CS310 Computer Science Project

USER INTENTION PREDICTION BASED ON EXPERIENCE

COPY ONE

Gary Johnson

Project Supervisor: Mike Joy
Second Assessor: Mike Luck
Moderator: Meurig Beynon

Table of Contents

Abstract

This project aims to study the possibility of predicting a user's intention on the basis of previous experience of their patterns of usage. In this report the scope is limited to the use of a Unix® command shell, although ideas for generalization are put forward.

Initially research is carried out through user interviews and questionnaires and a study of actual command shell logs to ascertain patterns and factors which affect the likelihood of a command line being entered. A prototype command shell is developed, based on these findings, with active prediction of the user's intention. Further research is then conducted into how the user interacts with this shell and possible extensions and amendments to the prediction algorithm used are discussed.

In conclusion, the commercial applicability of the work is considered in both command shells and in other areas, including word processing and OCR techniques.

Chapter 1
Introduction

1.0 Motivation

In this project I demonstrate the way computers can be used to predict the intention of a user based on knowledge of the user's previous behaviour when using the system and the context of the current use. The idea of this prediction is the completion of a word or sentence (although as I will discuss these techniques are applicable across many areas of computer use) in such a way that the user has to spend less time communicating with the computer system or reduce errors.

This prediction, based on partial information of the whole phrase being presented, is something that people do all the time, often without realizing it. Take, for example, the following phrase: "I went to the st..." If we look at this sentence without any information about grammar or the context of the sentence it would be very difficult to guess what the full phrase should be. If rules of grammar and syntax are added to aid with the prediction the final word can be narrowed down to be a choice from place nouns starting with "st." Even with this information one cannot give a very good guess as to where the speaker went without some idea of context. In this case if the speaker was relating a story of a train journey the chance that the complete word is "station" would be greater than for "stables."

On any computer system, the greatest bottleneck for the system is the user input - relying on the speed of the user's communication to dictate the total operating time of the commands. Although now rather a cliche, it is important to reduce the time spent waiting for the user input. Technological advances have led to the proportion of time spent idle whilst the user enters commands, compared to processing time between inputs to have grown, since processors have increased in speed whilst user interaction has remained very much the same throughout.

In this project I intend to put forward a method for the reduction of the time the user spends communicating with the computer through the prediction of the user's intention. New methods of human-computer communication have been developed over the last decade bringing with them promises of simpler and speedier communication methods. Rather than removing the need for the ideas I develop here, I suggest that these techniques can be applied to these developments. Methods of communication such as Character and Speech Recognition both suffer from setbacks brought about through the way in which they try to build up a complete input from the individual characters or phonemes making up the whole. I suggest the principles for prediction discussed in this report can be applied to these communication methods to allow prediction of the correct input when a conflict between two or more possible matches exists.

2.0 Aims and Objectives

Within this project I intend to discuss the concepts of prediction within the framework of a Unix command line. It is not the aim of this project to construct a fully-featured shell, rather, I aim to produce a shell with a minimum level of functionality, yet still allowing the verification of the prediction methods developed.

In the previous section I discussed the applicability of these prediction techniques to natural language. Although limiting this study to the use of a command line structure, many of the ideas discussed previously have similarity to a command entered at the command line. Within both a spoken language and using Unix the "user" will only use a limited subset of the complete available vocabulary. It is estimated that a typical English speaker [CO90] will have a

general vocabulary of about ten thousand words with a further five thousand words of technical vocabulary, i.e. used only in specific situations. The same is true in the use of computers. It is highly infeasible to suggest that one user will use all the commands available on that system, and generally true that each user will have their own specific set of commands common to their individual requirements.

Grammar and syntax rules structure the way words are linked together to form sentences in the English language. For example, a sentence is generally built up of the basic building blocks "noun-phrase" followed by "verb-phrase." This idea of syntactic structure can be translated on to the format of a command line. In the case of the `ls` command, the first word must be `ls` followed by a choice of `[-aAcCdfFgiLqrRstu1]` completed with an optional list of `filenames`.

In summary, the objectives of this project are:

- *Develop strategies for prediction on a Unix command line*
- *Create a simple Unix shell demonstrating and testing these findings*
- *Discuss further uses for the concepts developed*

3.0 Development Methodology

Two main themes are developed throughout this report: the research towards discovering the predictability of the user's intention and, following these findings, the development of the software on which to test these findings. The structure of this report follows closely the development methodology used.

Before any software development is started I research the way in which the command line is used, allowing me to develop a prediction strategy for the Unix command line. Beyond this, I discuss the development of the prototyping software used for the testing of this strategy.

Following the design and implementation of the software, I discuss the research carried out to test the prediction strategy used. Using this research, improvements are suggested on the strategy along with a discussion of the success and applications of the study.

Chapter 2
Initial Research

1.0 Aims of Research

So far all reasoning about the predictability of a user's intention on the command line has been based on speculation on my own part. Although discussion with colleagues has given me reason to believe that the patterns of behaviour seen in my own usage are common across many users, before looking at methods of prediction, it is necessary to construct an understanding of factors affecting the likelihood of a command being issued.

1.1 Demonstrating Predictability of User Intention

Before attempting to find factors which may be used to develop a function of command probability, one must first demonstrate that command usage is predictable, i.e. the command issued by the user is dependent on the way s/he has previously used the system. This basic test of predictability can be gained through asking users about the way in which they use Unix (see later).

1.2 Recognize factors affecting likelihood of command issued

After showing that a predictability of intention does exist, the next stage is to look at the factors which affect these predictions. Looking at logs of command line usage combined with

previous studies in this area, a pattern of usage can be generated giving an understanding of those factors which affect the command issued.

Through the development of these prediction factors, a knowledge of the elements of historical data needed on which to base the predictions can be formed. From further study of this data a weighting between each factor can also be revealed.

2.0 Previous Research

Much study has been carried out in the field of Intelligent User Interfaces. Most of this study has looked at ways in which to radically redesign existing interfaces and at methods of design for graphical user interfaces: little study has been done into improving the time spent communicating with the Unix command line. In spite of this, some studies have been carried out which demonstrate the applicability of this study. Here I discuss the relevant salient features from each study.

2.1 [NA92] "Knowledge of Command Usage in a Spreadsheet Program"

This report looks at the spread of commands used within a spreadsheet program in order to reduce the average number of keystrokes required. The main conclusion relevant to this report was the small number of commands used (ten per cent of commands made up approximately ninety per cent of command usage). This suggests that command usage can be modelled and predicted since commands are not used randomly across the whole available command space. In fact, as will be seen in section 3.0, the majority of users use less than three per cent of the total available commands (nearly one thousand five hundred on a simple DCS path).

2.2 [IS89] "User Interface Models by Connectionist Approach"

This study is based on predicting the correct answer from an incorrect submission and the modelling of command sequences and probabilities. It contains various conclusions as to the viability of this project since correct submissions are predicted from both the incorrect entry and knowledge of previous patterns of usage.

Ideas are also presented for the modelling of usage predictability based on overall frequency of use and a transitional probability between commands. Results are given of transitional probabilities between commands showing very high likelihoods of commands being entered dependent on the previous command (or commands).

3.0 Primary Research

In order to give a definite direction to the research, I first developed an hypothesis of factors most likely to affect the likelihood of a command. These factors are based on my own self-perception of Unix usage and from the findings of previous studies outlined above. The hypothesis formed is shown here.

3.1 Hypothesis

The probability and hence predictability of a command being entered on the command line varies as a function of the following variables:

(i) Time spent logged on in current session

(ii) The user's current working directory

(a) Directories are segmented into collections of files created and/or dependent on a subset of the command space

(b) The filename arguments are more likely to be in the current working directory

(iii) Commands entered will be available $PATH, builtins and aliases

(iv) Temporal locality of last use of command

(v) Command use frequency

(a) Overall

(b) In current session

(vi) Argument frequency of last use per command

(a) Overall

(b) In current session

(vii) Machine class being used

I shall now discuss the meaning of each variable presented in the hypothesis and explain the reasons for their inclusion.

3.1.1 Time Spent Logged-On in Current Session

When one initiates a session, I claim that a certain set of commands will always be executed and that certain commands will also be issued periodically throughout the session. For example, when one first logs-on, the first thing that is often done is to check for the receipt of any new e-mail or news. These commands are often also used periodically throughout the session to check for any newly received messages.

3.1.2 The User's Current Working Directory

As stated in the hypothesis this point is based on two different reasonings. Firstly, I believe that directories are created to hold specific groups of related files. Following from this I believe that a subset of all the commands used will be issued on these files due to their grouping. For example, in my directory structure I have the directory *$HOME/CS.First/CS120/a1/* within which I have a set of files from a Pascal programming course. The hypothesis then suggests that the commands *pc*, *jove*, *a.out* will be more likely than other unconnected commands.

Secondly, I suggest that the filename arguments will normally be contained within the current working directory. This is based on the reasonable assumption that the user would move into the directory containing their working files rather than use extended pathnames to call them.

3.1.3 Commands Entered will be in *$PATH*, Builtins and Aliases

Simply stated, this says that a user will have their most commonly used commands set-up for easy access. Any new commands which the user chooses to use will presumably be quickly added to their path or have an alias set for them.

3.1.4 Temporal Locality of Last Use of Command

Here I suggest that a cyclic command usage will exist within a session. For example, when working on a programming assignment the command cycle, *jove file.c - make file - file - jove file.c - ...* , will be seen during the debugging stages of the assignment. It is from this observation, I suggest a command's likelihood will be higher if it has been used in the recent past.

3.1.5 Command Use Frequency

This is probably the most intuitive point of the hypothesis. It states that a commands probability can be predicted from the number of times it has been used in the past. This can be seen on two levels, over all sessions and for the current log-in. This second point being based on the assumption that a particular session will have a single purpose.

3.1.6 Argument Frequency of Use per Command

Each command issued has a particular set of arguments associated with it, as defined by the method of use for that command. For example, one may use the argument *gjohnson@stone* with *finger,* but are unlikely to use this with *imaker*. As with Section 3.1.5, each argument will have a probability based on the number of times it has been used.

3.1.7 Machine Class being Used

Different classes of machines have different commands which can be used on them. For example, it would be unlikely to use *xmail* on a text terminal.

Using the hypothesis as a direction for this study I now intend to describe the two main studies I undertook in order to test the points of this hypothesis. As well as the two methods described here, a great deal of data was gained through discussion with colleagues as to their thoughts on the viability of the points contained in the hypothesis.

3.2 Questionnaire

3.2.1 Aims of the Questionnaire

The hypothesis I describe above is based on subjective assumptions as to the predictability of a command being issued. In order to objectively test the ideas put forward a questionnaire was prepared. As described below, this questionnaire was directed towards checking to see if usage patterns assumed in the hypothesis are in fact those of the “general user.” I described the aims of this research as being to show which factors affect the likelihood of a command and to offer some guidance as to the weighting attributed to each of these points. Through this ques-

tionnaire I hope to be able to test the first of these aims, simultaneously offering a test for the viability of this study.

3.2.2 Questionnaire Development

Before designing the questions for this study, a decision as to the demographics of those questioned had to be formed. I wanted to test a group of people all of which had only been using Unix for a short time. Through this I believed that the group would not have had time to develop personal patterns of usage that may disguise the underlying way in which all people use the Unix command line. The group chosen was primarily first year computer scientists although other scientific disciplines and some second years were represented.

Each question in the questionnaire had to be directed towards testing one or more of the points in the hypothesis. In this section I will discuss the reasons for the choice of questions which fall mainly into three groups. The first of these is a set of questions aimed at checking the demographics of the group being questioned as a check against any inconsistencies in the received data. The other two sets of questions ask directly about usage in the form of tick box answers to example scenarios and ask for the respondent to comment on the methods s/he has when using the command line. I shall now briefly discuss the questions chosen and the reasons for their inclusion.

- *Regularity of Machine Use and Machine Classification*

These questions were aimed at introducing the interviewee to the questionnaire and trying to ascertain some idea as to the extent that text-based terminals and textual interfaces are still being used. The regularity of use gives some indication of the competence with Unix although this is checked in a later question.

