounds of node p, be the set of nodes that lie on all paths from the immediate successors of a given node p to the terminal node t, i.e.

$$LB(p)= \{[v|\forall r\ \forall P[r\in IS(p) \wedge P\in FOP(r,t)\rightarrow v\in P]\}.$$

Also the greatest lower bound GLB(p) of node p is defined as the unique lower bound that precedes all other lower bounds in LB(p). Finally Range(p), called the range of a decision node p (pχD), as the set of nodes predicated by p is defined. i.e.:

$$Range(p)= \{n|\exists q|q\in IS(p) \wedge n\in MP(q,GLB(p))\}.$$

Thus Range(p) is the set of nodes that fall on any path between the immediate successors of p and the greatest lower bound of p. Let $Pred(n)=$ $\{p|n\in$ $Range(p)\}$ be the set of decision nodes p that predicate nodes n. This is the nesting level of nodes n. Thus, measure PEN is given by:

C5. PEN(G)= $\sum_{n\in N}|Pred(n)|$, the sum of the nesting level of nodes n of G

Measures of McCabe and similar

The following two measures evaluate the complexity of a program by counting the paths through which the control flows. The idea is that complexity increases as the number of linear paths of control flow increases. The two measures are not sensitive against nesting, loops, or structuredness of the flow graph. The effects of these are captured here by some of the other measures.

C6. The measure of McCabe (1976), calculates the number of independent paths in a flow graph, called The Cyclomatic Number. For a flow graph G, The Cyclomatic Number is given by $|E| - |N| + 1$, where $|E|$ & $|N|$ are respectively the number of edges and nodes in G.

C7. PATH(G)= Number of possible paths in a flow graph; this measure was suggested by Fenton (1994).

Measures which capture loops

Jayaprakash *et al.* (1987) suggested a number of properties for complexity measures. One of these properties is the ranking of the control structures. They suggested that a program having only sequential code should be treated as less complex than a program with a single selection structure, and in turn should be treated as less complex than a program with repetition structure.

Two measures to capture the loop effect in a flow graph are considered here. These are:

C8. Number of loops in a flow graph G, N-LOOPS by Hecht: N-LOOPS(G)= Number of backward edges in G.

C9. Sum of nesting level of nodes by loops: measure NL of Howatt *et al.* (Howatt, 1985, 1989).

Measures to capture unstructuredness

Piwowarski (1982) proposed a set of axioms for complexity. He stated that a structured program is considered to be less complex than an unstructured version of it. Several measures were devised to capture the unstructuredness in a flow graph; the following two were suggested by Zuse (1991). Their definitions are based on the concept of range in a flow graph as defined before.

C10. Number of nested pairs, UN(G) $=\sum_{p\in D}\sum_{q\in D}|NEST(p,q)|$; UN is the number of nested pairs (p,q) of ranges in G.

C11. UOV(G) = $\sum_{p\in D}\sum_{q\in D}|OVL(p,q)|/2$; the number of overlapped pairs *(p, q)* in G.

NEST (p,q) and *OVL(p,q)* are defined as follows:

NEST (p,q)= 1,	if *Range(p)* is nested in *Range(q), and*
= 0,	otherwise
OVL(p,q) = 1,	if *NEST (p,q)*=0 and *Range(p)* overlaps with *Range(q)*, and
= 0,	otherwise

5. Use of the system for marking

In the final calculation, the total complexity value is given by

$$C = \sum_i w_i C_i, \ 1 \le i \le 11$$

where the values of w_i are determined by what is believed to be the contribution of the measure C_i to the overall complexity. At the start, the weights $w_i, 1 \le i \le 11$ are fed into the system. The values of these weightings are determined by the lecturer as the students progress in their programming course. The system collects from the source code the information needed for measures C_3 and C_4. Also, the flow graph form of the program text is constructed for the rest of the measures. Finally the total complexity value C is calculated. C is then compared with the model complexity C' (which should have been calculated by the same procedure). The following simple model could be used to mark the student's program:

$M_1 \le C/C' \le M_2$	award the maximum mark Max,
$C/C' < M_1$	award more marks than Max in proportion to C/C',
$C/C' > M_2$	award less marks than Max. in proportion to C/C'.

Obviously the maximum is when $C/C' \approx 1$. The case of $C/C' < M_1$ is considered to reward students who write programs that are better (less complex) than the teacher's model. Again the teacher supplies the values of Max, M1 and M2.

Usually in the class students are encouraged to write less complex programs by following some rules or guidelines for good programming practices. Examples of these guidelines could be:

- Use fewer nesting of conditions.
- Use fewer nesting of loops.
- Use N-*case* statement instead of N-1 nested *if* statements.
- etc....

6. Conclusion

This chapter has demonstrated the use of complexity to assess the quality of programs in an educational environment. The proposed set of measures covers those properties that are usually considered when teaching students good programming practice. As in Chapter IV, the marking process is based on comparing the student's script with a best model supplied by the teacher, with the least value of complexity, deemed by the set of measures used. Adopting the view that, in relation to a fixed problem, program quality is the 'inverse' of program complexity, a student's program is then graded by measuring its complexity-ratio to this model.

When used for marking, the student is awarded a mark lower than the maximum in proportion to the increased complexity of his/her program against the teacher's model. However, the student will be rewarded if his/her program measures less complex (i.e. proves better) than the teacher's.

CHAPTER VI

AUTOMATIC TESTING OF SHELL PROGRAMS: MEASURING DYNAMIC CORRECTNESS

An automatic system for dynamic testing and then grading the correctness of Shell scripts is discussed below. Syntax correctness is a prerequisite if the script is to run and hence for the student to gain marks with this measure. The teacher will provide the system with both input test data and output data. The student program is graded by checking its output against the teacher's output data for the same input.

1. Introduction

In the early days of software development, testing was simply regarded as debugging. In late 1950's testing was separated from debugging and was regarded as finding bugs in the software. In those days Computer science programs were not concerned with software engineering or testing, but were dealing with numerical methods and algorithm development. In the 1980's developers realised the value of quality, and hence discussion of software engineering and testing followed.

Myers (1979) defines testing as 'the process of executing a program or system with the intent of finding errors', which focuses on what is done while testing. Hetzel (1973) defines testing as 'establishing confidence that the program does whatever it is supposed to do'. Another definition by Hetzel is 'the measurement of software quality'. This focuses on assessing program quality. Kit (1995), defines testing as both 'detecting specification errors and deviations from specifications' and also 'identifying differences between expected results and actual results, and confirming that a program performs its intended functions correctly'. Objectives of testing for assessment of program quality are aligned with the definitions of Hetzel and Kit.

As previously mentioned, the measure of the dynamic correctness of programs is measured. Here, correctness is defined as the degree to which a program is free from faults that prevent it from producing the correct result, as required by the problem statement, and the results of the model run provided by the lecturer. There are two main approaches to this type of testing. Functional, or black-box, testing where the tests are derived from the program specification, and structural, or white-box, testing where the tests are derived from knowledge of the program's structure and implementation.

The testing conducted for this thesis is functional, or black-box, testing, not structural. The outputs corresponding to certain inputs with respect to the model output is checked, as are the exceptional error conditions required by the problem statement. It was realised that white box testing is not needed for this system. Assignments given to students who are beginning to learn programming will usually produce programs that are much too simple to require white-box testing. It was also found that black-box testing is more effective in discovering faults (Sommerville, 1996). Programs receive certain input and their functionality is examined by observing the output. Of course, in black-box testing, not all the properties of the program output can be tested but it is possible to send out the errors to standard error, which is a special feature of UNIX Shell.

Students programs are tested against the test data supplied by the teacher and graded accordingly. As the system is used for educational purposes it should be adaptable to teacher needs as the student assignments progress from one stage to another. The teacher should be able to add to the test, checking off any missing function for example. In addition, it should be possible to provide both the teacher and the student with useful qualitative feedback.

Testing UNIX shell programs is somewhat different from testing programs in other procedural/imperative languages, such as Pascal for example. First of all, Shell is interpretive and not compiled. Secondly, a Shell program when running can create additional processes that can run in the background but which should also be considered in the testing processes. Thirdly, file handling in Shell is much more elaborate than in other languages, due to the use of wild cards and regular expressions.

2. Types of Program Testing

Program testing can be divided into two types: static analysis and dynamic testing. Static analysis involves the examination of the program source code without execution. The program source code structure and syntax are inspected so as to highlight static errors and produce statistical information for the programmer (Coleman and Pratt, 1996). Although compilation is a form of static analysis, the term 'static analysis' is normally used for activities intended to pick up other types of errors or potential error conditions, such as infinite loops, unreachable statements, conflicting conditions, improperly nested loops, and unused variables.

The ideal dynamic testing would be to test the program with all possible elements from the input domain, but this is obviously impossible in any real situation. Thus, testing can only be done in practice by using a small subset of data from the possible input domain. To ensure that as many potential errors as possible can be detected, the test data must be chosen carefully. Many techniques for selecting test data have been proposed. For example, using Data Flow Information (Rapps and Weyuker, 1985), Cn coverage measures (Meller, 1980), TERn measures (Woodward *et al.*, 1980) and boundary-interior testing (Howden, 1975).

Most testing processes are based on the assumption that an *oracle* is present. An oracle is a mechanism by which the correctness of the output can be checked (Weyuker, 1982; Zin and Foxley, 1992). Construction of an oracle is a difficult problem in any situation where the program output format is not exactly specified. Running a program against test data will produce output. Even in a simple student program, the number of possible outputs for the same input data could be large and the oracle must correctly interpr*et al.*l of these outputs. In some examples a procedure, rather than a complete program, may be the subject of the test. The problems of an oracle still exist, but are considerably simplified.

Another possible output from dynamic testing is some information about the program execution. This information, called a program profile, can be useful for the programmer. For example, it can be used for the identification of dynamically dead code, checking the number of loop iterations and helping to optimise the most frequently executed code segments.

The testing process thus involves the following activities:

a) program compilation,

b) static analysis,

c) for each set of test data:

   - execute the program against the data.
   - compare the output with the expected result.
   - analyse the output of the program profile.

3. The Unix Shells

Within UNIX environments there are a few Shells available. These are Bourne Shell which is the original Shell written by S.R. Bourne, C Shell, written at the Berkeley campus of California and Korn Shell written at AT&T. Bourne Shell is used throughout this work, as is the most common of all the Shells.

To support structured programming Bourne Shell supports both local and global variables; however global variables must be exported. Bourne Shell also offers constructs like if-then-else for decision making and looping constructs such as: for, while, and until. But it uses Unix utilities like test and expr to evaluate conditions of the expressions, either in loop constructs or in if-then-else constructs.

4. Handling Error Conditions

When any command is executed error messages can either be shown on the screen or by using the Shell input/output redirection, the error messages can be sent to an error file. Thus it is possible during the testing process to see whether any errors have occurred. This is usually done by typing:

```
2>errorfile
```

at the end of a command line. The '2' above indicates the stderr (standard error) which has a file descriptor equal to 2 (A file descriptor is a numeric that UNIX uses to identify the kind of file opened for processing).

In addition to redirecting stderr to an error file, there are two kinds of error conditions that can be checked in Shell scripts. Internal errors for instance, could happen due to the misspelling of commands or a lack of sufficient parameters for the commands, and so on. The others are external errors. These are sent to the Shell by various signals.

Internal errors could be considered to be user error conditions. There are two types; one type occurs when the wrong options and arguments are used or inputted to the Shell command. The other is due to command failures. For instance, if the programmer wants to execute a command which has no execute permission or where the file does not exist. In the testing program, the special shell built-in variable **$?** is checked for its value. If the command was successful then the return status of the command will be 0, otherwise it will be 1. Students lose marks if a command returns a status not equal to zero in any part of their program. This is done by printing the value of the **$?** Shell, variable by the echo command.

External errors, on the other hand, are caused by the actions of either the user or the system. Such errors will interrupt the Shell process. The UNIX system captures these errors and sends a signal to the Shell process, indicating that there is a problem and that certain actions may be necessary. It is possible to control this action by the help of the error handling trap command. The trap command allows for the selection of signals which need detecting, and an indication of the actions to be taken when such signals are received. Signals could be caused by any type of exit command, such as those caused by: interrupt, the delete key, or the Ctrl key. Students use the trap command to remove certain files when their processes receive 1-hangup, 2-interrupt, 3-quit, and/or 15-termination signals. This prevents a lot of temporary files occupying a lot of space. Students usually forget to take such actions. The lecturer reminds them when such an action is necessary. Even though it is useful to add such a control to mark actions to check whether temporary files are removed, in the marking schemes used here, checks were not made to ascertain whether the student had taken such actions. The testing program removes all such files automatically as a clean up action.

5. Testing Method

In this work only simple programs are assessed, the type given in an introductory course on Shell programming. Examples that run background processes were not included. In Shell it is possible to write scripts and run them as background tasks using commands such as at, sleep and those commands that begin with the & sign. This is one of the more difficult aspects of testing Shell Programs.

Not all programs can be tested in the same way. There are different situations. So the cases where each case has a different test drive were categorised. The course teacher sets up certain actions for each exercise and these are written in a file. Each column of this file is organised so as to give information about the test. For instance, the number of tests, the maximum marks for each test, and certain flags about the presence or absence of files used in the testing process. The teacher then sets up certain tests about each exercise to detect errors such as a set of valid input, a set of invalid input, and a combination of both. As a result of this, expected output and the actual output are observed. Test completion criteria require that all tests run to completion without any error being detected. A mark is calculated according to the decision of important points in each set of tests.

For each exercise in each section, in addition to preparing a file to represent the actions going to be taken, the course teacher writes Shell scripts to drive a student's program according to target items.

Tests and exercises may be divided into the following groups:

- those which use data file(s).
- those which need an executable shell drive to test but produce simple regular expression (RE), composed of a word or a sequence of words. The existence/non-existence of such an expression is tested. For instance, if an exercise needs certain arguments then the student is required to run her/his Shell program with these argument(s). She/he may also be required to test her/his program with no arguments to produce a regular expression such as "Needs arguments". The marking program checks whether such a regular expression (RE) is produced.

- those which need an executable Shell drive to test but produce longer regular expressions, for instance permutations of the string "the cat sat on the mat". An oracle will check whether a student's output matches the expected one.
- those which need an executable Shell drive to test but do not produce regular expressions. In this case the output is redirected to a file, which is difficult to check line by line for correctness. Then a facility in Shell programming, which compares two text files and detects the differences called diff command, is used.