- *Use of the E-Mail and News systems*

These questions have two aims. Firstly, it shows me if the user issues the commands for News and Mail every time s/he logs-on; hence demonstrating some periodic repetition of use. Secondly, based on point (vi) in the hypothesis it shows that the arguments given to commands may be shown to be a subset of the total arguments available. For example, of the 619 users available on the DCS site, only a few of them will ever be mailed by one person.

Questions are also included on whether the user checks for the existence of new e-mail/news before entering the reading program. These are used to demonstrate strong transitional probabilities, e.g. from *mailcheck* to *elm*.

- *Use of Directories*

This question aims to test point (ii) in the hypothesis. The question is designed to show if users use directories in the manner I have assumed throughout. The question was left open for the user's own words, thus avoiding the reasons to be suggested by the question.

- *Set of Commands Used*

Two questions are asked on this topic; firstly the user is asked if s/he has a subset of the total number of commands available which s/he uses and, secondly, the user is asked if s/he has a set of commands s/he uses immediately upon logging-in. The first of these questions is aimed at showing that command usage is not purely random across all the commands available to them. By asking the user to name their top five commands one can also ascertain that some commands used are more likely than others.

The second question is aimed at testing hypothesis point (i). If the user can show s/he uses a certain set of commands when logging-in it suggests that certain commands are based on the time spent logged-on.

A copy of the Questionnaire used is shown in Appendix A.

3.2.3 Discussion of Questionnaire Results[1]

The questionnaire was handed out to 103 people and 87 replies were received. Of these replies approximately 70 had been fully answered and about 80% of these had offered to help further in the study if required.

- *Regularity of Machine Use and Machine Classification*

The introductory questions served to check the demographics of the group of people questioned. The information gained from this showed that most people regarded themselves as

1. Full results data is available in Appendix A

Novice users (78%), each using the computers for between 30 minutes and two hours about three times a week. I believe from this that the usage displayed by those questioned will be at a sufficiently early stage such that individual idiosyncracies do not obscure the general patterns of usage.

The answers to the question of machine classification showed that although Computing Services is making a move over to graphical based machines a large number of those surveyed(40%), still primarily use text-based machines.

- *Use of the E-Mail and News Systems*

Although not proving it, the results from this section gave a good test for (vi:a) of the hypothesis. The results showed that 99% of respondents mailed only five people or fewer - highlighting the fact that each user has a personal set of arguments used with each command. These questions also showed a majority of people checking for the existence of new e-mail and news before entering the associated programs, hence showing a high transitional probability from one command linked to another, e.g. *checknews* and *trn*.

- *Use of Directories*

The results show that nearly all respondents had created a directory structure under their home directory. The reasons given for the creation of new directories were mainly for segmentation of files into related groups. Although few respondents actually stated so, I believe it is feasible to assume that the groups of related files are based on the commands used for their creation and modification.

- *Set of Commands Used*

In total some thirty different commands were listed as the top five for all the respondents. On compiling these five could be seen to be much more prominent than others. These were *ls, cd, elm, jove* and *pc*. The reasons for these being the top commands could be discussed, but more importantly, the fact that five commands came out as so much more commonly used suggests that the probability of commands used does vary immensely.

Also, many users gave lists of commands which they used immediately upon logging-on to the system. These being *elm, trn, startx* and similar commands showing, I believe, that some commands are used at specific points during the session.

3.2.4 Conclusions

From these results I believe that the following conclusions can be gained.

- Position in the user's directory structure dictates commands used.
- Some commands are used more regularly than others and these commands are only a subset of all those available.
- Each command uses only a limited subset of all arguments available for the command.

3.3 Command Usage History Files

3.3.1 Aims of History Study

The questionnaire has shown which points are the most salient in the use of the command line. The major drawback of the questionnaire as due to its style of the respondent reacting to the questions posed by me. Using histories allows me to study the respondents actions and form conclusions about behaviour which had not previously been anticipated. The study of histories also allows me to look at the importance of each factor in relation to each other to decide which factors have the greatest effect on the likelihood of a command.

3.3.2 Method of Study

On the questionnaire, respondents were asked if they would be willing to help by supplying logs of command usage. Of the 87 respondents, 50 had stated that they would be willing to offer this help. Upon asking for this help only four people offered this help. It is from this data and my own data logs that this study is based. Although only a very small study, I still believe that patterns of behaviour can be generalized on to most users.

The data was collected from users using a simple addition to their *logout* file. This addition caused the *history* file automatically created by *tcsh* to be mailed across to me. The mail header and line separators were then removed by a simple *perl* script leaving record of the user's session. I then stepped through the file looking for the following items:

- *Overall Command Probability*
- *Transitional Probabilities between Commands*
- *Use of Directories*
- *Linkage between Commands and their Arguments*
- *Re-use of Arguments between Successive Commands*
- *Any Other Factors*

3.3.3 Conclusions

Probably the most striking finding from the study of the History files was the very small number of commands which were used by the respondents. I found that approximately 1500 commands were available in the *path* of most users and of these the average number used was just eighteen. Of these eighteen, I found that just ten made up nearly eighty percent of all usage. Although I believe these results are not entirely typical, they do suggest that guessing just based on the overall probability of a command gives a very good estimate of the commands selected.

This study also gave a high degree of likelihood based on the last command used, i.e. the transitional probability (already shown in Ishikawa's paper). This predictability varied dependent upon the current session. Generally, the behaviour depicted by a user working on an project was much more cyclical than that of someone using the computers for recreational purposes.

Unfortunately, directories were used by only one of the people involved in this study and so little data on this use could be gained. Judging by the results gained from the questionnaire, this is not typical and hence I believe the conclusions discussed earlier although valid could not be checked thorougly.

The questionnaire showed that very few arguments are used for each command. This was highlighted by the history study. Most users had very few arguments which they used with each command and within each individual session this number became even smaller. Another point becoming obvious through this study was the repeated use of arguments across successive commands. This was very common, again, mainly within project work as opposed to for recreational users.

4.0 Research Conclusions

From these studies I have been able to generate an hypothesis of which factors I believed would affect the likelihood of the predictions and then formulate conclusions as to which factors are the most important. From these conclusions, I believe that a specification of a software methodology embracing these ideas for prediction can be formed. These conclusions also suggest the salient items of data which should be extracted and stored to allow the prediction method to function.

For more detail on the use of these conclusions for the formulation of a software methodology see “Use of Prediction Research” on page 23.

Chapter 3
Product Specification

1.0 Requirements Analysis

Before commencing any programming exercise a strong specification of the precise requirements is essential. This requirements specification must have three main characteristics. Firstly, it must be precise with no room for misinterpretation when implementing the software product. Secondly, it must not place unnecessary constraints on the programmer by stating how a problem must be solved, just what it must do, and finally it must be consistent with what is truly required of the system. Here I develop an 'informal' specification of requirements, looking mainly from a user perspective.

1.1 Conceptual Level

On a very high-level this product can be split into two main sections. The first of these is the basic shell which is to act as the base of the product. To allow for maximum usability it is important for the shell to 'look and feel' like a standard Unix command shell. By maintaining this requirement, it ensures that a user is able to use the shell with a minimal amount of extra training. Despite this basic consideration it must be stressed that the shell itself is to act merely as a means of testing the second section of this product, the prediction.

The aim of the prediction is to reduce the amount of time spent communicating with the command line. This is to be achieved through the use of prediction to cut down the number of keypresses required to enter a complete command. This prediction will be based on those factors discussed in the previous chapter and is expanded in Section 2.2 on page 23.

1.2 User-Perspective Specification

1.2.1 User Interface

The shell must have a user interface which is very similar to that of shells currently available and used by Unix users. This is done to lessen the time spent learning a new functionality and to increase user acceptance of the system. In order to ensure this the following concepts must be adhered to:

- *All keypresses must be associated with a screen response*
- *All failed commands should return a descriptive error message*
- *A prompt should be displayed before a command line*
- *The user should be able to edit what s/he has typed*
- *All keys should respond in their usual manner (excluding 'special keys')*

It will be seen later that in order for the prediction mechanism to be utilized, extra keys will be required in order for the acceptance/rejection of commands offered through prediction. In spite of this the user interface presented will allow the user to treat the shell as normal using the keys in their normal way in order to achieve the expected result. The prediction will not interfere with this normal usage, although the user will be expected to observe predictions when they are presented.

Any 'special keys' used should aim to fulfil the requirement of a quick learning time and as such must be keys standard to all keyboards and within easy reach of the user. For full details on the choice of these keys see "Methods of Human-Computer Interaction" on page 20.

1.2.2 Shell Functionality

It has been stressed throughout this document two main points relating to the shell's functionality. Firstly, as described above, the product must 'look and feel' like standard shells and secondly is only required to have a subset of the total functionality offered by these shells.

In order to test the prediction mechanism, and to allow for user testing, the following level of functionality is required of the shell.

- *All commands typed must be able to take arguments*
- *The built-in 'cd' must be handled*
- *Extended features such as pipes, redirection and background processes are not to be handled*
- *History mechanism available as per tcsh et al*
- *Should be able to be linked with prediction and extraction*

The third point of this functionality listing states that pipes *etc*. need not be handled. This is done for simplicity. The main thrust of this report is to describe the way prediction can be added into a shell and thus distraction from this main theme is avoided.

1.2.3 Tailorable Prediction

This product should allow each individual user predictions which are based on their personal usage patterns. To effect this, a file storing the historical data should be saved between sessions. The tailoring of prediction to a user's requirements should be extended beyond this basic level. The user should be able to select the historical data file before executing the shell, hence allowing them to have different predictions dependent upon the user's current project. To allow for further tailoring of the predictions to the user's requirements, a mechanism should be available to directly change the probabilities of the predictions given.

This mechanism should make available the following:

1. Removal of a command or argument from the historical data
2. Increase or decrease of a command's or argument's probability

3. Listings of the commands and arguments in the file as an aid to the above

1.2.4 Hardware

This product is to be developed to work on any Unix platform. As such all coding should adhere to international standards and conventions, hence allowing portability between systems.

Occasionally keyboards may appear differently or may supply different control characters. To allow for such cases any reference to the input of 'special' characters should allow for easy change.

2.0 Design Issues

2.1 Methods of Human-Computer Interaction

The Human-Computer Interface operates on two levels: input and output. It is the way in which the link between these two levels is managed which dictates the success of Human-Computer Interaction. I discuss here in isolation input and output although it must be remembered that every input causes a response to be made in the form of display output.

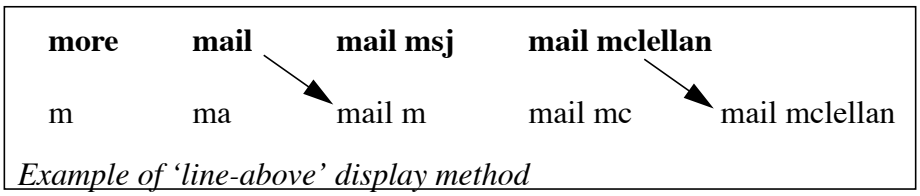
2.1.1 Command Line Entry Display

Without the addition of the prediction mechanism the display of the typed command is a very simple concept. Firstly, the prompt is displayed asking the user for input. Following this each character typed is displayed on the screen, excluding the delete character which removes the character directly before the cursor. With the introduction of prediction this is changed quite considerably.

Every time a prediction is returned this must be displayed for the user to make the choice of whether to accept or reject this guess. In the case of a rejection the guess should be removed (or possibly replaced with another prediction). The display of these guesses should adhere to the following two principles. The shell must be able to work on all hardware platforms and,

secondly, the predictions should be displayed in such a way so as not to cause confusion to the user or force them to have to look far from the line on which they are typing.

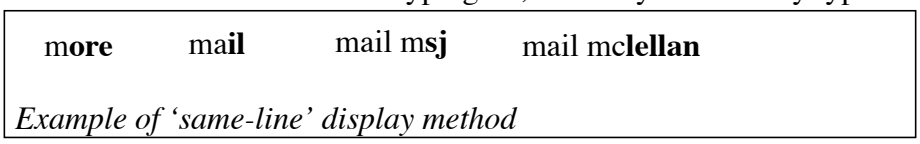
Two methods for the display of the predictions were considered. The first of these, I called the 'line-above' method. This worked by displaying all guesses on the line above that on which the user was typing the command.



Example of 'line-above' display method

The above example demonstrates the way in which predictions are displayed to the user and are then transferred over to the command line when they are accepted. This method has the advantage that it does not confuse the user over what they have typed so far. By corollary, this method may cause confusion over which line they are typing on. The main disadvantages associated with this method include the amount of screen space taken up with each command, leading to the screen filling quicker than usual; perhaps scrolling off information needed by the user and the possible ignorance of the user to the presence of a prediction.

The second method considered was the 'same-line' method. This involves displaying the prediction on the same line as the user is typing on, as if they had already typed it.



Example of 'same-line' display method

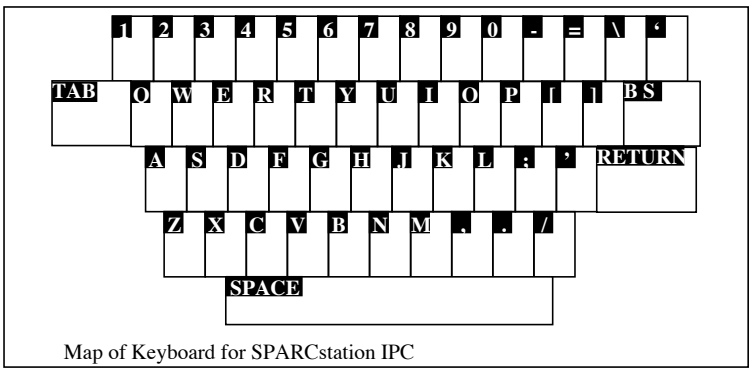
In this example, the predicted section of the command line is shown in **bold** and the part typed by the user (or previously accepted) is shown in normal type. This method has two advantages. Firstly, only the usual number of lines are taken up by the command line and secondly all output is displayed in the place which is most natural to a user of other shells. The main disadvantage with this method is the possible confusion which can be caused by directly altering the contents of the line on which the user is typing.

After much discussion with colleagues it was decided that the 'same-line' method should be adopted. This was mainly due to concerns about having to change where the user looks with

the ‘line-above’ method and also due to the scrolling problems mentioned. It was also decided that the following aspects of the HCI would need amending in order to use this method.

- *When a prediction is present Delete should only remove the guess and not the last typed character. This allows the user to recover from any confusion.*
- *Hitting Return when a prediction is present should automatically accept the prediction and execute the full displayed command.*
- *Once a guess has been rejected (i.e. a further character has been typed), even if the prediction is still valid it should not be displayed the next n[1] times.*

2.1.2 Prediction Selection Keys



Map of Keyboard for SPARCstation IPC

When contemplating the ‘special’ keys to be used for the acceptance and rejection of predictions the physical movement required by the user must be considered. The above keyboard map shows the location of all the keys which are used for the prediction. The only key which is not normally used in a basic shell is Tab. The Tab key is used to accept a prediction and move along to the next argument of the command line. This key was chosen due to its close proximity to the main keys thus making it a simple move for the user.

Other keys which have had their meaning slightly extended or altered are back space and return. The reasons for the changes to these were discussed in the previous section, with back space only removing the prediction when present and the return key causing acceptance and execution of the current prediction. I do not believe these changes are such that they will

1. The value of n is to be calculated during product testing.

increasing the time spent in learning to use this product, rather they will prevent mistakes being made by the user.

2.2 Use of Prediction Research

2.2.1 Method of Prediction

In Section 4.0 on page 15 I discussed the findings of the research carried out in order to form a method for the prediction of the user's intentions. These findings included the highlighting of four main factors leading to a 'high-quality'[1] prediction of a command: the command's overall frequency of use; the transitional probability from one command to the next; the directory in which the command is issued; and the last time the command was used. Similarly for arguments, the factors for prediction were seen to be the command associated with the argument and the last time the argument was used.

The weightings for these factors will be selected during the initial prototyping stages. Despite this the approximate order of importance for the factors can be seen from the data collected through the history studies. As such the weightings used in the initial model are as shown in table below. These weightings are only to be used in order to form a platform from which to

TABLE 1. Initial Weightings for Prediction Factors (Command : Argument)

Factor	Weighting	Factor	Weighting
Overall Use	4	Use with Command	2
Transitional	2	Temporal Locality	1
Directory	3		
Temporal Locality	1		

commence the modelling of the user intention. Once the initial prototype is available these values can begin to be tailored to more optimum figures, along with the threshold figure above which a prediction is returned.

1. High-Quality is a term used to mean regularly correct.

2.2.2 Data Extraction Requirements

In order for the prediction to be carried out a certain amount of data should be extracted from each command line entered to form a bank of historical data of the way in which the command line is used. This data should be held across from one session to the next to form a large model of the user's behaviour. Here I describe the data which must be kept in order for the prediction factors to be used.

- *Commands*

The four factors each require different data to be stored. Firstly, the overall command use simply requires the command name and a weighted frequency of use to be stored. (The weighted frequencies are discussed later.) The transitional probability uses the command names stored for overall use to form a table showing the w.f.'s between each pair of commands. A similar table must be kept, along with a list of directory names, of the w.f.'s of use within each directory. These three sets of data all require storage between sessions.

The fourth factor, recent use of the command, does not require saving across sessions and merely requires a list of commands to be stored showing how long ago they were last used.

Various problems can come about through the use of straight frequencies when adding new commands into the historical data; for this reason the frequencies are weighted, to allow new commands to become a likely prediction more quickly. Using these weights it is ensured that a pair of new commands do not continually swap in and out of the table because they have the lowest frequencies. To avoid problems of one command becoming overwhelmingly likely thus restricting the entry of a new command, frequencies should be normalized between sessions to avoid these irregularities.

- *Arguments*

The command linked frequencies are to be held as described above. The second factor for argument prediction requires a slightly different extraction method. After each successful command is issued the arguments should be stored in a list which can then be referred to in order to find the temporal locality.

3.0 Modularisation and Interface Specification

3.1 The Shell

3.1.1 Input

The input module will supply the shell with individual characters taken from the user. These characters should first of all be classified into one of the token groups described below. No error checking is to be done by the input module. This module is to have a calling function (`get_char`) which is given an address in which to store the inputted character and returns a token.

The tokens to be used are:

- RETN - the Return key
- TABB - the acceptance key (Tab)
- EDIT - an edit character (initially just Back-Space)
- SPAC - the space key
- HIST - one of the history retrieval keys (CTRL-P or CTRL-N)
- QUIT - the exit character (CTRL-D)
- NORM - a standard keyboard character (e.g. A-z 0-9 ...)
- FAIL - a character which can not be classified

3.1.2 Output

This module is to provide all user output of the command line as typed (including and predictions to be shown). It also provides the display of the prompt to request input. Two calling functions should be available for this (`out_line` and `out_prompt`). The line display function will take the list of arguments typed so far.

`out_prompt` should display the prompt in the form “Mc Name Time >”.

3.1.3 Command Execution

This module carries out the execution of all commands entered. It must also be able to deal with the 'built-in' command `cd`. The calling function is to be called `run_command` and should take as input the list of arguments typed by the user. This function should then return a number to represent the success of the execution (0 being normal).

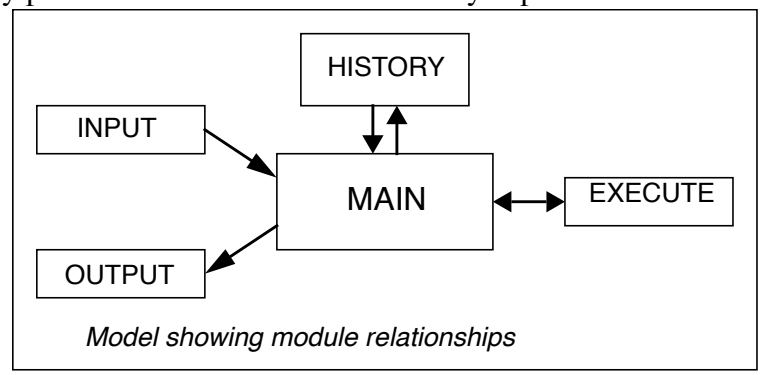
Error handling should be carried out within this function, providing a description of the nature of the error returned.

3.1.4 History Mechanism

The history mechanism has two parts to it. Firstly a function is required to store a command line (`add_hist`) for future retrieval if necessary and the second should return a command line from the history store (`retr_hist`). The returning function should return the command line dependent upon the position in the list which is requested as an argument.

3.1.5 Main Linking Module

The shell part of this product should be able to operate stand-alone although should be designed in such a way as to allow easy connection of the prediction functions. This main routine should initially set-up all global variables and initialize input and output modes. After this, the main function should repeatedly request input to create a command line, display this after each key press and then when the Return key is pressed should execute the command.



Model showing module relationships

The above model demonstrates the independence of each module and the way in which the main module links together all the parts. All calls take place through main hence ensuring that any dependencies can not be broken across different functions when changes are made.

3.2 Prediction

The prediction functions are to be designed to ensure easy linkage with the shell as described above. As will be seen these functions are designed to sit alongside those already described with prediction acting as a secondary input source and data-extraction being a simple call made after a successful execution.

3.2.1 Data extraction and storage

After each command is executed, the return value should be checked to see if the command returned success, and if so a call to `update` should be made. This function should take as arguments the list of arguments entered (including the command) and a reference to the directory in which the command was executed. This call will then update all the historical data stored with the new information.

Functions are also to be made available for saving (`save_store`) and retrieving (`load_store`) the data between sessions. These functions do not take any arguments although the variable `file_name` should be initialized at start-up.

Direct modification of the contents of the data store is available to the user through a series of shell commands, with the common prefix '`!@`'. The '`!@`' shell commands are also contained within this module and should be trapped and called by the main function before issuing the `run_command` function.

A number of functions should also be made available within this module for the retrieval of data by the prediction module. These functions will have the general form of taking command/argument/directory references and returning the weighted frequencies from the tables.

3.2.2 Prediction

Two functions should be externally available within this module: `predict_com` and `predict_arg.` The command prediction function will take as its argument the command typed so far and will return a reference to a command if a prediction can be made, else it should return `-1`. Similarly argument prediction should take as arguments the reference of the command typed (or -1 if an unknown command) with the argument as typed so far and return

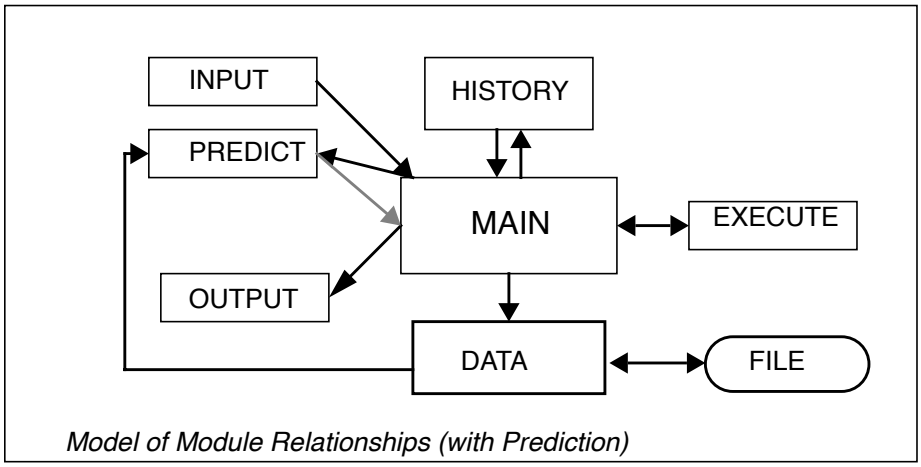
a reference to the argument returned (either in the temporal table or the command-argument lists).

3.2.3 Amendments to Main Linking Module

The main change to be made to append these modules is to make a call to the prediction module after each character is entered. The argument currently being typed should then be temporarily changed to the prediction (if made) and allow the user to revert to the typed part only if required.

Other minor amendments which need to be made are to call the `update` function after each successful command and to load and save the data between sessions.

These changes lead to a slightly revised model of the module relationships shown here.



Model of Module Relationships (with Prediction)

3.3 Global Variables and Constants

I have discussed the fact that the prediction factors and weightings will be changed once initial versions of the shell are made available. In order to allow this to be done easily the following constant definitions should be used.