In the above tests, checks were made to ascertain whether a student's program is producing a simple sequence which can easily be searched, or a longer RE which is can still be searched line by line. However, when longer text files are being considered, searching line by line is not feasible. First, the cmp command is used to check whether or not, after running their Shell scripts, the teacher's output file and the student's output are the same. If they are not the same, then the deduction marks will be determined after the result of the diff command.

- Exercises which use & and sleep command – background processes. It is difficult to test programs which contain such commands, but by checking the status of processes at certain times and printing the contents of $? it is possible to detect whether the program has errors or not. Also, by using the trap command it is possible to detect any interrupts and redirect the error to a file for checking. When such errors are detected marks will be deducted, although the work in this thesis did not test such cases.

6. The Oracle Program

The oracle is a program which recognises whether a given piece of text contains a particular required meaning. This type of activity is important in some areas of testing. It is used to check that the output from the dynamic tests of a program represent correct output, for instance, whether or not the program sources contain particular features and whether some of the answers contain certain words, or sequences of words.

7. Implementation

The implementation uses the UNIX concept of an RE (regular expression). REs are involved in many UNIX commands. A regular expression defines a set of one or more strings of characters. Several UNIX utilities, including ed, vi, emacs, grep and awk, use regular expressions to search in order to replace strings. A simple string of characters is a regular expression that defines one string of characters: itself. More complex regular expressions use letters and special characters to define different combinations of strings that are special to Shell programming. A regular expression is constructed to match any string it defines. REs are detected by using the grep family of commands and the sed stream editor.

The grep utility searches one or more files, line by line, for a pattern. The pattern can be a simple string or an RE. The grep utility takes various actions, specified by options, each time it finds a line that contains a match for the pattern. It takes its input from files specified on the command line or from its standard input.

The sed utility is a batch editor. Although sed commands are stored in a script file, simple sed commands can be given from the command line, as used in this work. The sed utility takes its input from the files which are specified on the command line or from its standard input. By default, sed copies lines from the file-list to its standard output, editing the lines by position within the file (line number) or context (pattern matching).

Only simple programs were tested here, so regular expressions in this case were simply in the form of strings. These are read from a file created by the teacher, who tells the program "oracle" what features are to be tested. This file of REs acts as an oracle. The program "oracle" uses grep and sed to detect the existence of these REs by analysing student programs. Then marks are given. Mark actions are also edited into the same file of RE to simplify output for the course teacher.

Oracles are used in several ways as a convenient method of checking general text._The program's output must be examined to see if the student has solved the given program correctly. The oracle program is made simpler according to how precisely the question is specified.

Consider the following exercise:

> Write a shell script where, as arguments, the names of two directories are given. The output should list the names of files which occur in both (directories).
>
> If there are more or less than two arguments, print the message
>
> "Needs exactly two arguments".
>
> If the arguments are not the names of directories, print the message
>
> "The arguments must be directory names".

In such an exercise the teacher is interested to see certain features, such as the printout of the message when or more or fewer arguments are given. In such a case only the occurrence of the first word "Needs" in the first case and "The arguments" in the second case are checked. Full marks are awarded – score 100 in this case – when such REs are found, and a score of zero is awarded when such REs are not found.

It is possible to use such oracles to check the number of occurrences of a word or a number before a score is decided. Also it is possible to mark multiple types of questions by checking REs like b$ which means that the answer is "b". Oracles for such cases were not used as this is beyond the scope of this work.

8. Conclusion

In general, a system for automatically testing and grading the dynamic correctness of programs is difficult to implement. However, this chapter has shown that this is feasible when such a system is intended to be used for automating the marking of programs written by students attending a programming course. Unix Shell programming was the case examined here although, due to some of its features, the problem of automating the testing of Unix Shell is not a trivial one.

Problems related to dynamic testing were examined only for simple programs. Programs including background processes, for example, were excluded. It is hoped that future research will include more cases and more powerful oracles to handle more complex output. Testing is done with the same students, but fewer students are considered.

Here, the aim has been to get the dynamic testing to work. The information sought has been whether or not the program is fulfilling the dynamic requirements.

CHAPTER VII

DISCUSSION OF RESULTS

This chapter briefly outlines the assessment system, explains how the assessment is performed and discusses the results of the typographic assessment and interpretation of the results. There are also comments on complexity, and finally testing.

1. Introduction

Two aspects of the quality of a program are assessed for maintainability, namely its typographic style and complexity, as mentioned in Chapter V. The third aspect assessed is the dynamic correctness of the program.

2. Typographic assessment

Typographic assessment is done using the method adopted by Rees (1982) - (see the figure reproduced in Chapter VI, page 48).

First, the score for each factor is collected from student Shell scripts. A mark is given based on each score. This mark is awarded according to the factors of Rees (1982). The points L, S, F, H have the following significance:

L: The point below which no mark is obtained.

S: The starting point of the ideal range.

F: The end point of the ideal range.

H: The point above which no mark is obtained.

The scores between S and F obtain the maximum mark, those between L and S and between F and H are calculated by interpolation according to their exact position within the range, and those outside the range (L, H) receive no marks. The values for L, S, F, H and the maximum mark for each are given as part of the assessment system. The optimal values for these factors are explained in Chapter IV. However, as part of the assessment, the tutor is able to make some changes to the values of the typographic factors. These changes may not be dramatic, but small deviations from the values discussed in Chapter IV occur are permissible.

3. Marking

Students receive exercises regularly. The tutor plans exercises according to the subjects given. The maximum mark changes for each exercise. At the beginning of the course students are given guidelines which explain how they should indent, how many spaces are considered optimal before and after special characters, and so on. These special characters have meaning and they instruct the Shell to do some specific actions. For instance '>' is called the output redirection operator. This means the command output is redirected to the file name followed by the '>'. The other important special character is the '|', the piping operator, which is used on a filter between the command lines on both of its sides. Also ';' is considered in this marking. It is known as the command line separator. For the typographic style of Shell programs it is important to leave one or two spaces before and after these characters for optimal readability and therefore comprehension.

Finally, students are awarded an overall mark which includes the typographic analysis, complexity and dynamic testing.

4. Analysis of Typographic Results

Shell programming exercises are prepared for novices to understand, experiment with, and test certain Shell features and commands. It is not expected that students will produce long scripts.

In the test study whose outcome is shown below, Results 1 and 2 were obtained by examining ninety programs. Results 1 represent the early exercises and Results 2 are taken from scripts written by students with more experience. The number of students falling into the five intervals were counted to see how many of them were able to obtain the optimal value for a particular factor, how many of them could not manage well and therefore received values which were either too low or too high, and so on. The distribution of the number of students for each interval is a good indication of how well the students are doing, and also to check their improvements. It shows whether they are leaving optimal spaces before and after certain operators, whether they are properly commenting their programs, and so on. Results 1 and 2 show that for the factors, '% of blank lines' and 'average spaces per line' there was not much improvement. This shows that novices need to be more strongly encouraged to take greater care to leave enough spacing, in order to reach the optimal value. However, improvement in areas such as commenting and indenting were observed as they produced more exercises. The number of students who made indent errors decreased. There was little change in the factors of average user-created variable length and therefore percentage of good user-created variable length. This was due to the fact that this course was given to novices for only one semester and that the exercises did not require very many user-created variables at that level. They were required to use built-in Shell variables such as '$?', '$*' and also the Shell variables '$1 $2 $3 ... $9' which store command line arguments. The variable '$*' contained the number of arguments on the command line. Students at a certain level in the course were required to demonstrate the use of such variables. The lengths of these variables were obviously much shorter than user-created variable names. Certainly, user-created variable names needed to be selected from meaningful names, so they had to be reasonably longer then Shell variables.

For the factor of average module length, all of the students for both Results 1 and Results 2 fell in the first interval. They used very short functions. There was insufficient time to use more complicated functions.

For the piping operator '|' there was an improvement. As their experience increased in Shell programming, more students attempted to use '|'. They also used spaces correctly. For the ';' semicolon operator, there was also improvement. In the first programs the students left hardly any spaces and most of them fell in the first interval, where no marks are given. Later on, more students appeared in the second interval. In the exercises where Results 1 and Results 2 were obtained there was no request to use the input redirection operator. So the number of students in all the intervals is zero. Of course, it is possible to prepare exercises which ask the students to demonstrate the use of '<'. Results1 and Results 2 show that for the factor '>' output redirection operator, there was an improvement in spacing before and after this operator.

Results 1 First exercises

DISTRIBUTION OF VALUES WITHIN THE METRICS

Interval 1 is :	value < l
interval 2	l <= value < s
interval 3	s<= value <= f
interval 4	f <= value <= h
interval 5	h > h value > h

av char per line

l	s	f	h
4	10	25	35

interval	num of stu.
1	20
2	10
3	50
4	10
5	2

ave spaces per line

l	s	f	h
8	10	30	35

interval	num of stu.
1	92
2	0
3	0
4	0
5	0

blank lines

l	s	f	h
8	15	30	35

interval	num of stu.
1	92
2	0
3	0
4	0
5	0

%comments

l	s	f	h
5	10	50	80

interval	num of stu.
1	3
2	63
3	10
4	12
5	3

%indentation

l	s	f	h
0	5	60	90

interval	num of stu.
1	80
2	5
3	4
4	3
5	0

%indenterror

l	s	f	h
60	70	80	90

interval	num of stu.
1	4
2	3
3	80
4	5
5	0

avr.
Usr.creat.var.len
l s f h
2 5 10 15

interval	num of stu.
1	11
2	80
3	1
4	0
5	0

%gd usr creat.var.len
l s f h
50 70 100 100

interval	num of stu.
1	88
2	2
3	1
4	0
5	0

Avr mod len
l s f h
3 5 15 35

interval	num of stu.
1	0
2	0
3	0
4	0
5	0

spaces before pipe
l s f h
0 1 2 4

interval	num of stu.
1	0
2	3
3	2
4	0
5	0

spaces after pipe
l s f h
0 1 2 4

interval	num of stu.
1	3
2	10
3	2
4	2
5	6

spaces before semi
l s f h
0 1 2 4

interval	num of stu.
1	70
2	12
3	0
4	0
5	0

spaces after semi
l s f h
0 1 2 4

interval	num of stu.
1	0
2	15
3	60
4	0
5	0

spaces before <
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	0
4	0
5	0

spaces after <
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	0
4	0
5	0

spaces before >
l s f h
0 1 2 4

interval	num of stu.
1	5
2	40
3	35
4	10
5	2

spaces after >
l s f h
0 1 2 4

interval	num of stu.
1	0
2	10
3	50
4	28
5	4

Results 2 After students have more experience

DISTRIBUTION OF VALUES WITHIN THE METRICS

Interval 1 is : value < l
interval 2 l <= value < s
interval 3 s<= value <= f
interval 4 f <= value <= h
interval 5 h > h value > h

av char per line
l s f h
4 10 25 35

interval	num of stu.
1	2
2	13
3	77
4	0
5	0

av spaces per line
l s f h
8 10 30 35

interval	num of stu.
1	92
2	0
3	0
4	0
5	0

blank lines
l s f h
8 15 30 35

interval	num of stu.
1	92
2	0
3	0
4	0
5	0

%comments
l s f h
5 10 50 80

interval	num of stu.
1	3
2	0
3	63
4	21
5	5

%indentation
l s f h
0 5 60 90

interval	num of stu.
1	0
2	2
3	89
4	1
5	0

%indenterror
l s f h
60 70 80 90

interval	num of stu.
1	89
2	1
3	2
4	0
5	0

avr. Usr.creat.var.len
l s f h
2 5 10 15

interval	num of stu.
1	11
2	80
3	1
4	0
5	0

%gd usr creat.var.len
l s f h
50 70 100 100

interval	num of stu.
1	84
2	7
3	1
4	0
5	0

avr.mod len
l s f h
3 5 15 35

interval	num of stu.
1	92
2	0
3	0
4	0
5	0

spaces before pipe
l s f h
0 1 2 4

interval	num of stu.
1	0
2	2
3	5
4	0
5	0

spaces after pipe
l s f h
0 1 2 4

interval	num of stu.
1	0
2	1
3	5
4	4
5	5

spaces before semi
l s f h
0 1 2 4

interval	num of stu.
1	0
2	74
3	2
4	0
5	0

spaces after semi
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	73
4	0
5	1

spaces before <
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	0
4	0
5	0

spaces after <
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	0
4	0
5	0

spaces before >
l s f h
0 1 2 4

interval	num of stu.
1	0
2	2
3	88
4	2
5	0

spaces after >
l s f h
0 1 2 4

interval	num of stu.
1	0
2	0
3	87
4	5
5	0

5. Analysis of Dynamic Correctness

Analysis of the results of dynamic correctness is not as significant as that of the typographic results. Here, the aim was to check whether the program was dynamically correct or not. It was necessary to determine whether the required output was produced, to test if the program stopped prematurely, and so on. Full marks were given if the required output was produced, and a partial mark given if the fulfilment of the requirement was not complete.

For each exercise, the tutor decided how many tests were needed, the requirements of each test, and the actions to be taken for each test.

Below is an example:

Question: Write a shell script which, given as arguments the names of two directories, will list the names of files that occur in both.

For test 1 in this case it is required to see if "Needs exactly two arguments" is the required result. For test 2, the result should have something like "must be directory names" according to the question. If a student gets these results then full marks will be awarded. If only one requirement is fulfilled, then a partial mark is given.

Analysis of Dynamic Correctness			
	Item	mark	out of
	Test 1	15	15
	Test 2	20	20

Score for dynamic correctness is 100 %

In the dynamic actions file the number of tests, mark for each file is written so that different tutors can access.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

In this thesis the development of a system for the automatic assessment of UNIX Shell programs in a teaching environment was examined. A combination of three methods for such assessment was chosen:

1. typographic analysis.
2. complexity measurement.
3. testing the dynamic correctness.

The argument for choosing these three methods is that, when combined, they could produce an effective assessment of student programs comparable to human grading when used for assessing students' coursework..

At the time of writing this thesis, a number of systems had already been designed for program assessments in an education environment. None of these systems used these three assessment methods together, with the exception of Ceilidh system. However, in Ceilidh system limited weight was given to complexity and only one complexity measure was used: McCabe's cyclomatic complexity.

Thus, for these three methods studied in this thesis the following conclusion may be made:

1. Typographic analysis is done to assess the (typographic) style of programs. Teaching students to style their programs typographically and improve their presentation is essential in learning programming languages, since it was found that 'properly' styled programs are more readable and therefore more comprehensible and maintainable. Hence, when marking students' programs, typographic style is one important factor that instructors should consider. Of course, typographic style on its own cannot be considered adequate for grading students' work and it has to be complemented by assessing the other two factors of quality in this system: complexity and dynamic correctness. The typographic assessment in this system is adaptable to reflect the change of emphasis as the course progresses. To achieve this, the instructor has the facility to change the values of the input parameters she/he supplies to the system.