TABLE 2. List of Constants and their Definitions

Constant Name	*Definition*
WEIGHT_[0-3]	The multipliers for prediction factors of commands
MFP	Threshold probability above which predictions are returned
ATM	Weighting for arguments in temporal table
POW_VAL & POW_MUL	Affect the number of characters to be typed before a guess can be returned after being rejected
NUM_COMS	Number of commands held in data store
NUM_ARGS	Number of arguments held for each command
NUM_DIRS	Number of Directories held in data store
TT_SIZE	Number of commands held in temporal table
AC_MULT	Probability of a newly added command to table

Chapter 4
Module Development

1.0 Issues for Module Development

In the previous chapter the subject of module interfacing was discussed with reference to those values which will be passed when a function is called and which values are to be returned. This interface specification is the most crucial point to be considered when developing the modules. The internal workings of each module function is irrelevant to other parts but the calls and returns must be clearly defined.

As well as the values used to return and call functions it will sometimes be necessary to use variables of wider scope than just within one module or function. In this case the module developer must take great care not to have global updates which overlap causing the wrong values to be stored. The final cross-product concern for module development is that of file storage of data which may be used by more than one module. To avoid this I intend all stored data to be retrieved only by the same module from which it was originally saved.

A further standardization issue is the handling of irrecoverable errors such as running out of memory or file system errors. These errors should be handled consistently so their results are predictable across all modules. In this implementation I use the macro "`PANIC`" detailed in Section 6.0 Support Modules for this purpose.

I now intend to describe the algorithms and codings used for each of the modules in turn. All algorithms are expressed as pseudo-code with the full C code being available in Appendix B.

2.0 User Input and Output

2.1 Input Sub-Module (`get_char.c`)

This sub-module will be called from the main routine returning a token and the actual character selected as described in the specification. The high-level algorithm used to achieve this is shown here.

```
1 Set Input Mode to prevent echo
2 Wait for character to be typed (place into
given location)
3 Select matching token
4 Return token
```

The first part of this function is the setting of input to both stop the characters being echoed and to cause characters to be returned individually without waiting for the end-of-line character to be pressed. So as to avoid unnecessary repeat calls of this each time the function is called, a record of the current input mode is kept which is updated both by this function and a secondary function which resets the input mode to normal before execution. This leads to the following code segment:

```
if not no_echo
      store old input mode for reset to normal
      set input to single chars and prevent echo
      no_echo = TRUE
```

The following three parts of this sub-module are all concerned with the actual production of those values to be returned to the calling procedure. In order to maintain an easy portability of code between machines where control codes and/or keyboard layouts may differ a series of macros are used for matching tokens. These macros can then be changed without the need for major changes to the internal code itself; the macros are currently defined as follows.

```
ISCHAR(c)  isalnum(c) || ispunct(c)
ISRETN(c)  c=='\n'
ISTABB(c)  c=='\t'
```

The prediction acceptance key.

```
ISEDIT(c)  c=='\b'
```

The only edit character available in this prototype will be backspace used to delete the last character typed. Further edit characters can be introduced later by simply adding to this macro and amending the calling procedures.

```
ISSPAC(c)  c==' '
ISQUIT(c)  c=='\4'
```

The CTRL-D character is used for exit from the shell.

These macros are used within a series of `if ... else` statements setting the tokens appropriately. If none of the macros give a match against the character read the token is set to the `FAIL` value for return.

2.2 Output Sub-Module (`out_scr.c`)

The access of any peripherals is always the slowest area of any computer system. In this case the terminal screen must be written to and hence the characters to be written should be made as few as possible to avoid this bottleneck. This is achieved here by the use of a ‘delta string’ which holds a list of characters as described below to edit the line currently displayed. This string is made necessary due to the changes to the display line which can be created due to the predictions being displayed and removed. The algorithm for this uses a copy of the line which was previously displayed to calculate the change needed to be made.

```
1) Check to see if line has been previously
output
2) Calculate length of delta and allocate space
3) Build the delta string
4) Write delta to terminal screen
5) Store copy of current display line
```

Obviously the ‘delta string’ for the first output of a line will be the actual line to be shown as it is been changed from an empty line. This is checked by the first step to avoid this case.

Parts 2 & 3 are best shown through example of the different possibilities of current and previous lines.

- *New line simply longer*

In this case the new line is simply the same as the old line with extra characters appended.

mail mc mail mclellan

In this case the two strings are seen to be the same up to the end of the old string - the delta will consist purely of those characters beyond the final ‘c’ of the last string, “lellan”.

- *New line simply shorter*

In this case the new line is identical to the previous line, although it contains fewer characters.

mail mclellan mail mc

Here the strings are seen to be the same up until the end of the new string and the delta will need to remove the final six characters currently displayed. To do this three passes are needed.

1. Back-Space to the first character to be removed
2. Place spaces over all characters to be deleted
3. Reposition cursor by back-spacing to last display character.

In this example the string would consist of:
“⇐⇐⇐⇐⇐⇐□□□□□□ ⇐⇐⇐⇐⇐⇐”

- *New line different from old line*

In this case, the break-point at which the strings become different is before the end of both strings.

mail mclellan mail cae

In this case a maximum of four passes are required dependent upon the relative lengths of the two strings. If as in this case the old line was longer than the current one, the remaining characters need to be blanked out as in the previous example.

1. Back-Space to the first character to be removed
2. Write from break-point to end of new string
3. (If necessary) place spaces over old characters remaining
4. (If necessary) reposition cursor using back-spaces

The string produced for this example would be:

“⇐⇐⇐⇐⇐⇐⇐⇐⇐cae□□□□□ ⇐⇐⇐⇐⇐”

The algorithm used to achieve this is as follows and can be seen to have been generalized across all cases (this generalization being visible from the previous example).

```
Find breakpoint of two strings
for each char from break to end of old string
      append a backspace character
for each char from break to end of new string
      append the current character
if new is shorter than old
      append |old|-|new| space characters
      append |old|-|new| backspace characters
Append the terminating character (\0)
```

The output sub-module also supplies the function `out_prompt()` which shows the prompt at the beginning of the display line. This is built up very simply as shown here.

```
Get HOST name and append to output string
Get and format TIME and append to output string
Append further prompt characters (i.e. “ >”)
Write string to terminal screen
```

2.3 Command History Sub-Module (`history.c`)

This is included as part of the input/output module because it is used to directly replace the current input string and can then be edited as if it had been typed by the user. It simply keeps a list of the last `HISTSIZE` commands and allows the calling function to retrieve any of these command lines. Two functions are provided: `add_hist` and `retr_hist`. As described in the last chapter these are used to, respectively, add a command line to the history and retrieve one of the command lines.

2.3.1 Add_Hist

Two cases exist for the addition of a command into the history list: previously unused command holders are available and conversely all holders are currently full. The algorithm used is shown here:

```
1) if holders available
2)    allocate storage space
3)    store copy of command line

4) otherwise
5)    remove oldest line stored
6)    move down all reference pointers
7)    store copy of command line as most recent
```

In order to test for holders being available, a variable `ht_capacity` is kept with the number of history elements currently stored. This number always being between zero and `HIST-SIZE`. Whenever a new element is added to the history this value will be incremented, until all holders are filled.

2.3.2 Retr_Hist

This function is called with a reference number requesting a particular command line from the history to replace the current line as typed by the user. This reference number refers to how far back in the history the required command is, i.e. 0 is for the last command entered and `ht_capacity`-1 represents the oldest stored command. The special case of -1 blanks the entire command line to leave the user typing on an empty line. Of course, for this function to work the arguments and the index of the argument currently being edited need to be directly

alterable by this function. In this case these variables are made `extern`ally available to this function.

In this function, the first thing to do is check for the special case -1. If this is not the case and the reference is within the valid range, the old command replaces the current command line. This is shown is pseudo-code below.

```
if reference = -1
      set first argument to ""
      free space for all other arguments

otherwise
      if reference is in range
           for each argument
                allocate space if necessary
                copy history argument to current
```

3.0 Command Execution[1] (`exec_com.c`)

The commands available for this shell can be classified into three main groups: historical data editing commands (!@ commands), built-in commands and simple commands. In this prototype the only built-in command used is `cd` for changing between directories, this needing to be caught in the execution module. Historical data editing commands are embedded within a different module and as such must be checked for by the calling procedure.

The outer level of this function's algorithm is shown here:

```
if arg[0] = "cd"
      change directory to given one or
      change to $HOME if no argument given
otherwise
      execute given command
return error condition
```

I shall now concentrate on the codification of the second, more general, case of this algorithm: executing simple commands. In the previous section on the user input sub-module I discussed

1. Ideas from [RO85] were used in this development.

the changing of input modes to prevent character echoing. This may not be the required mode for the command executed and hence it must be reset before calling the command. This is done quite simply since the ‘normal’ input mode is stored before it is amended allowing this to be re-substituted and the flag discussed above to be changed to signify that the mode is not ready for user input.

```
if no_echo
      reset input mode to ‘normal’
      no_echo = false
```

After this has been changed ready for execution, the command process must be created and executed. This is done using the `fork()` and `execvp()` commands. The pseudo-code and explanation of this part of the function is shown here.

```
1) if not cd
2)    process id = fork()
3)    case process id
4)    -1: PANIC
5)    0:   execvp(args, name)[1]
6)         print error message and exit
7)    default:
8)         wait for child to exit
9)         return exit status
```

In line (2) the child process is created which is to be replaced by the executed command. The id of this process is stored for use in the `case` statement which follows to differentiate the parent and child processes. If the `fork()` fails to create a new process the id will be -1 and (line 4) PANIC is called since this is classed as an irrecoverable error. The other two cases separate the child and the parent processes. The process id will be 0 for the child process (line 5) and this causes the `execvp()` command to be called and any error message is printed if the command does not execute properly. The default case at line (7) causes the parent calling process to halt execution until the child has exited properly and returns the error status to the calling procedure after this.

1. Child process will exit here if successfully executed, otherwise line (6) forces the child process to exit after displaying an error message.

Any non-zero return code is treated as an execution failure. The actual error code returned in this case would be that as defined by the macro `WEXITSTATUS` to be used as necessary by the calling procedure.

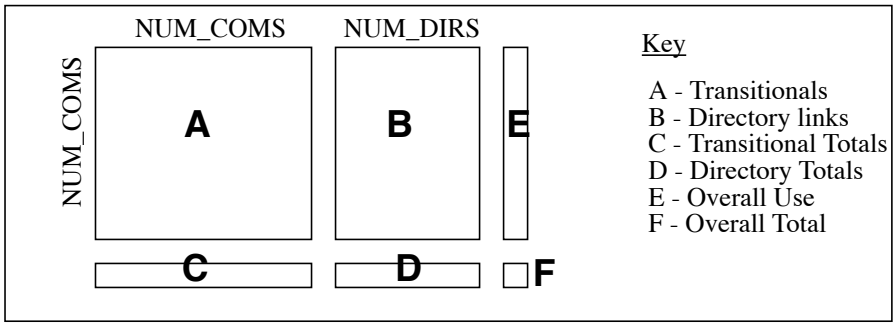
4.0 Extraction and Storage of Historical Data (`data_store.c`)

4.1 Data storage

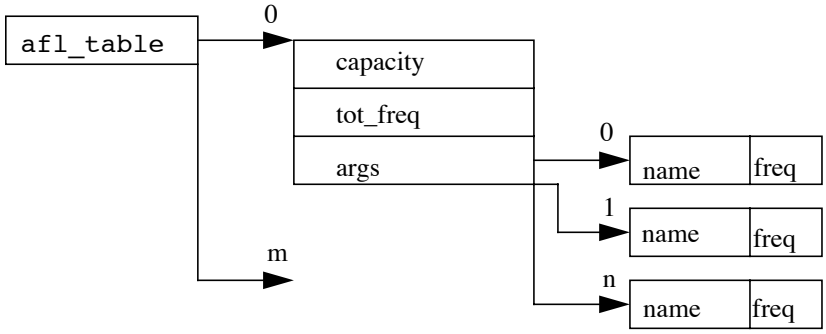
All the functions within this module will require access to the historical data for either amending/storing or retrieving the extracted data. For this reason the first thing I intend to describe here is the way in which the extracted data is stored. The data which is to be extracted is:

- *Commands used with their overall weighted frequency and one-step transitional weighted frequencies*
- *Directories from which commands are executed with command-directory weighted frequencies*
- *Arguments used with each command and their weighted frequencies*

Commands and directories are stored in fixed size arrays of character strings, i.e. `com_list[0⇒(NUM_COMS-1)]` and `dir_list[0⇒(NUM_DIRS-1)]`. The values for the weighted frequencies will then be stored in a table using the references from these lists to cross-refer the different data sets. The data in this table will be stored as shown in the figure below.



The argument data stored is held relative to individual commands only, with the infrequency of cross command usage rendering the table system shown above inefficient for this storage. Rather a separate list of arguments is held for each command (again referenced by their position in `com_list`). Each command has the following argument data held with it, using the structures show below.




```
afl_table {
      capacity
      tot_freq
      args {
           name
           freq
      }
}
```

Each command has a list of arguments attached to it with a maximum capacity of `NUM_ARGS`.

A list of command references is also held for the temporal list of command usage. After each command is executed its reference is stored in the most recent location of this list; its maximum length dictated by `TT_SIZE`. A similar list is maintained of arguments with the actual argument being stored since global references are not available in the same way as for commands.

4.2 Placement values, increments and normalization

Up until now I have simply stated that “weighted frequencies” are held in the data store for each probability group, i.e. transitional etc. Here I discuss the manipulations on these frequencies which aim to both shorten the time necessary for a new command to become likely and to

avoid the scenario of one (or a small group of) command becoming inproportionately more probable than all other commands.

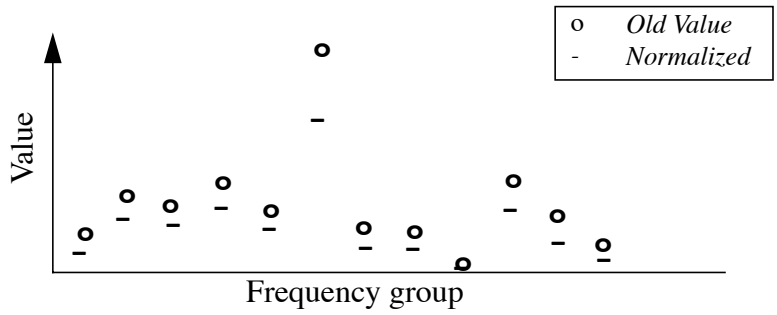
In order to achieve the first of these two objectives, when a command is first entered into the data store its initial 'placement' value is set to be a percentage of the total frequency over the whole frequency grouping. For example, if a new command is to be entered and the total frequency of all commands entered is 100 and `AC_MULT` is set at 10% the new command will be given the initial value of 11, that is:

```
NEW_COM_FREQ = TOT_FREQ * AC_MULT + 1
```

As with prediction weightings, discussed previously, `AC_MULT` will be tailored to achieve an optimum value during initial product testing and research. `AC_MULT` has the constraint that it must be greater than the reciprocal of the maximum number of commands stored; hence preventing swapping in and out of a pair of new commands when initially entering into a full command list.

Once a command has entered the data store all updates of this command's frequencies will cause increments of one. Although appearing arbitrary, I believe that the placement values with the normalization of the frequencies will maintain a well regulated data-set.

The second objective is to avoid excessively high values for a relatively small subset of the complete command list. This is achieved through a series of normalizations which are carried out on the data-set when it is stored at the end of a session. All frequency groups are scanned to check for large deviants from the mean across the group; under which circumstances all values are reduced to 'flatten out' these peaks. Although leaving the relative frequencies



approximately the same, a new command can increase its relative likelihood far quicker than if the table had not been normalized.

Shown here is the algorithm used to achieve this. When values do require normalization, they are reduced proportionately to make the greatest value, lower than `MAX_AVG*mean`.

```
for each frequency group
      calculate mean
      if greatest_value > MAX_AVG*mean
           reduce each value in group
           reset total frequency
```

4.3 Description of Salient Functions in Module

Within this data extraction module, the actual implementation of the data store is hidden from all other calling modules. In order to achieve this, a large number of functions exist which simply return a value from the data store. For example, the function `get_arg_prob()` takes as arguments a command reference and a reference to a related argument and returns the value related to this argument. Due to the relative simplicity of these functions, I do not intend to describe them here beyond a brief description of their use when called from within other functions.

4.3.1 Load and Save functions

All the data held in the data store except for the temporal lists is stored between sessions. The filename for this data can be specified by the user so as to allow more than one data set to be used for different applications. Here I shall describe the `save_store()` function as the reader may assume that `load_store()` is merely the inverse of this.

```
open file
write main command data table
write capacity of    command list
                     directory list
for each element in command list
      write command name
for each element in directory list
      write directory name
for each element in command list
```

```
        write total frequency for argument list
        write capacity of argument list
        for 1 to capacity
             write argument name
             write argument frequency value
close file
```

The data held within this file should not be modified by hand and as such it is stored in such a way as to discourage this practice. When retrieving the data, storage space must be allocated for each string as it is read in.

4.3.2 Data store update calls

After each command has been executed successfully, the data_store must be updated with the command and argument(s) used. This is called through a function `update` which takes as arguments the full command line and the reference for the current directory.

```
if command already in table
1)    update_com
otherwise
2)    add_new_com
3) add_temp_table

for each argument of command
      if already in list
4)         update_arg
      otherwise
5)         add_new_arg
6)    add_arg_tt
```

This function parses the command line into its constituent parts and then calls the relative sub-functions which carry out the actual updates. In line (1), `update_com` is called which increments the values for the overall command use, transitional command use and the directory command use; also updating the relevant totals. If the command is being used for the first time, it must be added to the command list and given its placement values as discussed in Section 4.2 Placement values, increments and normalization, using `add_new_com`.

Two cases exist for this function: the command list is full or the spaces exist. In the former case, the command with the lowest overall usage is swapped out of the list and all its associ-

ated values and linkages are removed allowing the new command to be inserted in its place. This involves resetting all table values, removing the argument list elements and deleting all entries from the temporal table. Following this, in line (3), the command is added to the temporal table.

`update_arg`, `add_new_arg` and `add_arg_tt` are then called for each associated argument in exactly the same way as for the commands. One extra complication does exist for the argument updates when more arguments are given than there are holders in the argument list. In this case an arbitrary decision as to which arguments are ignored has to be made. In this case, the last N arguments are stored, where N is the number of holders available, die to most command lines having the format: **<command> <flags> <filenames>** where the filenames are generally longer.

4.3.3 Data preparation for prediction calls

The values stored exist purely for use by the prediction module when making suggestions as to the most likely command. A number of calls exist which are used to present the data in a manner for use by this module. Before any predictions can start to be made a list of commands (or arguments[1]) must be created which match what the user has typed and/or accepted so far. This is achieved by a simple search of the command list for all those commands whose first n characters match those typed so far. A simple linear search is used for this due to the relatively small number of commands held (typically less than 40 although this can be changed) returning a list of command references.

```
for each command in list
      if first n chars of command = typed so far
      (where n = |typed so far|)
           append command ref to match list
```

Once this list has been provided to the prediction module, the set of values representing the four frequency groups is supplied through a function (`get_com_probs`) which selects the

1. The functions used for commands are discussed here although the ideas can be generalized across both arguments and commands.

relevant data items from the store. Similarly, `get_com_tot_freq` returns the totals for the frequency groups. Both of these are then returned as one structure:

```
struct prob_group {
          overall command frequency
          transitional frequency
          directory frequency
          number of times in temporal table }
```

4.3.4 Direct alteration of data store (!@ commands)

The fourth set of calls made externally to the historical data module are those which allow the user to directly alter the values held within the store. Table 3 shows the functions which are

TABLE 3. Direct Alteration of Data Store Functions

Function	Definition
list_coms	Displays contents of command list
list_dirs	Displays contents of directory list
list_args	Displays argument list for given command
rem_command	Removes given command from command list
rem_argument	Removes given argument for given command
rem_dir	Removes given directory from directory list
change_prob_com	Increase/decrease frequency of command by X%
change_prob_arg	Increase/decrease frequency of argument by X%

made available for these calls. I shall now discuss the salient points from these functions.

- *Rem_command, Rem_argument and Rem_dir*

These functions are all called with the name of the item which is to be deleted from the data store. For example, `rem_argument` is called with the name of the argument to be deleted and the command from which the argument should be deleted. The first part of each function checks to see if the given items are present in their associated lists; if not an error message is displayed and the function returns control. At this stage the position within the list is gained to be used for the removal.

When an item is removed from one of the lists the capacity is decremented by one. Also, all the remaining elements in the list need to kept adjacent, so each element above the one removed need to be shunted one position down the list with all its associated data values.

- *Change_prob_com, Change_prob_arg*

These functions are called with the name of the item which is to be removed along with the percentage value of the change and the polarity of the alteration (i.e. increase/decrease). As with the removal functions, the first thing to be checked is the presence of the items to be changed in the relevant list. Following this the associated values are changed accordingly and the change propagated through to the related totals.

5.0 Intention Prediction (`predictor.c`)

In the previous section, I discussed the functions which are made available for this module to access the required data from the historical data set. These two functions produced a list of possible matches for any string typed so far and return a structure of values for the any referenced command or argument. In this section, I discuss the way in which the data received from the data store is used to achieve a high-quality prediction of the next command.

5.1 Commands

Four values are available from the historical data for the prediction of commands: the overall use, the transitional frequency, the directory use and the number of times present in the temporal table. In the specification chapter, the weightings given to each command were described which are used to make up the likelihood value. Initially, the four weighted values are summated to give a figure for the independent likelihood of the command - i.e. without being in relation to the other possible predictions.

```
((overall use value ÷ ∑overall use)x WEIGHT_0) +
((transition value ÷ ∑transitions)x WEIGHT_1) +
```

$$((\text{directory value} \div \sum \text{dir values}) \times \text{WEIGHT\_2}) + ((\text{temporal entries} \div \text{temp capacity}) \times \text{WEIGHT\_3})$$

This value is then adjusted to take into consideration the possibility that it has already been offered as a guess. This adjustment prevents one guess being offered continually after each character is typed, whilst allowing it to be used as a prediction if it still the most likely after some number of characters has been typed. The number of characters to be typed before the command is given for a second time is defined by:

- *the original likelihood value of the prediction*
- *the two definitions POW_VAL and POW_MUL*

`POW_VAL` and `POW_MUL` are constants which are used for the exponential backoff formula given below which determines how quickly a previous guess can be used as a prediction. The formula takes the value produced above and amends it to take this into consideration.

$$\text{ind\_likelihood} \times (1 - \text{POW\_VAL}^{(-\text{POW\_MUL} * \text{chars})})$$

where `chars` is number of characters typed since the prediction was last given and rejected and `ind_likelihood` is the value produced above.

As each match is checked and this likelihood figure is calculated, a track of the highest value is kept and all the returned likelihoods are summated. After all the likelihood values have been generated, the highest likelihood is inspected relative to the other likelihoods to form a relative value thus:

$$(\text{largest ind\_likelihood}) \div \sum \text{ind\_likelihood}$$

Again at this point the value is adjusted using the same formula as above to take into consideration if the prediction has already been rejected. This is done for a second time, since if only one guess, for example, is available the relative likelihood is by default 1.0 which would cause it to be given as a valid prediction once again. This figure is now compared against `MFP`; if it is greater than this return threshold the command reference is returned, otherwise -1 is returned to signify no guess available.

5.2 Arguments

Two main differences exist between the prediction of commands and arguments. These are that only two values are used for the prediction and that matches may be taken from both the command's argument list and the temporal table separately. The match list in this case is created as follows:

```
Get match_list for argument list
for each element in temporal table
      if line typed so far matches element
      and element not in match_list
            append element to match_list
```

The following formula is then used to calculate the independent likelihood of the matches:

```
get_arg_prob() + (in_arg_tt() x ATM)
```

where `get_arg_prob` returns the value, if available, from the argument list frequencies and `in_arg_tt` returns the number of times the match is found in the temporal table. As with the command prediction these likelihoods are then adjusted to take into consideration previous guesses, inspected relative to one another and then possibly readjusted before being returned to the calling procedure, if the final likelihood figure is greater than MFP.