   Results of the output from a typical run of this part of the assessment conducted here have been presented based on a class of students in a Shell programming course. The experience with the development of a typographic assessment system has been generally positive. Since it has been shown that realistic grading can be achieved, and useful information feedback can be obtained, it is reasonable to conclude that automation of the assessment of this quality factor is feasible.

   Results of the typographic assessment were quite satisfactory. These students were studying Shell programming for the first time, the results of which were as expected. In their first programs, they made mistakes. However, as they completed more exercises there was improvement in some of the factors such as commenting and indenting. There was no time to examine the use of more complicated functions or to make more use of user-created variable names. The objective mentioned in Chapter IV has been met.

2. A major contribution of this work is the consideration of the measures of complexity as an important factor for the assessment and grading of student coursework. A set of complexity measures was selected that is suitable for this objective. From the three parts constituting the system, this measure can be adaptable to any language, not only Shell. Generally the complexity measures used are based on the program flow graph and not its source code. Hence, it can be tailored to any particular language. However, this complexity set was limited to measure only the control flow complexity; data flow complexity was not addressed. Future work should include this type of complexity in the assessment process. Investigation in this direction opens many avenues for future research. Part of this may involve the development of metrics, which specifically take into account data flow complexity together with control flow complexity. At the moment metrics cover only one of them at a time (sometimes with little account for the other) and there is debate about how to put them together. The first complexity measures which were selected did not prove satisfactory, and a program had to be written to collect these measures. However, it was later realised that they were not sufficient to measure the complexity of students' Shell programs. It then seemed appropriate to introduce the complexity measures, which are mentioned in Chapter V. While at this stage, programs to test these measures have not been written, as an extension to this work this is one aspect to be developed, and the programs will be written.

3. Problems related to dynamic testing were examined only for simple programs. Programs including background processes, for example, were excluded. To run programs in the background, it is not necessary to wait for the command to finish before another command is run. Therefore it would be desirable to expand the scope of testing in future research so that testing can be entirely automated. Furthermore, a more powerful oracle must be developed to cover more complex output.

Finally two further important issues have not been covered in this thesis and future work should address them. These are

- Firstly, as systems used for coursework assessment are relatively new, only limited experience is available of their use in a real education environment, and in comparison with actual manual grading. Some researchers have reported their experience of using Ceilidh and other similar systems. More investigation and analysis is required. It should be noted that no such investigation was attempted for the system discussed here. An evaluation of the complete system being used in a real education environment is also required in the future.
- Secondly, an important factor for the success of software systems is the availability of a proper interface. The design of a user-friendly, as opposed to basic, interface was not intended in this work. The main emphasis was towards the construction of the basic parts of the assessment. However, for the system to be useful in the future, an extension should include the design of a useful and user-friendly interface.

Bibliography

Allum, R. D. (1996). 'Two Years of Ceilidh', *CTC'96 - Fourth Annual Conference on the Teaching of Computing*, Dublin.

Anand, N. (1998). 'Clarify Function', *ACM SIGPLAN Notices*, 23(6), 69-79.

Arab, M. (1992). 'Enhancing Program Comprehension: Formatting And Documenting', *ACM SIGPLAN Notices*, 27(2), 37-46.

Arthur, L. J. and Burns, T. (1996). *Unix Shell Programming*, New York, John Wiley & Sons Inc.

Astruchan, O. and Rodger, H. S. (1998). 'Animation, Visualization and Interaction in CS1 Assignments', *Proceedings of the 29th SIGCSE Technical Symposium on computer Science Education*, Feb. 25 – Mar 1, Atlanta Georgia, 307-311.

Baecker, R. M. and Marcus, A. (1990*). Human Factors And Typography For More Readable Programs*, New York, ACM Press.

Barnet, L., Casp, J., Green, D. and Kent, F. J. (1998). 'Design and Implementation of an Interactive Tutorial Framework', *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education on Courseware*, Feb25 – Mar 1, Atlanta Georgia.

Belady, L A.; Evangelisti, C.J. and Power, L.R. (1980). 'A Graphic Representation of Structured Programs', *IBM Sytems Journal*, 19(4), 542-555.

Benford, S.; Burke, E. And Foxley, E. (1993). 'Courseware to Support the Teaching of Programming', *LTR Report, Department of Computer Science, University Of Nottingham*, Nottingham, UK.

Benford, S.; Burke, E. and Foxley, E. (1995). 'Integrating Software Quality Assurance Into The Teaching of Programming', *Software Quality Assurance and Measurement: A Worldwide Perspective*, London, International Thomson Computer Press.

Benford, S.; Burke, E.; Foxley, E.; Gutteridge, N. and Zin, A. Mohd. (1993). 'The Ceilidh Courseware System', *LTR Report, Department of Computer Science, University of Nottingham*, Nottingham, UK.

Berry, R. E. and Meekings, B. A. E. (1985). 'A Style Analysis of C Programs', *Communications of the ACM*, 28(1), 80-88.

Boroni, C. M.; Goosey, F. W.; Ginder, M. T.; Rockford J., and Ross, A. (1998). 'Paradigm Shift ! The Internet, the Web, Browsers, Java and Future of Computer Science Education', *the Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education,* Atlanta, Georgia.

Boroni, C.; Goosey, F.; Grinder M.; Rockford, R., and Wissenbach, P. (1997). 'Web Lab - A Universal and Interactive Teaching, Learning, and Laboratory environment for the World Wide Web'; *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, 199 – 203.

Bourne, S. R. (1987), *An Introduction To The Unix Shell*, New York, AT & T Bell Laboratories.

Bourne, S. R. (1989*). Unix System V*, Reading, Mass, Addison- Wesley.

Canup, M. C. and Russell, L. (1998). 'Using Software to Solve Problems in Large Computing Courses', *Proceedings of the 29th SIGCSE, Technical Symposium on Computer Science Education on Courseware,* Feb. 25-March 1, Atlanta, Georgia.

Casey, D. (1998). 'Learning from or through the Web: Models of Web Based Education', *6th Annual Conference on Integrating Technology into computer Science Education*, Dublin City University, Ireland, 51-55

Cavano, J.P and McCall, J. A. (1978). 'A framework for the Measurement of Software Quality', *Proceedings of the ACM Software Quality Assurance Workshop*, 133-139.

Cole, D.; Weinwright, R. and Schoenefeld, D. (1998). 'Using Java to Develop web Based Tutorials', *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education on Courseware*, Atlanta Georgia.

Coleman, M and Pratt, S. (1996). Software Engineering for Students, New York: Chartwell-Bratt Ltd.

Conte, S. D.; Dunsmore, H. E. and Shen, V. Y. (1986). *Software Engineering Metrics and Model*, New York: Benjamin/Cummings Publishing Company.

Culwin, F. (1998). 'Web Hosted Assessment, Possibilities and Policy' *6th Annual Conference on Integrating Technology into computer Science Education*, Dublin City University, Ireland, 17-21 August.

Curtis, B. (1979) In Search of Software Complexity, *Workshop on Quantitative Software Models for Reliability*, 95-106.

Dagdilelis, V. and Satratzemi, M. (1998). 'DIDAGRAPH: Software for Teaching Graph Theory Algorithms', *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education on Courseware*, Feb. 25-March 1, Atlanta, Georgia

Edward K. (1995). *Software Testing in the Real World*, New York: Addison-Wesley Pub. Co.

Ejiogu, L O. (1985). 'A Simple Measure of Software Complexity', *Computer World.*

Elliot, J.; Fenton, N. E.; Linkman, S.; Markham, G. and Whitty, R. (1998). (Editors), 'Structured-Based Software Measurement' Alvey Project SE/069.

Evangelisti, V. M. (1984). 'An Analysis of Control Flow Complexity', COMPSAC84, Nov. 7-9, 388-396.

Fenton, N. (1994) 'Software Measurement: a necessary Scientific Basis', *IEEE Transaction on Software Engineering*, 20(3), 199-206.

Fenton, N. and Kposi, A. (1987). 'Metrics and Software Structure', *Information and Software Technology*, 29(6).

Gorla, N.; Benander, A. C. and Benander, B. A. (1990). 'Debugging Effort Estimation Using Software Metrics', *IEEE Transactions on Software Engineering*, Feb, Se-16, No. 2, 223-231.

Gunning, R. (1962.) *Techniques of clear writing*, New York, McGraw-Hill.

Halstead, M. H. (1997). Elements of Software Science, New York, Elsevier North-Holland.

Harrison, W. and Magel, K. (1981). 'A Complexity Measure Based on Nesting Level', *ACM SIGPLAN Notices*, 16(3), 63-74.

Harrison, W., Magel, K., Kluczny, R. and DeKock, A. (1982). 'Applying Software Complexity Metrics to Program Maintenance', *Computer*, 9.

Hartley, S. (1994). 'Animating Operating Systems Algorithms with XTANGO', *25th SIGCSE Technical Symposium on Computer Science Education*, 344-348.

Henry, S. and Kafura, D. (1981). 'Software Metrics Based on Information Flow', *IEEE Trans. on Software Engineering*, 7, 5, 510-518.

Herbert L. and Brumund, D. P. (1998). 'Tools for Web-based Sorting Animation, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, Feb 25-March 1, Atlanta, Georgia.

Hetch, M. S. (1977). *Flow Analysis of Computer Programs*, New York, Elsevier.

Hetzel, B. (1998). *Complete Guide to Software Testing*, New York: John Wiley and Sons, Inc.

Hetzel, W. (1973). Program Test Methods, London, Prentice Hall.

Hoggart, G. and Lockyer, M. (1996). 'An Automated Student Diagram Assessment System' *6th Annual Conference on the Teaching of Computing- 3rd Annual Conference on Integrating Technology*, Dublin, 122-124.

Howatt, J. W. and Baker, A. L. (1989). 'Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting', *The journal of Systems and Software*, 10, 139-150.

Howatt, J. W. and Baker, A. L.(1985). 'A New Perspective on Measuring Control Flow Complexity', Technical Report No. 85.

Howden, W. E. (1975). Methodology for the Generation of Test Data, IEEE Trans. Computing, 24, 554-560.

Inman, D.; Hayes, A. and Mole, D. (1996). 'Simple World Wide Web Authoring', *4th Annual Conference on the Teaching of Computing*, Dublin.

Jayaprakash, S., Lakshmanan, K. B. and Sihna, P. K. (1987). 'MEBOW: A Comprehensive Measure of Control Flow Complexity', *COMPSAC 87*.

John Pardoe, J and Vickers, P. (1994). 'Using a Prototype Program Assessment Tools', *All-Ireland Conference on the Teaching of Computing*, Dublin.

Kernighan, B. W. and Plauger, P. J. (1978). *The Elements of Programming Style*, New York, McGraw-Hill.

Khuri, S. and Sugono, Y. (1997). 'Animating Parsing Algorithms', *Proceedings of the 29th SIGCSE Technical Symposium on computer Science Education,* 29(1), 25-29, Atlanta, Georgia.

King, T. (1996). 'Issues Arising During the Continuous Computer Aided Assessment of Large Groups', *CTC'96 - Fourth Annual Conference on the Teaching of Computing*, Dublin.

Kit, E. (1995). Software Testing in the Real World, Improving the Process, Addison-Wesley Pub. Co.

Leinbaugh, D.W. (1980). 'Indenting For The Compiler', *ACM SIGPLAN Notices*, 15(5), 41-48.

Levitin, A. V. (1986). 'How to Measure Software Size, and How to Do Not', *COMPSAC86*, 314-818.

Lewis, S. F. (1997) 'Developing a Modula-2 course for Ceilidh', *CTC'97 - Fifth Annual Conference on the Teaching of Computing*, Dublin, 26-29 Aug.

Lou, B. (1996). *New models of Natural Language for Automated Assessment*, unpublished *Ph.D. Thesis*, University of Nottingham, UK.

Marshall, D. (1997). 'Using the Internet to teach the Internet', 6th Annual Conference on the Teaching of Computing- 3rd Annual Conference on Integrating Technology, Dublin City University.

Mason, D. V., Denise, M. and Woit, D. M. (1998). 'Integrating Technology into Computer Science Examinations', *the Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education,* Feb 25-March 1, Atlanta, Georgia.

McAlpin, D. A., O'Docherty, B. A., O'Doneghue, P. G. and Teague, K. G (1997) 'Flexible Automatic Assessment of Clarity and Style of Student's Modula2 Programs', *CTI'97*, Ulster.

McCabe, E.M. and Troise, I. (1996). 'An Integrated Approach to Computer Aided Assessment', *CTC'96 - Fourth Annual Conference on the Teaching Of Computing,* Dublin.

McCabe, T. (1976). 'A Complexity Measure', *IEEE Trans. on Software Engineering,* Vol. SE-1(3), pp. 312-327.

McCabe, T., (1976). 'A Complexity Measure,' *IEEE Transactions on Soft. Engineering,* SE-2(12), 308-320.

McCall, J., Richards, P. and Walters, G. (1977). 'Factors in Software Quality', RADC TR-77-369.

McCall, Richards, J. P. and Walters, G. (1977). 'Factors in Software Quality', three volumes, NTIS A049-014,015,055.

MConnel, S. (1993). *Code Complete: A Practical Handbook of Software Construction*, Redmond, Washington: Microsoft Press.

Meller, E. (1980). 'Coverage Measure Definitions Reviewed' Testing Tech. Newsletter, 3, pp. 6.

Miara, R. J.; Musselman, J. A., Navarro, J A. and Shneiderman, B. (1983). 'Program Indentation and Comprehensibility', *Communications of The ACM,* Nov., 26(11), 861-867.

Mitrovic, A. (1998). 'Learning SQL with a Computerized Tutor', *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, Atlanta, Georgia,

Myers, G.J. (1979). *The Art of software Testing,* New York, Addison-Wesley Pub. Co.

Mynatt, B. T. (1990.) 'Why Program Comprehension Is (Or Is Not) Affected By Surface Features', *Human-Computer Interaction Conference 90,* 945-950.

Naps, L.T (1990). 'Algorithm Visualization in Computer Science Laboratories', *Proceedings of the 21st SIGCSE technical Symposium on Computer Science Education*, 22(1), 105-110.

Naps, L.T. and Bressler, E. (1998). 'A Multiwindowed environment for Simultaneous visualization of related algorithms on the WWW', *Proceedings of the 29th SIGCSE*

Technical Symposium on Computer Science Education, Feb25 – Mar 1, Atlanta, Georgia.

Nulden, U. (1998) 'The Excon Project : Advocating Continuous Examination', *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education on Courseware,* Feb 25-March 1, Atlanta, Georgia.

Oliver, R.G. (1996). 'Automatic Assessment of Programming Assignments by Inspection of Their Output Using PEST and Awk', *CTC'96 - Fourth Annual Conference on the Teaching of Computing*, 27-30 August.

Oman, P. W. and Cook, C. R. (1998). 'A Paradigm For Programming Style Research', *ACM SIGPLAN Notices*, 23(12), 69-78.

Oman, P. W. and Thomas, E. J. (1990) 'A Bibliography of Programming Style', *ACM SIGPLAN Notices*, 25(2), 7-16.

Oviedo, E. I. (1980). 'Control Flow, Data Flow and Programmers Complexity', *Proc. of COMPSAC 80,* Chicago IL, 146-152.

Parkash, A., Wei-Tek, T., Yamaura, T. and Anupam, B., (1985). 'Metrics Guided Methodology' *COMPSAC*, 111-120.

Pierson, C. W. and Rodger, H. S. (1997). 'Web-based Animation of Data Structures Using JAWAA', *Proceedings of the 28th SIGCSE Technical Symposium onComputer Science Education*, 29, 1, March.

Piwowarski, P. (1982). 'A Nesting Complexity Measure', *ACM SIGPLAN Notices*, Vol. 17, No. 9, pp. 44-50.

Rapps, S. and Weyuker, E. J. (1985). 'Selecting Software Test Data Using Data Flow Information', *IEEE Trans. Software Engineering*, 11(4), 367-375.

Redish, K. A. and Smyth, W. F. (1987.) 'Evaluating Measures of Program Quality,' *The Computer J.*, 30(3).

Rees, M.J.(1982). 'Automatic Assessment Aids for Pascal Programs', *ACM SIGPLAN Notices,* 17(10), 33-42.

Rimmer, A. (1997). 'SPROUT and the Automatic Generation of Alternative Program Statements', *CTC'97-Fifth Annual Conference on the Teaching of Computing.*

Roberts, F. S. (1979). 'Measurement Theory with Application to Decision-Making, Utility, and the Social Sciences', *Encyclopaedia of Mathematics and its Applications,* New York: Addison Wesley.

Rodger, S. (1995). 'An interactive Lecture Approach to Teaching Computer Science', *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education,* 1995, 278-282.

Salman, A. ' Automatic Testing of Shell Programs', to be published.

Salman, A., 'Automatic Measurement of Complexity of Shell Scripts'', to be published.

Sangwan, S. R.; Korsh, J. F. and La Follette, P. S. Jr, (1998).'A System for Program Visualisation in the Classroom', *Proceedings of the 29th SIGCSE Technical Symposium on computer Science Education*, Feb25 – Mar 1, Atlanta, Georgia.

Shen, V. Y. (1985). 'Identifying Error-prone Software—An Empirical Study', *IEEE Transactions on Soft. Eng.*, SE-11(4), 317-324.

Shneiderman, B. (1980). Software Psychology: *Human Factors In Computer and Information Systems,* Cambridge Mass: Winthrop.

Sobel, M.G. (1995). *UNIX System: A Practical Guide*, RedwoodCity: The Benjamin/Cummings Publishing Company, Inc.

Sommerville, I. (1998). Software Engineering, New York: Addison Wesley Longman Limited.

Stasko, J. T. (1990). 'Tango, a framework and System for Algorithm Aimation', *IEEE Computer,* 23(9), 39-44.

Stasto, J. T. (1997). 'Using Student-Build Algorithm Animations as Learning Aids', *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, March, 25-29.

Tai, K. C. (1984). 'A Program Complexity Metric Based on Data Flow Information in Control Graphs', *Proceedings of the 7th International Conference on Soft. Eng.*, IEEE Los Alamitos, Calif.: Computer Society Press.

Teasley, B. E. (1994). 'The Effects of Naming Style and Expertise on Program Comprehension', *Int. J. Human-Comp. Studies,* 40, 757-770.

Thorburn, G. and Rowe G. (1997). 'Assessing the Assessment System - an Evaluation of the Use of PASS, an Automated Program Assessment System' *CTC'97 - Fifth Annual Conference on the Teaching of Computing*, 26-29 Aug.

Thorburn, G. and Rowe, G (1996). 'PASS – An Automated Program Assessment System'*: CTC'96 - Fourth Annual Conference on the Teaching of Computing,* 27-30 August.

Tsai, W. T.; Lopez, M. A.; Rodriguez, V. and Volvik, D. (1986). 'An Approach Measuring Data Structure Complexity', *COMPSAC86,* 240-246.

Turton, B. C. H. (1996). 'A Computer-Aided Continuous Assessment System', *Association For Learning', Technology Journal,* Alt-J 4(2), 48-60.

Van Verth, P. B. (1985). 'A System for Automatically Grading Program Quality', *SUNY (Buffalo) Technical Report* No. 85-05.

Van Verth, P. B. (1985). 'A System for Automatically Grading Program Quality', SUNY (Buffalo) Technical Report No. 85-05.

Wang, P. S. (1988). An Introduction to Berkeley Unix, California: Wadsworth Publishing Company.

Weyuker E. J. (1988). 'Evaluating Software Complexity Measures', *IEEE Transactions on Soft. Eng.*, 14(9).

Weyuker, E. J. (1985). 'Evaluating Software Complexity Measures', *TR #149, Courant Institute of Mathematical Sciences*, New York.

Weyuker, E.W. (1982). 'On Testing Non-Testable Program', *The Computer Journal*, 25(4), 465-470.

Wilson, J. and Aiken, R. (1996). 'Review of Animation Systems for Algorithm Understanding', *Proceedings on Integrating Technology into Computer Science Education*, 75-77.

Woodfield, S.; Dunsmore, H. and Shen, V. (1981). 'The Effect Of Modularization and Comments on Program Comprehension', *Proceedings of The 5th International Conference on Software Engineering,* IEEE Computer Society Press, Los Alamitos Ca, 215-223.

Woodward, M. R.; Hedley, D. and Hennel, M. A. (1980). 'Experience with Path Analysis and Testing of Programs', *IEEE Trans. Software Engineering*, 6, 278-286.

You, S. S. and Collofello, J. S (1980). 'Some Stability Measures for Software Maintenance', *IEEE transactions on Software Engineering*, 6(6), 545-552.

Zin, A and Foxley, E. (1991). 'Automatic Program Quality Assessment System', *Proceedings of the IFIP Conference on Software Quality*, S P University, Vidyanagar, India.

Zin, A. M. and Foxley, E.(1992). The Oracle Program, *LTR Report, Computer Science Dept.,* Nottingham University, UK.

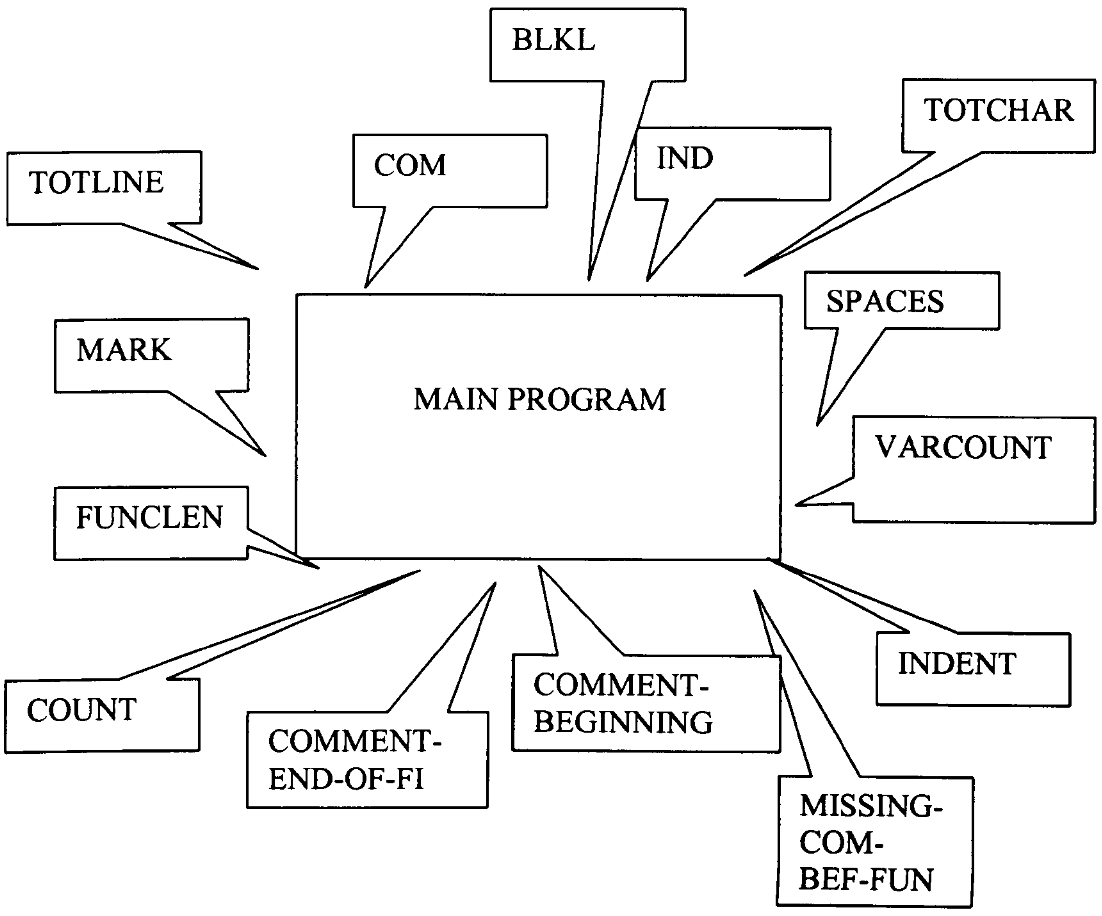
Zuse, H. (1991). *Software Complexity: Measures and Methods*, Berlin: Walter De Gruyter.

Zuse, H. and Bollmann, P. (1989). 'Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics', *ACM SIGPLAN Notices*, 24(8), 23-33.

All the programs were written in ANSI-C and also in Bourne Shell.

APPENDIX 1

Main program and the procedures



Description of the program

Initialise all the variables

```
Start of the main program
Check verbosity to produce different results
While the EOF is not reached
    Read the test file
    Open the file (which contains names of student files as a single column
            of names )
    calculate all comments  (com* )
    calculate comments at the beginning of a function (missing –comm-bef-
        fun )
    calculate comments at the end of each structure (comment-end-of-fi )
    calculate indentation errors ( indent )  and its percentage to total lines of
        code ( totline )
    calculate the length of user created variable names  (varcount )
    calculate the number of user created variables  (varcount )
    calculate the percentage of indents ( indent )
    calculate average user created variable length
    if verbosity > 3
        then print all the totals
        open an output file output for writing
```

```
                    print all the totals into 'output'

                    close output

if verbosity > 1 open another file for outputting  all the results and also marks obtained
corresponding to this values for each factor
                    print headings

                    for the metric pointer  'p'  equals or points to the first element of

                    the metric ( which is one of the factors mentioned in this thesis )

                          if the first sting is  ( do string comparison )  'ACPL'

                     (  average characters per line )

                                      print the value obtained for this factor, calculate and

                                      print the mark corresponding to this factor ( by

                                      calling the procedure mark to collect marks )

                                      repeat the above algorithm for the rest of the

                                      factors:

                                            'BLL'  blank lines

                                            'ASPL' average spaces per line

                                            '%COM'

                                            ....

                                            ....

                                            and the rest of the factors

              if verbosity > 0

              then do the above algorithm this time to calculate the cumulatives of the

              totals

              end of while  ( which reads from the file 'testfile' )

              close the files opened
```

[1]• all these names written in brackets are the names of procedures.

main program end here

Appendix 1 programs take care the typographic analysis of students' Shell programs. After compiling, the programs are run by typing the following command line:

```
a.out -v2 name
```

where a.out is the command to run the compiled program. v1,v2,v3,v4, are flags fed in to the command line which means verbosity: v1 produces single total mark for the typographic analysing of the program, v2 produces marks for each metrics, v3 is the same as v2 and v4 in addition displays weak points for each factor also.

Metrics values are as explained in Chapter 3. For instance the l,s,f,h values of each factor are defined by using the structure facility of the C language.

The Shell scripts of students are kept in a file called 'testfile' for typographic assessment. Each file in turn is opened for reading. The following procedures collect the values to be marked for typographic assessment.

In order to do these, the following procedures are used:

<u>**totline**</u>

Counts the number of total lines in the program.

<u>**com**</u>

Calculates the percentage of comments. In a Shell script, any line starting with the # sign is a comment. So those lines are counted and compared to total number of lines.

<u>**blkl**</u>

Calculates empty lines.

<u>**ind**</u>

Counts the number of indents. If the line starts with a blank character it is counted as indent, also tab is counted as indent. More detailed indentation is also calculated in another function.

<u>**totchar**</u>

Calculates the total number of characters.

spaces

Calculates the space characters used. This should not be confused with blank lines used.

Varcount

In Shell programming variable names are preceded by the '$' sign. So words which start with a '$' are detected and the length of the variable is calculated with the built-in function strlen (). Please note that only user defined variables are being calculated and detected, not the Shell built-in variables which also start with a '$' sign; such as $1,$2, and so on.

mark

Calculates the score according to the values of l,s,f, and h.

funclen

Counts the number of lines in a function. Comments are not included. In Shell programs special characters like \ ,\\ are significant and they have special meaning. Functions start with a curly, and end with a curyl in Shell programs. But these curlys could also be inside \ or \\ (single or double back quotes), so such cases are checked inside the program. Statements and empty lines are added together to calculate the function length.

count

Counts the spaces before and after special characters such as | (symbol for piping), <, >, >> ,<< (input output redirection operators). It should be checked that these signs are not in back quotes (\, \\ ).

comment-end-of-fi

This function measures comments with more details. Comments at the end of if structure, loops must be commented. This is checked and counted.

comment-beginning

It is recommended that students make comments at the beginning of each Shell program. There should be at least one comment. Bourne Shell scripts start with '#!' signs. This is checked first and if it does not appear in the , no marks are given.

missing-comm-bef-fun

Checks whether a function is commented or not.

indent

Checks whether *if-then-else-fi*, *while-do-done*, *for-do-done*, words are all at the same column for each construct. Every time an *if-while-until-for* is found, the corresponding construct count is incremented. *fi-done* are the ending keywords for constructs; when these are found the corresponding construct count is decremented. This is done to check the indentation of inner loops.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>



  /*      Typographic Analysis of a Shell script ( Bourne Shell ) 2nd Version

    Program reads input from a file called input; input isjust the name
     it could be the name of the text to be examined .
    Program is run as follows:

             eg.      a.out -v2 name

   ( 'name' is a file consisting of filenames of Shell scrips to be marked.
      Each line of 'name' contains only name of one Shell script.
      We could either examine & mark only one script or more if we want.
     'name' should look like :

                              CmpPr
                        CAudit
                        CAddSt
                        C....
                        C....

         each line of above file must  start from the begin. of each line..