6.0 Support Modules

6.1 Panic Function (`panic.c`)

This function exists for two reasons. Firstly it supplies debugging information in the event of an error judged to be irrecoverable and secondly it forces the shell to exit cleanly preventing unpredictable results after these errors occurring. The `PANIC` macro itself is defined in a header file for use in all modules as:

```
panic(__FILE__,__LINE__)
```

This causes the panic function to be called with the source file and source line number within this file. Following this, panic displays the line number and filename containing the error

along with the actual error condition which has occurred before aborting the program's execution.

6.2 Usage Messages (`usage.c`)

This set of functions is used purely in order to avoid having large blocks of `printf` calls in the main code for the display of usage, version etc. messages. This also enables the simple editing of these messages when necessary, for example, when the shell's functionality is extended. The functions contained in this module are:

`shell_com_help()`	Displays list of `!@` commands available
`print_usage(command name)`	Command line usage message

Chapter 5
Module Testing

1.0 Issues for Module Testing

As a software product grows in size and complexity so does the number of errors, bugs and coding problems which are present. In order to isolate these problems more easily, each module in the system is tested individually using scaffold code to simulate the exogenous environment for the module. Goodenough and Gerhardt [G075] describe four classes of errors: requirements errors, design errors, specification errors and construction errors. It is hoped that all problems with the requirements and specification errors have been resolved and this chapter is concerned with the discovery and removal of construction errors.

Test scaffolding falls into two main categories: test drivers and stubs. Test drivers are used for the situations as described above, for simulating the calling procedures and the environment of the calling procedure. Conversely, test stubs are used to simulate functions which are called by the module being tested. This can have several uses including cross-module calls, function calls which return indeterminable results etc.

Writers[1] have discussed several strategies for the testing of software each with a general aim of testing a subset of all possible events. The different strategies each attempt to test all different sections of code. For this project, I intend to use the ‘every path’ method which forces execution through each code statement. This is achieved by passing values to the various functions which cause all conditionals to branch through all parts of the condition statements. Within all test drivers and stubs, I will insert statements to output all results, and, if available, partial results, from the functions. This therefore allows errors to tracked down more quickly as the point where the Error has occurred can be estimated.

Some errors are indiscriminate: for example, out of memory or disk failures. These cases can not be easily tested for, which is the reason the `PANIC` macro has been used at all points within the modules where access to external resources, e.g. primary, secondary storage.

2.0 User Input and Output

This module has three main sections: input, output and history. In order to achieve, simpler and more effective testing these are tested separately.

2.1 Input sub-module

Eight character classes have been defined as the tokens in this function. These classes are tested for by a series of `if ... else` statements, hence in order to force all possible paths through the code of this module at least one character from each class should be entered. The test driver used here is as shown below.

```
loop forever
      get_char()
      output TOKEN and character
endloop
```

In testing this module, two characters were used for the testing of some classes. This number was chosen due to the make up of some of the macros, which test two conditions. For exam-

1. I base my ideas here on [GO75] and [ZE78]

ple, `ISCHAR()` tests both the library functions `isalnum()` and `ispunct()`. The test data used and the output from the test driver can be seen in table 4. As well as the formal test data

TABLE 4. List of test data for input sub-module

Input	Driver Output
g	NORM : g
@	NORM : @
<RET>	RETN :
<SPACE>	SPAC :
<TAB>	TABB :
<B/S>	EDIT :
^P	HIST :
^N	HIST :
^D	QUIT :
<F1>	FAIL :
<HOME>	FAIL :

shown here, several other characters were also used and their results noted. From both these sets of test data, this module was seen to behave as defined in the specification. Inherent in these tests was the verification of the functions used to set/reset the input mode. These were also shown to work correctly as both echo and canonical input could be toggled on and off after calling these functions.

2.2 Output sub-module

Three cases were discussed in the development of this function, that I then claimed could be generalized as one function. These cases were where the new line was simply longer than the previous, the new line is simply shorter and where the new line is different from the old string at some point within each string. I also believe that a further special case exists when the output line contains more characters than fit across the display.

Again, a test driver was used for this sub-module of the following form:

```
out_prompt()
out_line( )
pause
out_line( )
pause
out_line( )
out_prompt()
```

The `pause` lines in the driver allow the user to inspect the current line prior to the next modification being displayed. `out_prompt` is called more than once since it both resets the previous line to empty line and resets all counters before the next input line; hence having a direct affect on the `out_line` function.

TABLE 5. List of test data for output sub-module

Out_line() call	Line displayed on screen
* PROMPT *	tamarind Wed 10:51>
mail mc	mail mc
mail mclellan	mail mclellan
mail mc	mail mc
mail cae	mail cae
* PROMPT *	tamarind Wed 10:53>
* Very Long Line *	Completely displayed on two lines
mail mclellan	mail mclellan

All function calls were successful giving results as predicted on both initial and successive calls to the functions.

2.3 Command History Sub-Module

Within this module, three functions are made available for external calls: `retr_hist`, `add_hist` and `list_hist`. When testing this sub-module I shall use all three of these functions together to create a test plan which covers all possible paths through the code.

- *Create history list*

The first section I intend to carry out is the building of a history list using `add_hist` and checking the contents of this list with `list_hist`. In order to achieve a more efficient testing procedure I intend to initially reduce the capacity of the list to three (`HISTSIZE=3`) to reduce the number of iterations required before no spare line holders remain. Each line entered into the history table can have any number of attached arguments which requires many memory (de-)allocation calls. In order to test this the entered command lines will have varying numbers of arguments. A simple test driver is used for testing this.

```
loop until no more input
      input list of arguments
      add_hist(list of arguments)
      list_hist()
endloop
```

The arguments entered with the output from the test driver is shown below. Note the history list is expected to hold the most recent `HISTSIZE` elements.

TABLE 6. List of test data for history sub-module

Argument List	Test Driver Output
ls -al	ls -al **
mail	mail ** ls -al **
rn -g general	rn -g general ** mail ** ls -al **
finger msj cae jet	finger msj cae jet ** rn -g general ** mail **
lash	lash ** finger msj cae jet ** rn -g general **

- *Retrieve elements from list*

Using the history list which has now been created, `retr_hist` is added to the test driver allowing each individual selection to be retrieved at random. In normal use, the elements would be retrieved successively as CTRL-P and CTRL-N are pressed to increment/decrement the request for a history line. The test driver extension is as follows:

```
create dummy command line variable
loop forever
      input number of required element
      retr_hist(number)
      display command line
endloop
```

In order to force all paths through the code I shall make calls to valid elements in the table, the special case -1 (returning empty line) and invalid calls which should have no affect on the displayed command line.

TABLE 7. Test data for retr_hist function

Input	Test Driver Output
0	lash
1	finger msj cae jet
2	rn -g general
1	finger msj cae jet
-3	finger msj cae jet
-7	finger msj cae jet
-1	

3.0 Command Execution

The `run_command` function used to call this module initially checks the command entered to see if the built-in `cd` is to be executed. In this case, a secondary function is called to change the directory. Here I use test data which will force both sets of commands to be executed. A check is also made for null commands, i.e. a command which has no arguments, in which case control should be immediately returned.

Two test scaffolds are used for this module; a driver and a stub. The test stub enables the testing of the error condition in the command execution `switch` when `fork` returns -1. In order to do this, I remove the header file containing the library function `fork` and substitute the following function:

```
int fork () {
      return -1;
}
```

This is used purely for the test of this condition unlike the test driver which is used to call the `run_command` function with any list of arguments. The return code from this function, which would be used by the calling procedure to consider further action, is also displayed by the test driver (0 representing successful execution, non-zero otherwise).

```
loop forever
      input list of arguments
      run_command(list of arguments)
      display return code
endloop
```

The test data used with this function is shown here. It can be seen that some commands seem to return an incorrect exit value, e.g. `mb`. This is a fault on the part of these programs and is not catered for here. For an extended description of this see “Main routine” on page 66. The

TABLE 8. List of input data for Execute module

Command Input	Result	Return Code
ls -al	Directory displayed	0
elm	“elm” mail reader starts	0
mb -x msj craggy	successfully runs with arguments	3
cd Project/SHELL	Directory now Project/SHELL	0
cd	Directory now $HOME	0
cd IllegalDir	cd: No such file or directory	1
wibblewobble	No such file or directory	1
* NULL *	* NO ACTION *	0

test driver was also executed with `fork` replaced as described above and the consequences were:

```
Panic in line 41 of file panic.c
Unexpected library error :
```

4.0 Extraction and Storage of Historical Data

Some functions made available for calling in this module are themselves calling procedures for many other functions. In order to enable simpler testing, the drivers in this module will call these sub-functions and testing of the larger functions will not be done until confidence in these sub-functions is assured. Since the tests used here will depend heavily on data held in the historical store, the functions for the updating of the store will be carried out primarily, with retrievals etc. tested on this data.

A number of different cases exist in the updating and retrieval of data in this module. These are mainly concerned with the capacity of the storage lists, i.e. whether free holders exist. In order to enable simpler testing of these cases the maximum capacity of the tables is lowered so that `NUM_COMS=3`, `NUM_DIRS=3` and `NUM_ARGS=3`.

Throughout the test drivers used for the testing of this module a set of functions were used to display the contents of the data store on the screen. This consisted simply of a series of `printf` statements and did not interfere in anyway with the workings of the functions being tested. Test stub contents and test data with results can be found on Appendix D.

4.1 Data store update calls

After each successful command execution, the function `update` would normally be called with the command and arguments just executed. This would then be broken down into its constituent parts and the commands `update_com/add_new_com` and `update_arg/add_new_arg` are called. For reasons described above I shall test these sub-functions independently.

4.1.1 Adding new values

In the `update` function, when it is called with a command or argument which does not currently reside in the data store it calls `add_new_com` or `add_new_arg` which, if necessary, remove an old value from the table, add the new command/argument and set up the "placement values" for the new entry. Here I test the functions under the two cases, empty holders exist and the table is full. The test driver used for these functions is shown here:

```
loop forever
      input string (command/argument)
      add_new_com[1] (string)
      display com_list and com_table
endloop
```

4.1.2 Updating frequency values

When the `update` function is called with a command or argument which is already existent in the table, `update_arg` and `update_com` are called which increment the frequency values stored in the table. In the case of `update_com`, the overall frequency, the transitional frequency and the directory frequency are all updated. The code for the `update` function prevents this function being called with a non-existent command or argument and as such there is only one special case. This is when `update_com` is called with `last_comm=-1` signifying that no command has been executed before this one and the transitional probability must not be entered into the table. The test driver used is very similar to that for the `add_new_` functions.

```
loop forever
      input string (command/argument)
      update_com[2] (string)
      display com_list and com_table
endloop
```

1. Or add_new_arg as appropriate
2. Or update_arg as appropriate

4.1.3 Temporal tables

On each call to the `update` function the temporal tables are also updated with the command reference or list of arguments. Again, two cases exist for each of the functions `add_temp_table` and `add_arg_tt` these being when spare holders do and do not exist in the table. In order to achieve more efficient testing of this the table capacity is lowered to make `TT_SIZE=3`. The result of this function should be a list of the last three numbers (or character strings) with which it was called. The test driver used repeatedly calls the respective functions and displays the contents of the table.

```
loop forever
      input values to be added
      add_temp_table(values to be added)
      display temporal table values
endloop
```

4.1.4 The `update()` function

Each call from the `update` function has now been verified and the external call function can now be tested as a single unit. This function is simply a calling procedure to the various functions described above using the auxiliary functions `in_com_list` and `in_arg_list` to select which functions to call. Tests on this calling procedure are concerned with the selection of the correct functions to call and so test stubs are used to represent the called functions. For example, this stub is used to replace the function `add_arg_tt`:

```
add_arg_tt(name)
      output "Function add_arg_tt called with"
      output name
```

As well as the use of stubs a simple test driver is again used to initiate the calls to `update`.

```
loop forever
      input command line
```

```
        update ( command line )
endloop
```

The test carried out were based on the values which had been placed in the command and argument lists by the previous tests, in particular, adding new values.