      The file named "ouput" will contain the cumulatives of the values
      of  features if

                         _________________
                         | flag  is   -v4 |
                         |_ _____________|

       and also  typographic results  of each script ( if marking more )
     will appear according to the value  of the flag.
       It is easier to read if "output" is deleted before each
       run, so the results of each run can be seen clearly.



       Flags are as in "ef-typog" .


 verbosity  1   :single result
 verbosity  2   :each metrics
 verbosity  3   :as 2
 verbosity  4   :computed values

 More flags and values to be added soon.

 Remindr1!!!! In function funclen when a func is found the
              ending } will be counted as a statem. and as
              a line  for the time being, maybe all the time

 Reminder2!!!! When going through the metrics if the code did not
               exist, eg. student did not use '|' or '<', so on
           the code will appear for the time being.

 Reminder3!!!! (  spaces before and after 2>err not done ;rest all done
  Reminder4!!!! ( Do not forget to add comments and %NGL calculations
                  to total and cumulative sums ...

  Reminder5!!!!  For COMMENTS only  constructs are  measured.

  Reminder 6!!!! If the first line does not contain #! /bin/sh, then
                  no marks will be given to that program!!
                                                               */



#define OUT 2      /* outside word  */
#define NPARAM (sizeof metrics/sizeof metrics[0] )
#define INDBLEQUOT 1  /* inside dblequot         */
#define OUTDBLEQUOT 2
#define INESC 1       /* inside '/'          */
#define OUTESC 2

int feedback   ;
int count_func ;
int statem_count ;
int newline_count, before_semi, after_semi, before_pipe, after_pipe ,
                   before_input, after_input, before_output, after_output,
                   count_semi,  count_pipe, count_input,  count_output ;
int no_good_usrvar ;
int end_of_fi_esac_done, comments_after_fi_esac_done ;
int feeedback ;
float lost , max_lost ;
char *max_lost_mess ;

/*  Defof functions   */

       int  totchar               (FILE *ft);
      int totline                 (FILE *ft);
      int blkl                    (FILE *ft);
      int ind                     (FILE *ft) ;
      int spaces                  (FILE *ft) ;
      int com                     (FILE *ft);
      int varcount                (FILE *ft, char name[] );
      int funclen                 (FILE *ft ) ;
      float mark                  (float value ) ;
      void count                  (FILE *ft ) ;
      void comment_end_of_fi      (FILE *ft ) ;
      int comment_beginning       (FILE *ft ) ;
      int comment_all             (FILE *ft ) ;
      int missing_comm_bef_func   (FILE *ft ) ;
      int indent                  (FILE *ft ) ;
      void PreProcessLine         (char * c)  ;

      struct metric {
         int l ;
          int s ;
        int f ;
        int h ;
        int maxval        ;
        char *code        ;
        char *info        ;
          int min           ;
          char *lowmessage  ;
        char *highmessage ;
      } metrics[] =  {
          4, 10, 25, 35, 20, "ACPL", "Average char per line",
     20, "Not enough characters per line", "Too many char per line",
        0, 30, 35, 50, 20, "%BLL", "% blank lines",
     20, "Not enough blank lines", "Too many blank lines",
        0, 7, 15, 30, 15, "ASPL", "Average spaces per line",
     15, "Not enough spacing", "Too much space per line",
        25, 90, 100, 120, 25, "%COM", "% of comments done accordingly",
     25, "Not enough comments", "Too many comments",
          0, 5, 60, 90, 20, "%IND", "%indentation",
     20, "Too little indentation", "Too many unnecassary indents",
          0, 30, 100, 100, -20, "%INDERR", "%indent errors",
     0, "Indentation errors", "constructs indent errors",
          4, 5, 10, 15, 15, "AIDL", "Average identifier length",
     15, "Too short identifier", "Too long identifier",
          50, 70, 100, 100, 5, "%NGL", "Names with good length",
     5, "Too few identifiers with good length", "Too many identifiers",
          3, 5, 15, 35, 20, "AMDL", "Average module length",
     20, "Too short function", "Function too long",
          0, 2, 3, 5, 5, "SBEPI", "Spaces before pipe",
     5, "Too little spaces bef. pipe", "Too many spaces bef. pipe",
          0, 2, 3, 5, 5, "SAFPI", "Spaces after pipe",
     5, "Too little spaces after pipe", "Too many spaces after pipe",
          0, 2, 3, 5, 5, "SBESE", "Spaces before semicol",
     5, "Too little spaces bef. semicol", "Too many spaces bef. semicol",
          0, 2, 3, 5, 5, "SAFSE", "Spaces after semicol",
     5, "Too little spaces after semicol", "Too many spaces after semicol",
          0, 2, 3, 5, 5, "SBEIN", "Spaces before input",
     5, "Too little spaces bef. input", "Too many spaces bef. input",
          0, 2, 3, 5, 5, "SAFIN", "Spaces after input",
     5, "Too little spaces after input", "Too many spaces afer input",
          0, 2, 3, 5, 5, "SBEOU", "Spaces before output",
     5, "Too little spaces bef. output", "Too many spaces bef. output",
          0, 2, 3, 5, 5, "SAFOU", "Spaces after output",
     5, "Too little spaces after output", "Too many spaces after output",

       };

     struct metric *p ;
     struct metric metrics[NPARAM] ;

main(int argc, char *argv[] )

{
  extern int count_func ; extern int statem_count ;
  extern int newline_count ;

  extern int before_pipe , after_pipe, before_semi ,after_semi ,
             before_input, after_input, before_output, after_output,
             count_semi, count_pipe, count_input, count_output ;


  extern int no_good_usrvar ;

  extern int end_of_fi_esac_done, comments_after_fi_esac_done ;
  extern int feedback ;
  extern float lost, max_lost ;
  extern char *max_lost_mess ;

  FILE *fh ; FILE *fd ; FILE *ft ; FILE *fout ; char *aboutput ;
  float avchar, pblank, pind, avspace, pcom, pergood_ident,
pcom_after_fi_esac_done, avusrvarlen, tot_funclen,tot_mark, pind_err ,
avspace_before_pipe,
  avspace_after_pipe, avspace_before_output, avspace_after_output,
  avspace_before_input, avspace_after_input, avspace_before_semi,
  avspace_after_semi ;

  float sum_totline, sum_totchar, sum_blankli,sum_indent, sum_space,sum_com,
  sum_usrvar, avfunclen, sum_funclen ;


  int usrvarlen ; int sumlen; int usrvarcount,out_of ;
  int verbosity = 0 ;
  int write_val ;
  int no_statem ;
  int numberli ; /* lines in filenames file  */
  int totc ;
  int func_comments ;
  int other_comments ;
  int good_ident = 0 ;
  char name[64] ; char testfile[30] ;

   sumlen = 0 ;
  usrvarcount = 0 ;
  pind = 0 ;
  pergood_ident = 0 ;
  out_of = 0 ;
  tot_mark = 0 ;
  numberli = 0 ;
  avfunclen = 0 ;
  tot_funclen = 0 ;
  sum_totline =sum_totchar=sum_blankli=sum_indent=sum_space=sum_com = sum_usrvar
= sum_funclen = 0 ;
  count_func = 0  ;
  statem_count = 0 ;
  no_statem = 0 ;
  comments_after_fi_esac_done = 0 ;
  totc = 0 ;
  func_comments = 0 ;
  other_comments = 0 ;

  after_semi = 0 ;

  if(argc == 0)  {
    fprintf(stderr, "please specify a file\n") ;
    exit(1) ;
  }

 printf("argc = %d argv = %s %s\n", argc, argv[1], argv[2]);

/* we type ab-typog -anyflag input file ; inputfile=fh=argv[2]=argv[argc-1]
    where inthe my command line we have 3 arguments   */


  fh=fopen(argv[argc-1],"r" ) ;
  if(fh==NULL ) {
    printf("unabale to read") ;
    exit(2) ;
  }
```

```
     write_val=0 ;

/*  THIS PART IS FOR HANDLING FLAGS      */

  while (argc > 2 && argv[argc-2][0]=='-' )   {


   switch( argv[argc-2][1] )   {

   case 'v' :
     if( argv[argc-2][2] ) {

  verbosity = atoi(argv[argc-2]+2) ;

   /* atoi is being applied to the   flag which is argv[1]=argv[argc-2]
        argc=3       */

    } else {
       verbosity++ ;
    }
   break ;



   case 'F'   :
    if (argv[argc-2][2] )
       feedback = atoi(argv[argc-2]+2 ) ;
    else
       feedback++ ;
    break ;



  case 'w' :
    write_val = 1 ;
    break ;

   default :
        fprintf(stderr,"Invalid Flag\"%c\"\n",argv[argc-2][1] ) ;

  }

  argc-- ;

  printf("Verbosity = %d\n", verbosity);

  }  /* end of flag arguments while loop  */


  /* here we read each test file which is written in argv[argc-1] whicg is fh*/
  /* each line of fh contains strings of names; these names are stored in
     testfile as a string, and are treated as filenames.First fh is read
     by fscanf.Each name contained in fh as a char string is passed to  fopen a
s an argument , and then read by fscanf and scores collected. */

   /*  while (fgets (testfile,29,fh)!=NULL )
    {   ft = fopen (testfile,"r") ;
        printf("open file") ;
        if(ft==NULL) {
        printf("unable to read") ;
        exit(3) ;
```

```
      }   */


      while ( fscanf(fh,"%s",testfile) !=EOF )
    {  ft = fopen(testfile,"r") ;
       printf("%s\n",testfile) ;
       if(ft==NULL) {
       printf("unable to read ft") ;
       exit(3) ;
       }
   /* spaces are not included in avchar                        */
    avchar = (float)(totchar(ft )-spaces(ft))/totline( ft )    ;

  /* pblank =(float)blkl( ft ) /totline(ft )*100 ;cancelled this calc.*/

   avspace =(float)spaces(ft)/totline( ft ) ;

   pind =(float)ind( ft )/totline( ft )*100  ;



 /*  pcom =(float)com( ft ) /totline(ft)*100 ;  cancelled this calc. */
   tot_funclen+=(float)funclen(ft) ;
   if(count_func > 0 )
     avfunclen = tot_funclen/count_func  ;
   no_statem = newline_count + statem_count ;

   /* blank lines are compared to num. of statements, but
      not to total lines                                         */
    pblank = (float)blkl(ft)/no_statem*100 ;

   count( ft ) ;
   /* call count(ft) then calculate av spaces before and after pipes,
      < , > ; so on . Mark the avspaces values according to metrics */

   avspace_before_pipe =   (float) before_pipe/count_pipe ;
   avspace_after_pipe =    (float) after_pipe/count_pipe ;
   avspace_before_output = (float) before_output/count_output ;
   avspace_after_output =  (float) after_output/count_output ;
   avspace_before_input =  (float) before_input/count_input ;
   avspace_after_input  =  (float) after_input/count_input ;
   avspace_before_semi =   (float) before_semi/count_semi ;
   avspace_after_semi =    (float) after_semi/count_semi ;




   /* calculation of comments are here;  we aim to calculate
      pcom= (a +  b ) /num of constructs including func and main where
                   comm are needed
          a is other_comments put additionally; we later consider to
            give little extra marks to these as well not now

        b is the num. of  comments found that are placed correctly

          the value pcom obtained will be marked

        func_comments is the num of functions that are commented;
          student will make a mistKE AND WON'T COMMENT  some  functions
          function_comment calculates this.
          missing_comm_before_func(ft) returns missing_marks only if func
          i s not commented; note that a negative value is returned...
                                                                       */
    if(comment_beginning(ft) == 10)
    /*  fscanf(fh,"s",testfile) == EOF ; */
     continue ;

     comment_end_of_fi( ft ) ;


     pcom_after_fi_esac_done = (float)comments_after_fi_esac_done /
end_of_fi_esac_done*100 ;


    /* totc is the total value of required comments; which we called
       construct above                                              */
     funclen(ft) ;
     if ( count_func > 0 )
        func_comments = count_func + missing_comm_bef_func (ft) ;

     totc= 1 + count_func + end_of_fi_esac_done ;

   /*  other_comments =  comment_all(ft) - comment_beginning(ft) -
                       comments_after_fi_esac_done  - func_comments ; */

    pcom = (float) ( comment_beginning(ft) + comments_after_fi_esac_done +
                                       func_comments )/ totc*100 ;


    /* calculate indentation errors and percentige to total lines of code  */


     pind_err = (float) indent(ft)/totline(ft)*100 ;


     rewind(ft) ;
     while((usrvarlen = varcount(ft,name)) != 0 ) {
      sumlen=sumlen + usrvarlen ;
      usrvarcount = usrvarcount + 1 ;

    /* here go through the metrics and chech whether the usrvarlen lies in
       good metrics boundaries.......       */
      for( p=metrics ; p<metrics+NPARAM ; p++ )
     {
       if (strcmp(p->code , "AIDL") == 0 )    {
       if (p->s <= usrvarlen <= p->f )  {
         good_ident++ ;

          }
       } /* outer if ends  */

     } /* for loop ends    */
     pergood_ident = (float)good_ident /usrvarcount*100 ;


     avusrvarlen = (float)sumlen /usrvarcount ;
 }  /* here is the end of while ((usrvarlen....

        in  fscanf.....
        for each file total marks  are collected:  tot_mark, out_of are
        calculated
 by summing inside verbosity loops.These values before next file is
 accessed,  must be initialized to zero.Otherwise the prev. file's
 totals  might mix with the next one and so on.Each single value
 can be initialized to zero inside for (p=metrics .....) loops
 after fprintf and calculations.                                 */

 tot_mark = 0 ;
 out_of = 0 ;



/* print values collected verbosity > 3 */
if(verbosity > 3 )    {

 printf("Total lines               %5d\n",totline(ft) ) ;
 printf("Total characters          %5d\n",totchar(ft) )   ;
 printf("Total blank lines         %5d\n",blkl(ft) ) ;
 printf("Total indented lines      %5d\n",ind(ft) ) ;
 printf("Total spaces              %5d\n",spaces(ft) ) ;
 printf("Total comments            %5d\n",com(ft) ) ;
 printf("Total user created var.   %5d\n",usrvarcount ) ;
 printf("Total len.ofuser creat.var%5d\n",sumlen ) ;
 printf("Total modules length      %.2f\n",tot_funclen );


 /* sumlen and usrvarcount after printing should be assigned to 0
    if one needs to run the this prog for  '<1' number of times
    to examine more files,  obviously 2nd time it gives cumulative
    results.It does not effect other verbosity;  assignment is made
    inside verbosity >3 loop                                   */
/*    sumlen = usrvarcount = 0 ; */


/* calculate the cumulatives for every  measure of each program  */
 sum_totline+=totline(ft) ;
 sum_totchar+=totchar(ft) ;
 sum_blankli+=blkl(ft) ;
 sum_indent+=ind(ft)  ;
 sum_space+=spaces(ft) ;
 sum_com+=com(ft)      ;
 sum_usrvar+=usrvarcount ;
 sumlen = usrvarcount = 0 ;
 sum_funclen+=tot_funclen ;
/*  numberli = totline(fh)  ;*/

/* output results will be written on to afile called "output" ;
   to see just type cat output...         */

 fout= fopen ("output","w" ) ;
 if (fout==NULL ) {
  printf("cant open to write" ) ;
  exit(4) ;
  }

  fprintf(fout,"Sum of lines           %.2f\n",sum_totline) ;
  fprintf(fout,"Sum of chars           %.2f\n",sum_totchar ) ;
  fprintf(fout,"Sum of blank lines     %.2f\n",sum_blankli  ) ;
  fprintf(fout,"Sum of indents         %.2f\n",sum_indent  ) ;
  fprintf(fout,"Sum of spaces          %.2f\n",sum_space   ) ;
  fprintf(fout,"Sum of comments        %.2f\n",sum_com     ) ;
  fprintf(fout,"Sum of usr.creat.var   %.2f\n",sum_usrvar ) ;
```

```
        fprintf(fout,"Sum of func. lengths  %.2f\n",sum_funclen )  ;
      /*  fprintf (fout,"Sum of progs examined %.2d\n",totline(fh) ) ;*/

       fclose(fout) ;

     } /* verbosity ends */



    /* print all results if verbose >1 */
     else  if( verbosity > 1 )    {
      /*  rewind(fh) ;*/
       fout=fopen("output", "a" ) ;
        if (fout == NULL ) {
         printf("cant open fout for marks ") ;
         exit(5) ;
         }

 printf( "----------------------------------------------------\n") ;
 fprintf(fout,"%s\n",testfile ) ;

 fprintf( fout,"----------------------------------------------------\n") ;
 printf("|       Typographic  Analysis                          | \n" ) ;

 fprintf(fout,"|       Typographic  Analysis                          | \n" ) ;
 printf("____________________________________________________\n") ;
 fprintf(fout,"____________________________________________________\n") ;
   printf("|             item     score      mark      out of  |     \n") ;
 fprintf(fout,"|             item     score      mark      out of   |     \n")
;

      for( p=metrics ; p <metrics + NPARAM ;p++ )
   {
       if (strcmp ( p->code, "ACPL" ) == 0 )  {
       printf("| Ave.char per line      %.2f     %.2f        %.2d
|\n",avchar,mark(avchar),p->maxval )    ;

       fprintf(fout,"| Ave.char per line      %.2f     %.2f        %.2d
|\n",avchar,mark(avchar),p->maxval )    ;

         tot_mark = tot_mark + mark(avchar) ;
         out_of = out_of + p->maxval ;

      } else if (strcmp ( p->code, "%BLL" ) == 0 )         {
       printf ("| % blank lines           %.2f     %.2f        %.2d
|\n",pblank,mark(pblank),p->maxval )    ;

         fprintf (fout,"| % blank lines           %.2f     %.2f        %.2d
|\n",pblank,mark(pblank),p->maxval )    ;

         tot_mark = tot_mark + mark(pblank) ;
         out_of = out_of + p->maxval ;

      } else if (strcmp (p->code,"ASPL" ) == 0 )         {
       printf("| Ave.spaces per line    %.2f     %.2f        %2d
|\n",avspace,mark(avspace),p->maxval )   ;

         fprintf(fout,"| Ave.spaces per line    %.2f     %.2f        %2d
|\n",avspace,mark(avspace),p->maxval )   ;

         tot_mark = tot_mark + mark(avspace) ;
```

```
          out_of = out_of + p->maxval ;

       } else if (strcmp (p->code,"%COM" ) == 0 )           {
          printf("| %%comments               %.2f    %.2f       %2d
|\n",pcom,mark(pcom),p->maxval )  ;

          fprintf(fout,"| %%comments               %.2f    %.2f       %2d
|\n",pcom,mark(pcom),p->maxval )  ;

          tot_mark = tot_mark + mark(pcom) ;
          out_of = out_of + p->maxval ;

       } else if (strcmp (p->code,"%IND" ) == 0 )            {
          printf("| %%indentation            %.2f    %.2f      %2d
|\n",pind,mark(pind),p->maxval ) ;

           fprintf(fout,"| %%indentation            %.2f    %.2f      %2d
|\n",pind,mark(pind),p->maxval ) ;

          tot_mark =tot_mark + mark(pind) ;
          out_of = out_of + p->maxval ;

       } else if (strcmp (p->code,"%INDERR" ) == 0 )            {
          printf("| %%indenterror            %.2f    %.2f     %2d
|\n",pind_err,mark(pind_err),p->maxval ) ;

        fprintf(fout,"| %%indenterror            %.2f    %.2f     %2d
|\n",pind_err,mark(pind_err),p->maxval ) ;

          tot_mark =tot_mark + mark(pind_err) ;
          out_of = out_of + p->maxval ;


       } else if (strcmp (p->code,"AIDL" ) == 0 )            {
          printf("| Avr usr creat.var.len  %.2f     %.2f      %.2d
|\n",avusrvarlen,mark(avusrvarlen),p->maxval ) ;

          fprintf(fout,"| Avr usr creat.var.len  %.2f     %.2f      %.2d
|\n",avusrvarlen,mark(avusrvarlen),p->maxval ) ;

          tot_mark = tot_mark + mark(avusrvarlen) ;
          out_of = out_of + p->maxval ;

        } else if (strcmp (p->code,"%NGL" ) == 0 )            {
          printf("| %%gd usr creat.var.len %.2f     %.2f      %.2d
|\n",pergood_ident,mark(pergood_ident),p->maxval ) ;

          fprintf(fout,"| %%gd usr creat.var.len %.2f     %.2f      %.2d
|\n",pergood_ident,mark(pergood_ident),p->maxval ) ;

          tot_mark = tot_mark + mark(pergood_ident) ;
          out_of = out_of + p->maxval ;

       } else if (strcmp (p->code,"AMDL" ) == 0 )            {
          printf("| Avr module len         %.2f     %.2f       %.2d
|\n",avfunclen,mark(avfunclen),p->maxval ) ;

          fprintf(fout,"| Avr module len         %.2f     %.2f       %.2d
|\n",avfunclen,mark(avfunclen),p->maxval ) ;

          tot_mark = tot_mark + mark(avfunclen) ;
        out_of = out_of + p->maxval ;
       /* avfunclen must be initialized to 0 here.If in the next file
         there is no function the prev. value of funclen is carried to
         the next.Initialization avoids this.          */
        avfunclen = 0 ;

     }else if (strcmp (p->code,"SBEPI" ) == 0  && before_pipe > 0 )       {
        printf("| Spaces before pipe     %.2d        %.2f        %.2d
|\n",before_pipe,mark(before_pipe),p->maxval ) ;

        fprintf(fout,"| Spaces before pipe     %.2d        %.2f        %.2d
|\n",before_pipe,mark(before_pipe),p->maxval ) ;

        tot_mark =tot_mark + mark( before_pipe) ;
         out_of+=p->maxval ;

     }else if (strcmp (p->code,"SAFPI" ) == 0  && after_pipe > 0 )      {
         printf("| Spaces after pipe      %.2d        %.2f        %.2d
|\n",after_pipe,mark(after_pipe),p->maxval ) ;

        fprintf(fout,"| Spaces after pipe      %.2d        %.2f        %.2d
|\n",after_pipe,mark(after_pipe),p->maxval ) ;

        tot_mark+=mark(after_pipe ) ;
         out_of+=p->maxval ;

     }else if (strcmp(p->code,"SBESE" ) == 0 && before_semi > 0 )         {
      printf("| Spaces before semicol  %.2d        %.2f        %.2d
|\n",before_semi,mark(before_semi ),p->maxval ) ;

      fprintf(fout,"| Spaces before semicol  %.2d        %.2f        %.2d
|\n",before_semi,mark( before_semi ) ,p->maxval ) ;

      tot_mark+=mark(before_semi) ;
      out_of+=p->maxval ;

     }else if (strcmp(p->code,"SAFSE" ) == 0 && after_semi > 0 )        {
      printf("| Spaces after semicol   %.2d        %.2f        %.2d
|\n",after_semi,mark(after_semi) ,p->maxval ) ;

      fprintf(fout,"| Spaces after semicol   %.2d        %.2f        %.2d
|\n",after_semi,mark(after_semi) ,p->maxval ) ;

      tot_mark+=mark(after_semi) ;
      out_of+=p->maxval ;

     }else if (strcmp(p->code,"SBEIN"  ) == 0  && before_input > 0 )       {
        printf("| Spaces before <        %.2d        %.2f        %.2d
|\n",before_input,mark(before_input),p->maxval ) ;

        fprintf(fout,"| Spaces before <        %.2d        %.2f        %.2d
|\n",before_input,mark(before_input),p->maxval) ;

       tot_mark+=mark(before_input) ;
       out_of+=p->maxval ;

     }else if (strcmp(p->code,"SAFIN" ) == 0  && after_input > 0  )       {
       printf("| Spaces after  <        %.2d        %.2f        %.2d
|\n",after_input,mark(after_input),p->maxval ) ;
```

```
        fprintf(fout,"| Spaces after  <         %.2d         %.2f         %.2d
|\n",after_input,mark(after_input),p->maxval ) ;

        tot_mark+=mark( after_input ) ;
        out_of+=p->maxval ;

      }else if ( strcmp(p->code,"SBEOU" ) == 0  && before_output  > 0 )       {
        printf("| Spaces before >         %.2d         %.2f         %.2d
|\n",before_output,mark(before_output),p->maxval ) ;

        fprintf(fout,"| Spaces before >         %.2d         %.2f         %.2d
|\n",before_output,mark(before_output),p->maxval ) ;

       tot_mark+=mark(before_output) ;
       out_of+=p->maxval ;

     }else if ( strcmp(p->code,"SAFOU" ) == 0  && after_output > 0  )       {
       printf("| Spaces after  >         %.2d         %.2f         %.2d
|\n",after_output,mark(after_output),p->maxval ) ;

       fprintf(fout,"| Spaces after  >         %.2d         %.2f         %.2d
|\n",after_output,mark(after_output),p->maxval ) ;

       tot_mark+=mark(after_output ) ;
       out_of+=p->maxval ;


          /*  print out the total if you like  */
/*          printf("|---------------------------------------------|\n" ) ;
         fprintf(fout,"|-----------------------------------------------------|\n"
) ;
         printf("| total mark is  %.2f  out of %d  %.2f
%%\n",tot_mark,out_of,(float)tot_mark*100/out_of ) ;

        fprintf(fout,"| total mark is  %.2f  out of %d  %.2f
%%\n",tot_mark,out_of,(float)tot_mark*100/out_of ) ;

         printf("------------------------------------------------------|\n" ) ;
         fprintf(fout,"--------------------------------------------------------|\n"
) ;   */


      }  /*  else
         fprintf( stderr, "code %s \n",p->code ) ;  */


  }    /* for loop ends    */

      /*  print out the total if you like  when verbosity >1 after all
          the marks are collected, out_sum cumulatives collected       */

          printf("|---------------------------------------------------------|\n" ) ;
          fprintf(fout,"|----------------------------------------------------------
|\n" ) ;
          printf("| total mark is  %.2f  out of %d  %.2f
%%\n",tot_mark,out_of,(float)tot_mark*100/out_of ) ;

         fprintf(fout,"| total mark is  %.2f  out of %d  %.2f
%%\n",tot_mark,out_of,(float)tot_mark*100/out_of ) ;

          printf("--------------------------------------------------------|\n" ) ;
        fprintf(fout,"-----------------------------------------------------
|\n" ) ;

       fclose(fout) ;

 } /* if verbose > 1  ends   */



  /*  verbosity > 0 SINGLE result; the total */
    else  if(verbosity > 0 )  {

   for(p=metrics ; p<metrics+NPARAM ; p++ )
   {
     if(strcmp(p->code , "ACPL" ) == 0 )  {
      tot_mark = tot_mark + mark (avchar) ;
      out_of = out_of + p->maxval ;

    } else if ( strcmp(p->code , "%BLL" ) == 0 ) {
       tot_mark = tot_mark + mark(pblank) ;
       out_of = out_of + p->maxval ;


    } else if (strcmp(p->code ,"ASPL" ) == 0 )  {
       tot_mark = tot_mark + mark(avspace) ;
       out_of+=p->maxval ;


    } else if (strcmp(p->code ,"%COM" ) == 0 )  {
       tot_mark =tot_mark + mark(pcom) ;
       out_of = out_of +p->maxval ;



    } else if (strcmp(p->code ,"%IND" ) == 0 )  {
       tot_mark =tot_mark + mark(pind) ;
       out_of = out_of +p->maxval ;


    } else if (strcmp(p->code ,"%INDERR" ) == 0 )  {
       tot_mark =tot_mark + mark(pind_err) ;
       out_of = out_of +p->maxval ;


    } else if (strcmp(p->code , "AIDL" ) == 0 ) {
       tot_mark =tot_mark + mark(avusrvarlen ) ;
       out_of = out_of + p->maxval ;


    } else if (strcmp(p->code , "%NGL" ) == 0 ) {
       tot_mark =tot_mark + mark(pergood_ident ) ;
       out_of = out_of + p->maxval ;

    } else if (strcmp(p->code , "AMDL" ) == 0 ) {
       tot_mark = tot_mark + mark(avfunclen ) ;
       out_of = out_of + p->maxval ;

   } else if (strcmp (p->code,"SBEPI"  ) == 0  && before_pipe >0 )          {
       tot_mark =tot_mark + mark( before_pipe) ;
        out_of+=p->maxval ;
    }else if (strcmp (p->code,"SAFPI" ) == 0  && after_pipe > 0)          {
        tot_mark+=mark(after_pipe ) ;
        out_of+=p->maxval ;