4.2 External Storage and Normalization

The testing for the functions `load_store` and `save_store` uses the data set created by the tests outlined above to check for consistency between the data which is stored and that which is retrieved later. In doing this I originally removed the `normalize_cpt` call from the saving function to ensure the data displayed before calling `save_store` is the same as that which was actually stored. A visual check was then made of the data displayed before calling `save_store` and that which was retrieved later.

Following this `normalize_cpt` was checked independently of its calling function again on the data set used above. Before calls were made to this function, a number of `update` calls were made to cause some frequencies to be outside of the allowed range.

4.3 Data preparation for prediction calls

Nine functions are called from the prediction module in order to retrieve stored values for use in the manufacture of likelihood values. Out of these nine, six simply return a value (or values) from the data store at the position indicated by the references given. The references provided when these functions are called are guaranteed by the calling procedures to be within the valid range of the function. These are tested simply by the use of test drivers requesting and displaying particular data items from the data set created through the tests in section 4.1 above.

Of the remaining three calls, one (`in_arg_list`) simply returns a count of the number of times the string it is called with occurs in the temporal table and the other two return a list of references (`search_com_match`) or strings (`search_arg_match`). Again simple tests are used to display the return values from each function.

4.4 Direct alteration commands

Eight functions are made available within this module to display, remove and amend the values held in the data store. Each of these functions validates the values with which it is called and returns an error condition (0 for success, 1 for failure). To demonstrate this the test drivers used display the returned value after each call and data for each possible error condition is used along with valid data.

5.0 Intention Prediction

The prediction and data extraction modules are very closely linked, with all data used by the prediction module being supplied from the data extraction module using the functions tested in Section 4.3 on page 61. To allow for modular testing each of the function calls from the prediction module must be replaced by test stubs which return well defined results without the possibility of errors propagating from other modules. As well as the use of test stubs to replace these functions, all external variables (e.g. command list capacity) must be explicitly set by the test drivers. The test stubs are of the following format; this example replacing the function `search_com_match` which usually scans `com_list` for matches.

```
search_com_match ( so_far )
      len = strlen ( so_far )
      if (strncmp(so_far,"load",len)==0) {
           match_list[count]=0;
           count++;}
      if (strncmp(so_far,"more",len)==0) {
           match_list[count]=1;
           count++;}
```

5.1 Command prediction

Stubs are used to replace the function calls to `get_com_tot_freqs`, `search_com_match` and `get_com_probs`. There are several conditional statements in `predict_com` which will each cause different paths through the code and, hence, require test driver calls to force each of these alternatives. These conditionals are

- *No matches found, some matches found*
- *Likelihood of current match is greatest so far*
- *Most probable prediction is in old guess list*
- *Likelihood of most probable is (not) greater than MFP threshold*

5.2 Argument prediction

The argument prediction function uses seven functions from the data extraction module all of which are replaced here by stubs.

Although very similar to the command prediction function, `predict_arg` has many extra conditional statements due mainly to the checking of arguments which are present in the temporal table as well as those in the associated command list. These give rise to a larger set of test data shown in Appendix D.

6.0 Support Modules

These functions have no conditional statements in them and as such require only a minimal amount of testing. The `panic` function has already been tested via the command execution module, yet I shall still test it here in isolation from all other functions. A simple test driver will be used here which calls the function which should then output the line number and filename (`panic.test.c`) at which it was called. The results of this are shown here:

```
Panic in line 8 of file panic.test.c
Unexpected library error :
```

The second line of this does not show which error has occurred due to the fact that the `errno` when called was 0.

The remaining two functions `shell_com_help` and `print_usage` are now tested by direct call from a test driver. These functions executed as expected, although `shell_com_help` needed some lines amending for the purposes of a neat display.

7.0 Conclusions

These tests show how the modules work independently of each other. I am confident from the tests used that the functions contained within these modules work as per the specification. However, the interaction between modules, the erroneous setting of global variables and the possibility of interference between memory locations can not be tested at this stage. This initial testing does, nevertheless, does remove the possibility of many coding errors allowing more efficient testing and debugging at the system testing level.

Chapter 6
Product Implementation

1.0 Issues for Module Integration

In this chapter I take the individual modules developed and tested in the previous chapter and integrate them, along with extra bridging code, to form the specified product. After this module integration, the Look-Ahead Shell *(LASh)* can be tested as a complete unit after which, initial versions of *LASh* will be available for beta-testing.

Two main themes are developed here: bridging code and sub-system integrations. It was discussed during module development the need for a main function which would initiate the global variables, call modules and quit the program. The second theme is that of sub-system integration. As will be seen it is highly infeasible to create the complete product immediately as any errors or incompatibilities will not become immediately apparent and their location will be harder; for this reason various executables are created which allow testing of individual module calls and links.

2.0 Bridging Code Requirements

2.1 Main routine

During input after each character is entered the four modules input, output, predict and data retrieval are all called. For this reason it appears sensible for the main routine to carry out all input string conversions as this limits the number of cross-module calls and hence limits the number of links which could result in incompatible data transfers. The main routine is of the following form.

```
parse and validate arguments
initiate globals
load_store()
loop forever
      while not TOKEN = RETN
            input char
            create output string
            output
      prepare to execute
      if !@ command
            call !@ functions
      otherwise
            execute command
            if success
                  update data store
endloop
```

The first part of this procedure prepares all internal data for use by all the modules and takes the filename from the command line arguments if given. This is only done once and after this the loop is entered and iterates until the user chooses to quit the program (i.e. presses CTRL-D). Within this loop a cycle of input-execute-update is executed. Note that the data store will only be updated if a command executes correctly, thus preventing mistakes to enter the command and argument lists. Certain problems have arisen from this due to a very small number of commands returning non-standard values. (`mb` returns the number of arguments and `more` returns zero independent of any illegal arguments).

The prediction functions are called during input dependent on the character being entered which is integrated into the build-up of the command string. The command line is built-up using the tokens and characters returned from `get_char` as follows.

```
switch token
      FAIL:     skip;
      QUIT:     save and return(0);
      EDIT:     if prediction currently shown
                     cancel guess
                otherwise
                     remove last character
      NORM:     if prediction currently shown
                     cancel guess
                append character
                call prediction module
      HIST:     calculate history reference
                substitute for current
      SPAC:     if prediction currently shown
                     cancel guess
      TABB:     increment argument count
endswitch
```

2.2 Calling Procedures

A number of functions are not called directly from the switch statement shown above. Rather they are called via another calling procedure. This is done to avoid having many lines of code within the switch which obscure the meaning and make modification of the code more difficult.

The calling procedures generally do two things. Firstly they prepare the values needed with which to call the external functions and secondly they handle the returned values, if necessary displaying error messages and so on which arise from the return values. Here I used three calling functions for these reasons.

The first of these is the parsing of the !@ commands and calling the required functions with the required values. For example the command:

```
!@-a ls -l 50
```

would result in the function call:

```
change_prob_arg("ls","-l","50",-1).
```

If the function call then returned the failure value, -1, its calling procedure would output the error message:

```
Illegal shell command: type !@he for help
```

The remaining two calling procedures are `call_pred_com` and `call_pred_arg` which are used to call the prediction module, their function being to take the returned command or argument reference and amend the output string as necessary. I.e. if a command reference is returned the current input string is replaced by this, otherwise if no guess is returned (-1) the input string is not changed.

3.0 Sub-system Integrations

Throughout each stage of the integration process I selected groups of modules which could be linked together to form an executable product. This was achieved by amending the switch statement given above to add calls to prediction, execute and so on as modules were integrated. Doing this allowed testing of compatibility between modules at each stage due to the visible results returned from the executable.

This gradual approach gave the following sub-systems:

1. **Input/Output**

These two sub-modules were linked together to form a string display program which parsed the input according to the switch statement above with only NORM, EDIT and SPAC active.

2. **Input/Output/Execute**

This was the first product which could be referred to as a command shell. It was a simple extension of the first which executed the input string when RETN was pressed.

3. **Input/Output/Execute/History**

After each command was executed, the history list was updated. Also the HIST token was activated in the switch statement to allow substitution of old command lines for the current one.

4. **Input/Output/Execute/History/Data Extraction**

After each command was executed, if it was successful (i.e. returned 0) `update` was now called. The addition of this module also meant the addition of directory reference checking before each command and QUIT was extended to save the data store between sessions. The checking for and calling of !@ commands was also incorporated at this point.

5. **Input/Output/Execute/History/Data Extraction/Prediction**

This is the full version of *LASh* which required the incorporation of the two calling procedures described above, the activation of TABB in the switch statement and the addition of calls to prediction within the NORM case of the switch statement.

4.0 Product Testing

The gradual integration of the modules in the way described above allowed each module link to be tested in turn. This was achieved through re-running the module tests, where possible, through the input of commands in the sub-system integrations. For example, the output sub-module was originally tested with a series of different command lines. This test was carried out naturally by typing the command lines through the input module and main routine, hence simultaneously testing the links to the input module.

This sub-system testing showed a number of module interface problems which could be easily located and changed as the modules were appended to the main system. For example, the main routine was originally expecting a string from `get_char` although it supplied a single character.

Through rigorous module testing and now through extended use of the sub-systems and *LASh* I now have a high level of confidence in the quality of this product. I also believe that it meets the requirements as set out in the original specification.

Chapter 7
Further Research

1.0 Requirement for Further Research

The initial research used to develop the set of factors and their weightings within the prediction module was based upon a limited and possibly subjective reasoning. This being due to the problems faced in finding patterns of command line usage which were not initially apparent to myself. Through the release of *LASh* to a subset of users, I have been able to get feedback on problems with the user interface and suggestions on extra prediction factors as well as being able to keep logs from the predictor to analyse weightings and the way in which users actually interact with the product.

Some of the changes recommended through this research meant that minor modifications were made to the product in its development. Although these have been discussed in the development phase I discuss the initial methods used and why changes were made away from these where applicable. Where amendments were not made to *LASh* on issues stemming from this research I discuss the way in which such modifications could be carried out.

2.0 Results and Amendments

2.1 User Discussions

A small number of those completing the original questionnaire offered to trial the initial versions of *LASh*. The six people who used the product were encouraged to send any comments or suggestions on all aspects of the shell to me and I also regularly spoke to them to gauge their level of satisfaction with it. Various new versions were released and at times small changes were tried to gauge opinion on user interface issues.

The main areas which were discussed are:

- *Reasons for rejection of correct predictions*
- *Confusion caused as a result of predictions being displayed*
- *Repeat display of predictions after rejection*

The first of these was of the greatest concern as it suggested at times that the prediction facility was not used and this would question the commercial applicability of such an idea. The main reason that was given was the fact that most commands used would typically only be a few characters long (e.g. ls, more, elm) although arguments where predictions were often accepted were of a greater length. The reasons for this became clear as the users had more experience with the shell. It appeared that those users who had used the shell the most were also the most likely to accept predictions. From this I made the assumption that this problem was caused by the initial learning period where users neglected to look at the predictions given or were not used to using the <TAB> key. However this point does encourage a possible extension to the command and argument prediction functions to offer predictions of greater length ahead of shorter ones which are often rejected

The second and third points here are very closely linked as the third problem originated from the initial fix for the second. On the original version, the fact that a prediction had been rejected once made no difference to the following prediction made, in that a prediction would be just as likely the second time. This caused some confusion to users when a pair of similar commands such as **mail** and **mailtool** were used and one of them was being predicted incorrectly, i.e. the user wished to type **mail** but the prediction was **mailtool** confusion was caused

over what was typed and how to execute **mail**. (This actually involving the user typing mail<space><return> to first remove the prediction.)

In order to fix this problem I made an amendment to the prediction algorithm which prevented the prediction of a command or argument which had previously been rejected. This immediately gave rise to the third area of discussion since it was felt that sometimes a prediction may be missed accidentally. As discussed in the development of the prediction module this was solved through the use of an exponential backoff algorithm forcing the requirement that a certain number of characters must be typed before the guess was redisplayed. A second remedy was also introduced by changing the use of <BACKSPACE> to remove predictions and show just what has been typed so far.

2.2 Session Logs

Within the prediction module a number of debugging lines of code are included which write to file the string typed so far, the matches found, their likelihood values and the returned prediction. An example of the code used for this is shown here.

```
#ifdef DEBUG_FILE
      fprintf (match, likelihoods, last guessed)
#endif
```

The data from this has been used to check the weightings and level of prediction rejection when in use. It was from this data that I originally noted the number of times that correct prediction were rejected by the user. By changing the weightings of the different factors the chosen predictions altered and I recorded the number of incorrect predictions against the number of correct predictions for different weights. The threshold value above which guesses are returned was also varied to see how this affected the quality of the predicted commands and arguments.