   }else if (strcmp (p->code,"SBESE" ) == 0  && before_semi > 0 )         {
        tot_mark+=mark(before_semi ) ;
        out_of+=p->maxval ;


  } else if (strcmp (p->code,"SAFSE" ) == 0  && after_semi > 0)           {
        tot_mark+=mark(after_semi ) ;
        out_of+=p->maxval ;


    }else if (strcmp(p->code,"SBEIN"  ) == 0  && before_input  > 0)        {
      tot_mark+=mark(before_input) ;
      out_of+=p->maxval ;

    }else if (strcmp(p->code,"SAFIN" ) == 0  && after_input > 0  )         {
     tot_mark+=mark( after_input ) ;
     out_of+=p->maxval ;

   }else if ( strcmp(p->code,"SBEOU" ) == 0  && before_output > 0 )       {
    tot_mark+=mark(before_output) ;
    out_of+=p->maxval ;

  }else if ( strcmp(p->code,"SAFOU" ) == 0  && after_output > 0 )         {
    tot_mark+=mark(after_output ) ;
    out_of+=p->maxval ;

  }/*    else
     fprintf( stderr, "code %s \n",p->code ) ;  */

 }  /* for loop ends here */
   printf("Score for Typographic analysis is:: " ) ;
   printf("%5.1f%%\n",(float)tot_mark*100/out_of ) ;
}  /* verbosity > 0 ends */



 /* HANDLE FEEDBACK HERE       */

 if ( feedback)  {
  for ( p=metrics ; p<metrics+NPARAM ; p++ )
  {
    if (strcmp (p->code , "ACPL" ) == 0 )                {
    mark (avchar)  ;

    } else if (strcmp (p->code , "%BLL" ) == 0 )         {
      mark (pblank) ;

    } else if (strcmp (p->code , "ASPL" ) == 0 )         {
      mark (avspace) ;

    } else if (strcmp (p->code , "%COM" ) == 0 )         {
      mark (pcom) ;

    } else if (strcmp (p->code , "%IND" ) == 0 )         {
      mark (pind) ;
      } else if (strcmp (p->code , "%INDERR" ) == 0 )          {
        mark (pind_err) ;

      } else if (strcmp (p->code , "AIDL" ) == 0 )          {
        mark (avusrvarlen) ;

      } else if (strcmp (p->code , "%NGL" ) == 0 )          {
        mark (pergood_ident) ;

      } else if (strcmp (p->code , "AMDL" ) == 0 )          {
        mark (avfunclen) ;

      } else if (strcmp (p->code , "SBEPI" ) == 0 )          {
        mark (before_pipe) ;

      } else if (strcmp (p->code , "SAFPI" ) == 0 )          {
        mark (after_pipe) ;

      } else if (strcmp (p->code , "SBEIN" ) == 0 )          {
        mark (before_input) ;

      } else if (strcmp (p->code , "SAFIN" ) == 0 )          {
        mark (after_input) ;

      } else if (strcmp (p->code , "SBEOU" ) == 0 )          {
        mark (before_output) ;

      } else if (strcmp (p->code , "SAFOU" ) == 0 )          {
        mark (after_output) ;

      }  /* else
      fprintf( stderr, "code %s\n",p->code ) ;  */

    }   /* for loop ends here     */



      if ( max_lost > 0  && feedback >=1 )   {
       printf("Largest Typographic loss of mark was due to :
\"%s\"\n",max_lost_mess) ;

     /*  fprintf(fout,"Largest Typographic loss of mark was due to :
\"%s\"\n",max_lost_mess) ;*/

     }


 }     /* if feedback..... ends   */


     fclose(ft) ;


    } /* end of while fgets(testfile ..... )  */


  /*     fclose(ft) ;*/
      fclose(fh) ;

      return(0) ;
}   /* this is the end of main       */
```

```
   /* Define totchar  */

    int   totchar(FILE *ft )
   {
       int nc ;
       nc = 0 ;
       rewind( ft) ;
       while ( fgetc(ft) !=EOF )
         ++nc ;

         return nc ;
     }


 /* Define space      */

      int spaces(FILE *ft )
   {
         int c, blk ;
           blk = 0 ;
         rewind (ft) ;
         while ((c=fgetc(ft)) !=EOF )
            if(c == ' ') {
               ++blk;

         }

         return blk;
     }

 /* to count #def     */




 /* Define total lines in input */

  int totline(FILE *ft)
  {
    int c, nl ;
    nl = 0 ;
    rewind(ft) ;
    while ((c = fgetc(ft)) != EOF)
      if (c=='\n')
        ++nl ;
    return nl ;
   }


/* Define per. comments */
  int  com(FILE *ft)
  {
  char s[80] ;
  int ncom,c ;
  ncom = 0 ;
  rewind(ft) ;
  while ( ( fgets (s,80,ft ) ) )  {
    if ( s[0]=='#' )  {
      if ( s[1] == '.' )
        break ;
      else
        ++ncom ;
    }
 }   /* end of while  */

  return ncom ;

}



/* Define number of blank lines */
int blkl(FILE *ft)
{
      int blkli ;
      char s[10];
      int c , count ;
      blkli = 0 ;
      count =0;
      rewind(ft) ;
      while( (fgets(s,10,ft)) ) {
        if (s[0]=='\n' )
        blkli++ ;

      }

        return blkli ;
    }



  /* To count indented lines */
int ind(FILE *ft)

{
    int count,n,i;
      char s[80];
      count = 0 ;
     rewind (ft ) ;
    while( fgets(s,80,ft)) {
       if(s[0]==' ' || s[0]=='\t' )
       ++count ;
    }

    return count ;
}



/* also each time a var found , count it and store for total */
     int varcount(FILE *ft,char array[])
  {
     extern int no_good_usrvar;
     int state, i,c,nc,varlen ;
     float avidlen ;
   /* int sumlen ;*/
     /*  initializations start here   */
    no_good_usrvar = 0 ;
    i = 0;   varlen = 0  ;
  /*  sumlen=0 ; */
    state = OUT ;
    avidlen = 0 ;
       /* rewind (fh ) ;*/
    while((c = fgetc(ft)) != EOF ) {
     if ( c== '$' )    {
      while( ( c=fgetc(ft)) != EOF )   {
         if( isdigit(c) || isalpha(c) || c=='-' )
           array[i++] = (char)c ;
         else
           break ;
       } /* 2nd while loop */
      array[i]='\0' ;
      varlen =strlen(array ) ;
       /*  sumlen = sumlen+varlen ;*/
      return varlen ;

      }  /* if stat */

    }  /* 1st while  */

      return 0 ;

}



 /*  To mark the values collected */


    float mark (float fvalue )
  {
      extern int feedback ;
      extern float lost,  max_lost ;
      extern char *max_lost_mess ;

      float score ;
        score = 0 ;
    /*  printf("%d,%d,%d,%f\n", p->f, p->h, p->maxval, fvalue);*/
      if( (float)(p->f) < fvalue && fvalue <= (float) (p->h))  {
         score = (float) p->maxval * ( (float)p->h - fvalue ) /
                                      (float) (p->h - p->f )  ;
    /*   printf("%f\n", fvalue); */
   /*    getchar();    */
      } else if (fvalue >=  p->l  && fvalue < p->s ) {
       score = (float) p->maxval* (fvalue- p->l)  / ( p->s - p->l ) ;
      } else if ( fvalue >=  p->s &&  fvalue <= p->f ) {
       score = p->maxval ;
      }  else {
        score = 0 ;
      }

    /* check whether score is not full and marks are lost;
        p->min is = p->maxval ; if this maxval is not
        achieved  for that factor, then messages are printed;
        saying the value of that factor student got is
        either too long or too little. Also in which field
      student lost max. marks is calculated.
```

```
         Notice the exchange max_lost = lost     */

       if ( feedback >= 2 && fvalue > 0 && p->min > score )    {
        if( fvalue <  p->s    )
        printf("Typographic warning : %s\n", p->lowmessage ) ;
       else
        printf("Typographic warning : %s\n", p->highmessage ) ;

       }  /* end of if (feedback....  */



       lost = p->maxval  > 0 ? p->maxval - score : -score;
       if ( max_lost < lost )  {
         max_lost = lost ;
         if ( fvalue <  p->s ) {
         max_lost_mess = p->lowmessage ;
         }   else {
         max_lost_mess = p->highmessage ;
         }  /* inner else ends   */

       }  /* first if ends   */


       /* to print a single messg. to tell  due to which factor  marks are
        lost the most.                                             */

     /*  if ( max_lost > 0  && feedback >=1 )  {
         printf("Largest Typographic loss of mark was due to :
\"%s\"\n",max_lost_mess) ;
       }  */


       return score ;




     }  /* end of func.    */

int funclen ( FILE *ft )
{
  extern int count_func ;
  extern int j ;
  extern int statem_count ;
  extern int newline_count ;
  int l=0;
  int c, i, state1, state2, flag, opencurly, line_count ;
  int funclevel = 0;
  int fun_statem_count ;
  int length ;
  int semicol ;


  char array[64] ;

  opencurly = 0 ;
  line_count = 0 ;
  count_func = 0 ;
  statem_count = 0 ;
  fun_statem_count = 0 ;
```

```
newline_count = 0 ;
length = 0 ;
semicol = 0 ;
state1=OUTESC ;
state2=OUTDBLEQUOT ;
rewind(ft) ;

while ( ( c=fgetc(ft) ) != EOF )
  {
    switch(c)
    {

    case '\\'  :
      c=fgetc(ft);
      break;

    case '\"'  :
      if (state2==INDBLEQUOT)
        state2=OUTDBLEQUOT;
      else
        state2=INDBLEQUOT ;
      break ;

    case '#':
      c=fgetc(ft);
      while ((c!='\n') && (c!=EOF))
        c=fgetc(ft);
      ungetc(c,ft);
      break;

    case ')':
      if (funclevel!=0)
        break;
      if (state2== INDBLEQUOT)
        break;

      while ( ( c=fgetc(ft) ) ==' ' || c=='\n' )
        {
          ;
        }
      if ( c=='{' )
        {
          funclevel=1;
          ++opencurly ;
          ++count_func;
        }
      else
        ungetc(c, ft);
      break;

    case '{':
      if (funclevel==0)
        break;
      if (state2==INDBLEQUOT)
        break;
      ++opencurly;
      break;

    case '}':
      if (funclevel==0)
          break;
        if (state2==INDBLEQUOT)
          break;
        --opencurly;
        if (opencurly==0)
          funclevel=0;
        break;

        case ';' :
          ++statem_count ;
          if (funclevel > 0 )
            ++fun_statem_count ;

          c=fgetc(ft) ;
          if (c==';')  {
                while ( isspace( c=fgetc(ft) )  )
                    {
                        ;
                  }
                break ;  }

          else
             ungetc(c,ft) ;

          while ( (  c=fgetc(ft) ) ==' ' || c=='\t' | c== '\n'  )
          {   ;
           }

             /* end of while which checks spaces after ';' and cancels
                the newline_count                         */
           break ;


      case '\n':
        l++;
        newline_count++ ;
        if (funclevel>0 )
            line_count++;
 /*             if (semicol == 1 )
                  {   --fun_statem_count ;
                      --statem_count   ;
                        semicol =0 ;   }
          else
              if ( semicol == 1 )
                    {   --statem_count ;
                         semicol == 0 ; }          */
         break ;

      }       /* end of switch      */
    }     /*   end of first while before switch    */

  length = line_count + fun_statem_count ;
  return length ;

}         /* end of funclen    */


/* function which counts spaces before and after " ; |, >, < "
    called as  'count(ft)'    */

 void count(FILE *ft )
 {
```

```
extern  int  before_semi, after_semi, before_pipe, after_pipe, before_input,
             after_input, before_output, after_output,
             count_semi,  count_pipe, count_input, count_output ;


int c, i, state1,state2 ;
int pos ;
 char array[64] ;

 before_semi = after_semi = before_pipe = after_pipe = before_input =
 after_input = before_output = after_output = 0 ;
 count_semi = count_pipe = count_input = count_output = 0 ;


state1 =OUTESC ;
state2 =OUTDBLEQUOT ;

rewind (ft ) ;

 while ( ( c=fgetc(ft) ) != EOF )
   {
     switch(c)
     {

     case '\\'  :
       c=fgetc(ft);
       break;

     case '\"'  :
       if (state2==INDBLEQUOT)
         state2=OUTDBLEQUOT;
       else
         state2=INDBLEQUOT ;
       break ;

     case '#':
       c=fgetc(ft);
       while ((c!='\n') && (c!=EOF))
         c=fgetc(ft);
       ungetc(c,ft);
       break;

       case '|':
         count_pipe++;
         pos = ftell(ft) ;
         before_pipe = -1 ;
         do
       {  before_pipe++ ;
           fseek(ft,-2,1) ;
           c=fgetc(ft) ;

         }  while ( c==' ') ;
           fseek ( ft,pos,0 ) ;
          while ( ( c=fgetc(ft) ==' ') )
            after_pipe++ ;
           break ;

    case '>' :
            count_output++ ;
            pos = ftell(ft) ;
          before_output = -1 ;
          do
        {  before_output++ ;
             fseek(ft,-2,1) ;
             c=fgetc(ft) ;
            }  while ( c==' ') ;

       fseek ( ft,pos,0 ) ; /*puts pos bact to where we were   */
         c=fgetc(ft) ;             /* now we're back, get the char to check
                                  if it's < then we deal with <<   */
       if ( c=='>' )    {
           while ( ( c=fgetc(ft) ) ==' ' )
          after_output++ ;
         break ;    }

       else
         ungetc(c,ft) ;    /*  put the char read back    */
               while ( ( c=fgetc(ft) ==' ') )
           after_output++ ;
           break ;


  case '<' :
    count_input++ ;
      pos = ftell(ft) ;
      before_input = -1 ;
      do
    {  before_input++ ;
         fseek(ft,-2,1) ;
         c=fgetc(ft) ;
       }  while ( c==' ') ;
         fseek ( ft,pos,0 ) ;  /* position bact to where you were */
         c=fgetc(ft) ;         /* now we're back, get the char to check
                                  if it's < then we deal with <<   */
              if ( c=='<' )    {
                  while ( ( c=fgetc(ft) ) ==' ' )
              after_input++ ;
              break ;    }

              else
               ungetc(c,ft) ;
                   while ( ( c=fgetc(ft) ==' ') )
               after_input++ ;
        break ;


case ';'   :
               count_semi++ ;
               pos = ftell(ft) ;
             before_semi = -1 ;
               do
             {
             before_semi++ ;
             fseek(ft,-2,1) ;
             c=fgetc(ft) ;
             } while ( c==' ' ) ;
                fseek(ft,pos,0) ;

                c=fgetc(ft) ;
              if ( c==';' )    {
                  while ( ( c=fgetc(ft) ) ==' ' )
                after_semi++ ;
                break ;    }

                else
                 ungetc(c,ft) ;

                 while ( ( c=fgetc(ft) ) ==' ' )
                 after_semi++ ;
                   /* end of else  */
                 break ;


   }    /* end of switch  */

  }     /* end of while before switch  */


 return ;

 }     /* end of function */


/* function for checking detailed commenting ;
   at the end of fi, esac, done  must be commented,
   before start of each function must be commented,
   begin of code must be commented.                        */

   void comment_end_of_fi ( FILE *ft )
  {
   extern  comments_after_fi_esac_done,  end_of_fi_esac_done  ;

   int  i, c, state2 ;
   char array[64] ;

   i= comments_after_fi_esac_done = end_of_fi_esac_done = 0 ;
   state2 = OUTDBLEQUOT ;

 rewind (ft) ;
 while ( ( c=fgetc(ft) ) != EOF )
 {
    switch(c)
    {

    case '\\'  :
      c=fgetc(ft);
      break;

    case '\"'  :
      if (state2==INDBLEQUOT)
        state2=OUTDBLEQUOT;
      else
        state2=INDBLEQUOT ;
      break ;

       default:
        if(state2 == INDBLEQUOT )
        break ;
        if ( isalpha(c) )
        {  array[i++] = (char)c ;
       /*    printf("value of array=%s\n",array ) ;*/
        }
   else {
     array[i] = '\0';
     i= 0 ;
   /*  printf("value of array after \0=%s\n",array ) ; */

     if( strcmp( array, "fi" ) == 0 )     {
       end_of_fi_esac_done++ ;


 /*      check now to see whether the end of "fi" is
         commented.Do similar with "esac" and "done"
         we are still inside if (strcmp.... "fi" )        */

       while ( isspace ( c=fgetc(ft) ) )
         {
           ;
         }


       if ( c=='#' )
     {  comments_after_fi_esac_done++ ;

         c=fgetc(ft) ;  /* these 4 lines are usef to ignore the
                                words  after '#'; there could be words
                       "fi" after '#'.We must not count those!*/

         while ((c!='\n') && (c!=EOF) )
       c=fgetc(ft) ;
      ungetc(c,ft) ;
        }
     else  /* if c is not='#'  */
         ungetc( c,ft ) ;
         /*  could not find comment, mark down! */

   } else if(strcmp( array, "esac" ) == 0 )    {
       end_of_fi_esac_done++ ;

      while ( isspace ( c=fgetc(ft) ) )
        {
          ;
        }
        c=fgetc(ft) ;
          if ( c=='#' ) {
            comments_after_fi_esac_done++ ;
            while ( ( c!='\n')  && ( c!=EOF) )
             c=fgetc(ft) ;
        }
       else
         ungetc( c,ft ) ;
         /*  could not find comment, mark down! */

 } else if(strcmp( array, "done" ) == 0  )              {
      end_of_fi_esac_done++ ;

      while ( isspace ( c=fgetc(ft) ) )
        {
          ;
        }

       if ( c=='#' )
          comments_after_fi_esac_done++ ;
```

```
           else
              ungetc( c,ft ) ;
              /*  could not find comment, mark down! */
        }    /* end of last if(strcmp....."done" )   */

      }  /*  else part of if (isalpha) ... else
             in default:                             */

    break ;
    }  /*    switch end here                              */

  }    /*  end of while before switch end here         */

  return ;

 }  /* enf of fun. void comments(  ..)    */



  /* function comment_beginning  calculates at least 1 comment before
     begin of code                                   */
    int comment_beginning ( FILE *ft )
   {
     char s[80] ;
     int found_comment_beginning ;

     found_comment_beginning = 0 ;

     rewind (ft ) ;

     fgets(s,80,ft ) ;
     if ( s[0] !='#' && s[1] != '!' ) {
       printf("not BOURNE SHELL script;no marks will be given!!  \n") ;
       return(10) ; }


    /* here we make sure that there is one line starting with a
       comment '#' ; we check s[0]; the first char of the line
       read by fgets(...      )                            */

      while ( ( fgets(s,80,ft) ) )     {

       if( s[0] =='#')  {
         found_comment_beginning++ ;

         break ; }

       else if ( s[0] == ' ' || s[0] == '\n' || s[0] == '\t' )  {
         continue ; }

    }  /* end of while      */

  return found_comment_beginning ;

 }  /* end of int comment_beginning     */


/* function to calculate all the comments; beginning or anywhere
   in the line                                   */
  int comment_all ( FILE *ft )
 {
      int c, state2, total_comments ;

      state2 = OUTDBLEQUOT ;
      total_comments = 0 ;

    rewind (ft) ;
    while ( ( c=fgetc(ft) ) != EOF )
    {
       switch(c)
       {

       case '\\'  :
         c=fgetc(ft);
         break;

       case '\"'  :
         if (state2==INDBLEQUOT)
           state2=OUTDBLEQUOT;
         else
           state2=INDBLEQUOT ;
         break ;

       case '#'   :
         if ( state2 == INDBLEQUOT ) {
           break ;  }
            total_comments++ ;

          break ;

      }  /* end of switch   */

   }   /* end of while just before switch    */

       return (total_comments-1) ;    /* 1 is subtracted from total_comments
                                   because  B. sh. check #! is counted
                                   inside case '#' of switch       */


   }  /* end of function all_comments        */



/* function to check before a func. in B sh. script
   is commented or not, if not commented there will be
   missing marks                                          */
    int  missing_comm_bef_func (FILE *ft )
    {
     extern int count_func ;
     int c, state2, missing_marks ;
     int commentflag, funclevel, currentcomm ;

     count_func = 0 ; missing_marks = 0 ;
     state2 = OUTDBLEQUOT ;
     commentflag =  currentcomm = funclevel = 0 ;

     rewind (ft ) ;

      while ( ( c=fgetc(ft) ) != EOF )
     {
       switch(c)
       {
```

```
case '\\'  :
  c=fgetc(ft);
  break;

case '\"'  :
  if (state2==INDBLEQUOT)
    state2=OUTDBLEQUOT;
  else
    state2=INDBLEQUOT ;
  break ;


case ' ' :
    c=fgetc (ft ) ;
  if ( c == ' ' )  {
    while ( isspace ( c=fgetc( ft ) )  )
      {
      ;
      }
    break ; }  /* if part of c ==' '  */
    else
    ungetc ( c,ft ) ;
  break ;


case '#':
    commentflag = 1 ;
/*  currentcomm = 1 ;*/
  c=fgetc (ft);
  while ((c!='\n') && (c!=EOF))
    c=fgetc(ft);
  ungetc(c,ft);
  break;


case '\n' :
    if ( commentflag == 1 )    {
      currentcomm = 1 ;
    commentflag = 0 ;

    break ;              }
    else {

      currentcomm = 0 ; }
  break ;


case ')':
  if (funclevel!=0)
    break;
  if (state2== INDBLEQUOT)
    break;

  while ( ( c=fgetc(ft) ) ==' ' || c=='\n' )
    {
      ;
    }
  if ( c=='{' )
    {
      funclevel=1;
```

```
            ++count_func;

              if ( currentcomm == 0 )
                missing_marks-- ;
            }
          else
            ungetc(c, ft);
          break;

        }  /* end of switch                         */
      }   /* end of while .. before switch          */

      return missing_marks ;

   }   /* end of missimg_comm_bef_func              */



   /* function to check indentation for if-then-else-fi, while-do-done,
      until-do-done, for-do-done. if-while-until-for keywords are the begin
      of each costruct; so eveytime it is found construct is incremented
      construct++.
      fi-done are the ending keywords, so construct no. is decremented
      everytime we read one.We do this to check if there are inner loops. */

      int indent ( FILE *ft )
  {
   char buffer [1001];
   char c;
   int numconst=0;
   int constoff[50];
   char * token;
   char * tk;
   int flag;

   int MarksDown=0;
   int curpos;

   constoff[0]=1;

   rewind (ft);

   while((fgets(buffer,1000,ft))!=0)
     {

       PreProcessLine(buffer); /*deals with quotes etc. */
       curpos=0;
       while ((c=buffer[curpos++]))
       {
       if (!isspace(c))
         break;
       }
       flag=0;
       token = strtok (buffer, " ;\n");
       if (token == NULL)
       continue;

       if (strcmp(token, "fi")==0)
       {
         flag=1;
         if (curpos != (constoff[numconst]-2))
```

```
    MarksDown++;

  }

  if (strcmp(token, "done")==0)
  {
    flag=1;
    if (curpos != (constoff[numconst]-2))
      MarksDown++;

  }

  if (strcmp(token, "then")==0)
  {
    flag=1;
    if (curpos != (constoff[numconst]-2))
      MarksDown++;
  }

  if (strcmp(token, "do")==0)
  {
    flag=1;
    if (curpos != (constoff[numconst]-2))
      MarksDown++;
  }


  if (strcmp(token, "elif")==0)
  {
    flag=1;
    if (curpos != (constoff[numconst]-2))
      MarksDown++;
  }
        if (flag==0)
  {
    if (curpos != (constoff[numconst]))
      {
        MarksDown++;
      }
  }
  /* this section deals with the possibility of ifs or
   cases coming after other statements */
  tk=token;
  while (token)
  {

    if (strcmp(token, "fi")==0)
      {
        numconst--;
        flag=1;
      }

      if (strcmp(token, "done")==0)
      {
        numconst--;
        flag=1;
      }

    if (strcmp(token, "if")==0)
      {
        /* function to check if there was non white space before if */
```

```
            numconst++;
            constoff[numconst] = (token-buffer+1)+2;
            break;
          }

          if (strcmp(token, "while")==0)
          {
            /* function to check if there was non white space beforewhile */
            numconst++;
            constoff[numconst] = (token-buffer+1)+2;
            break;
          }

         if (strcmp(token, "for")==0)
          {
            /* function to check if there was non white space before if */
            numconst++;
            constoff[numconst] = (token-buffer+1)+2;
            break;
          }

          if (strcmp(token, "until")==0)
          {
            /* function to check if there was non white space before if */
            numconst++;
            constoff[numconst] = (token-buffer+1)+2;
            break;
          }

        token = strtok(NULL, " ;\n");

/*      if (flag==1&&token)
          {
            flag=0;
            MarksDown++;
          }*/
      }
    }

  return(MarksDown);
}

void PreProcessLine(char * c)
{
  static int inquot=0;
  static int indbquot=0;
  for (;(*c)!=0;c++)
    {
      if ((*c)=='\"')
      {
        if (!inquot)
          {
            indbquot = ! indbquot;
            *c = 'X';
          }
      }
      if ((*c)=='\'')
      {
        if (!indbquot)
          {
            inquot = ! inquot;
            *c = 'X';
          }
      }
      if ((*c)=='\\')
      {
        (*c)='X';
        *(c++)='X';
      }
      if ((*c) == '\n')
      continue;
      if (inquot || indbquot)
      {
        (*c)='X';
      }
    }
}




       /* end of function indent (FILE *ft )  */
```

APPENDIX 2

Programs related to testing shell scripts are included in this section. They are written in Bourne Shell.

```
                                #! /bin/sh


# This script is used to check if the stu prog is
# run with all arg and less than all args.
# It is then up to the lecturer to give or deduct
# marks for that
# chech the TEMP.4 file if marks are  to be given



echo "ayse arg-less-act"
echo "arguments are : $ARGS; first=$1, 2nd= $2"


i=1
until [ $i -gt $# ]
do
  echo " in loop arg-less-act"
  echo "$1 $2"
  stu.sh $* > output.$i  2>&1
  shift
  I=`expr $I+1`
  echo "Needs more arguments" > TMP.4
  # for marks actions it is possible to check  TMP.4
  # to do this see if it's not empty test -s
  # if it contains the necassary message
  # eg. needs more arguments
done
echo "in arg-less-act $ARGS"
```

```
#! /bin/sh


#          *******ARG-TEST-ACT*************
#arg-test-act script reads the arguments entered, checks whether
# they are strings or dir or files

# Note that if files/dir then
# make sure that the file called keywords does not exist; it should only be
# available if the args are keywords; not files or dir.
# For files/dir case
# a temp file dir-or-files is created with the first line
# containing  "dir-or-files" string.


# if files or dir  then checks if they are readable
# if not readable,  deducts marks
# if ok, runs stu progs this these arg supplied sufficiently.
# This is done by the arg-less-act  script; which checks to see
# whether the arguments are supplied sufficiently.



FLAG=0

export ARGS
echo "now in ARG-TEST-ACT"
echo "arg-test-act arguments are :ARG-TEST-ACt  $ARGS first=$1 2nd=$2"

# check to see if the arg are dir or files
       for i in $*
       do
         if test -d "$i" -o -f "$i"
         then
           echo "HOPE YOU MADE SURE THAT KEYWORDS FILE HAS NO INFO"

           echo "dir-or-files"  >  dir-or-files
           echo " in arg-test-act dir-or-files exist"

           echo "now testing if the arguments are readable" ;
           if test -r "$i"
           then
             echo  "readable"
         else
             echo "cannot read" > $TMP.3
             N=`expr $N-2`
             echo "N= $N"
         fi #if args are readable


         fi #end of if test "$i" -o -f  "$i"

      FLAG=1
      done  # end of for loop that checks the args;  if they are dir/files




# check to see if the arg are strings of keywords
       for i in $*
        do
```

```
 # here check if the args are strings then do keyword/oracle action
         # here do a grep; edit the strings in afile
       # and check if it's there
         if grep "keywordsexist"  keywords  >  /dev/null  2>@1
         then echo "keywordsexist" >  TMP.KEY
              echo "arg-test-act arg are:$ARGS"

         fi
      FLAG=1
      done

echo "ARG-TEST-ACT JUST BEFORE FLAG"
if test $flag -ne 1
then
  echo "wrong arguments"
  exit 2
fi



if test  -s  dir-or-files  -a ! -s TMP.KEY  # if the files/dir exist  readable
then                                     # and args are NOT strings of keywo.
  echo "in arg-test-act if args are dir /files, then this will appear"

  arg-less-act  $ARGS  # then run stu.prog  & model
                   # with less args (dir/files)

  echo " arg-test-act after arg-less-act arguments are:$ARGS"

fi
```