The table below shows the final values selected for the weights and the threshold value.

These values were found to give the best predictions for the group of users on which I carried out these tests. Naturally, as was found in the initial research, different users have different patterns of usage. For example, some users make little or no use of directories, at least not in

TABLE 9. Values selected for prediction constants

Name	Definition	Value
Weight_1	Multiplier for Overall probability	5
Weight_2	Multiplier for Transitional	2
Weight_3	Multiplier for Directory relation	4
Weight_4	Multiplier for temporal use	1
MFP	Threshold likelihood value	0.55

the way assumed. I do not believe this leads to a requirement to change these values for different users since this would lead to a random distribution between directory and command, hence causing little effect in the likelihood calculation.

A secondary point noted from the session logs showed a possible new factor in the prediction of arguments. This was the argument position within the command line. For example, the command line:

```
mb -x mclellan stone
```

has a different meaning from:

```
mb -x stone mclellan
```

which is in fact an invalid command. Under the prediction system used at the moment, all other things being equal, `-x`, `stone` and `mclellan` would have an equal chance of being displayed for each position in the command line.

To address this issue I propose a possible extension to the data extraction and prediction modules to store the average position in the command line entered and use this when calculating the likelihood of an argument. I believe this would have a beneficial effect in the area where *LASh* was used most extensively, i.e. long command lines.

Chapter 8
Summary

1.0 Achievement of Aims and Objectives

At the beginning of this report, I set out in detail the aims and objectives of this work. In summary these were:

- *Discussion of prediction concepts within a Unix command shell*
- *Development of a command shell on which to prototype ideas*
- *Extension and generalization of concepts beyond command shell*

The first of these objectives was addressed in chapter two with the demonstration of the principles of this style of prediction and, following from this, the development of a set of factors which affected the likelihood of the command line parts. In this discussion I showed that the further aims were viable since a predictability existed in the commands issued. From this research, the remaining chapters have concentrated on the translation of these principles into a working product on which the ideas developed have been demonstrated. Finally I set out to show how these ideas could be extended and generalized to cover various other areas of input prediction. It is this objective which I address in the latter part of this summary.

2.0 Avenues for Future Investigation

2.1 Scope for Improvement of *LASh*

As initially set out at the beginning of this report, *LASh* is purely a test-bed for the demonstration and validation of the concepts and ideas discussed. This leaves much scope for the improvement and continued development of *LAS*h on two levels. The first and most obvious of these is the addition of some of the advanced features supported by many commercially used Unix command shells.These include input/output redirection, pipes, background processes and aliases. These have not been included here for reasons discussed previously but with the addition of these features I believe that *LASh* could be widely accepted among certain user groups. These groups include users for whom keyboard interaction is slow, although necessary, such as disabled people and users who use a lot of complex command lines containing many arguments and so on. For both these groups I believe *LASh* could make a considerable reduction in the time spent communicating with the computer.

The second level of improvement to this product would be of the prediction algorithm and, primarily, the factors used for this prediction. I discussed in the previous chapter some of the additional factors which may be included, for example argument position and command set matching. These factors originated from an investigation of the way *LASh* was being used and I believe that through the addition of these and other factors would improve the predictions offered and further reduce the time spent communicating with the command shell.

2.2 Other Applications

The third objective discussed above is the generalization of the concepts discussed here. In the section on motivation I suggested various other uses for these techniques in such areas as optical character recognition and speech recognition. After carrying out this study I believe that compelling evidence exists to suggest that these techniques can be used in many areas such as this.

For example, suppose a series of characters had been read by an OCR device and the results had been doubtful with certainty only on the following characters: **the cake re–i–e**. Providing

historical data existed for this set of commands the transitional likelihood for **recipe** would be much higher than that for **revise**. On this basis and on the basis of the set of words used in the text, predictions as to the correct word could be made. This involves a movement away from OCR packages which study each character in turn and in isolation from those around it when translating a text image. Similarly the phonemes of speech are often studied individually by speech recognition systems and these ideas could allow higher success rates by looking at the likelihood of a word being spoken dependent on the context.

Other more widely used packages can also have these prediction concepts applied to them. I discussed at the beginning of this report the similarities of build up between natural language and command lines, both being made up of syntactic parts in set structures. Hence, prediction of the next word in sentences could be made available by word processors, from these techniques.

To summarize, I believe that the ideas which have been demonstrated here, although valid as they stand, have themselves opened up many avenues of investigation for other applications. *LASh* itself, through the reduction of keystrokes, has reduced the time required to enter commands, but this is only one possible use of the concepts presented here.

Appendix A

Contents

Questionnaire

This questionnaire has been made to enable me to gain some background research for my third year project in Computer Science. It aims to show the way in which students at Warwick use the Computer Services Unix System.

Please try and answer all the questions as accurately and in as much detail as possible.

The answering of questions requesting personal data is totally optional. Any information given will be treated confidentially.

Thank you.

Do you have a user code on the Computer Services Unix System?

YES ☐ **NO** ☐

If yes, please specify. _____ (Optional)

Which machines do you normally use?

Graphics Lab (Herb)	☐	**ADM-3e**	☐
Teaching Lab	☐	**DCS Suns**	☐
Remote on PCs	☐	**Newbury Terminals**	☐
Other	☐	Please Specify _____	

How regularly do you use the Computer Services Unix System?

Every Day	☐	**Three Times a Week**	☐
Once a Week	☐	**Less Often**	☐

How long do you spend on the computers in a typical session?

☐	☐	☐	☐	☐
<30 mins	**30-60 mins**	**1-2 hours**	**2-4 hours**	**4 hrs +**

What would you describe as your main uses for the Computer Services System?

Communication ☐ **Playing Games** ☐

Word Processing ☐ **Other Recreational** ☐

Academic Projects ☐ **Other (Please Specify)** ☐

Personal Projects ☐ _____

Do you use the electronic mail system?

YES ☐ **NO** ☐

How regularly do you use electronic mail?

Every Day ☐ **Three Times a Week** ☐

Once a Week ☐ **Less Often** ☐

How many people do you mail on a regular basis?

☐ **One** ☐ **Two** ☐ **Three** ☐ **4 - 5** ☐ **6 - 8** ☐ **9 +**

Do you check for new mail before entering your mail reader?

YES ☐ **NO** ☐

Do you make use of the computer news system?

YES ☐ **NO** ☐

How regularly do you read computer news?

Every Day ☐ **Three Times a Week** ☐

Once a Week ☐ **Less Often** ☐

Do you check for the existence of unread news before entering your news reader?

YES ☐ **NO** ☐

Do you make use of directories for file storage?

YES ☐ **NO** ☐

Under what circumstances would you create a new directory?

How would you describe your competence with Unix?

Novice ☐ **Intermediate** ☐ **Expert** ☐

Can you list five commands you most regularly use?

Is there a set of commands you use immediately upon logging-on?

Is there a set of commands you use immediately before logging-out?

Could you describe a "typical" session?

Would you be willing to answer some more questions if necessary?

YES ☐ **NO** ☐

Would you be willing for me to use logs of your command usage if necessary? (This would involve no disturbance to yourself.)

YES ☐ **NO** ☐

It would be helpful if you could supply the following information (optional)

Name _____

Department _____ Year _____

Questionnaire Results

Question One: Which machines do you most regularly use?

Machine Type	Machine Name	Frequency
Graphics	Graphics Lab	19
	Teaching Lab	32
	DCS Suns	16
	TOTAL	*67*
Text-Based	P.C.s	22
	ADM-3e	10
	Newbury	18
	TOTAL	*50*

Question Two: How often do you use the CSV computer system?

Regularity	Frequency
Every Day	26
Three Times a Week	41
Weekly	14
Less Often	2

Question Three: How long do you spend on the computers in a typical session?

Time	Frequency
Less than 30 mins	7
30 - 60 mins	34
60 - 120 mins	37
2 - 4 hours	4
Longer than 4 hours	0

Question Four: What are your main uses of the CSV systems?

Use	Frequency
Communication	50
Word Processing	13
Academic Projects	71
Personal Projects	17
Games	6
Other Recreational	10

Question Five: How regularly do you use electronic mail?

Regularity	Frequency
Every Day	19
Three Times a Week	26
Weekly	15
Less Often	14

Question Six: How many people do you mail on a regular basis?

Number of People	Frequency
One	17
Two	21
Three	12
Four/Five	21
Six to Eight	0
Nine or more	1

Question Seven: Do you make use of directories?

Yes	62
No	18

Question Eight: How would you rate your competence with Unix?

Competence	Frequency
Novice	62
Intermediate	16
Expert	0

Question Nine: The most popular commands? (Top ten shown)

Command	Frequency listing command
ls	64
cd	41
elm	37
jove	33
pc	24
more	10
cp	9 =
mv	9 =
man	8 =
rn	8 =

Appendix B

Contents

Code Listings

Appendix C

Contents

Manual Page for *LASh*

Appendix D

Contents

Test Data and Results for Data extraction Module

Functions `add_new_com` *and* `add_new_arg`

TABLE 10. Test Data and Results: `add_new_com`

Argument	Return	Command	List	Values
more	0	more		
ls	1	more	ls	
less	2	more	ls	less
load	0	load	ls	less
mail	1	load	mail	less

TABLE 11. Test Data and Results: `add_new_arg`

Com Ref	Argument	Return	Argument	List	Values
0	stone	0	stone		
2	everything.c	0	everything.c		
2	.cshrc	1	everything.c	.cshrc	
1	mclellan	0	mclellan		
1	gjohnson	1	mclellan	gjohnson	
1	hyuvo@csv	2	mclellan	gjohnson	hyuvo@csv
1	psubu@csv	0	psubu@csv	gjohnson	hyuvo@csv
1	msj	1	psubu@csv	msj	hyuvo@csv

Functions `update_com` *and* `update_arg`

TABLE 12. Test Data and Results: `update_com`

Last Comm	Dir Ref	Com Ref	Old Values	Old Total	New Values	New Total
-1	0	2	1 - 1	7	2 - 2	8
2	0	2	2 0 2	8	3 1 3	9
2	0	2	3 1 3	9	4 2 4	10
2	0	1	2 0 2	10	3 1 3	11
1	0	0	2 0 2	11	3 1 3	12
0	1	1	3 0 0	12	4 1 1	13

TABLE 13. Test Data and Results: `update_arg`

Com Ref	Arg Ref	Old Freq	Old Total	New Freq	New Total
0	0	1	1	2	2
2	0	2	5	3	6
2	1	2	6	3	7
2	2	1	7	2	8
2	2	2	8	3	9
2	2	3	9	4	10

Functions `add_temp_table` *and* `add_arg_tt`

TABLE 14. Test Data and Results: `add_temp_table`

Com Ref	Temporal	Table	Values
0	0		
1	1	0	
2	2	1	0
2	2	2	1
0	0	2	2

TABLE 15. Test Data and Results: `add_arg_tt`

Argument	Temporal	Table	Values
everything.c	everything.c		
-a	-a	everything.c	
.cshrc	.cshrc	-a	everything.c
stone	stone	.cshrc	-a
gjohnson	gjohnson	stone	.cshrc

Function: `update`

TABLE 16. Test Data and Results: `update`

Command Line	add_new _com	update _com	add_new _arg	update _arg	add_arg_ tt	add_temp _table
more .cshrc	more		.cshrc		.cshrc	2
ls	ls					0
mail msj jet cae		1		1	msj	
			jet		jet	
			cae		cae	1
cd Project	cd		Project		Project	1
cd Project		1		0	Project	1

Function: `normalize_cpt`

TABLE 17. Test Data and Results: `normalize_cpt`(commands)
Left before normalization and right after normalization

	Com			Dir				Com			Dir			
	0	**1**	**2**	**0**	**1**	**2**	**Tot**	**0**	**1**	**2**	**0**	**1**	**2**	**Tot**
0	1	5	1	0	4	3	7	1	4	1	0	4	3	6
1	0	2	2	1	2	1	6	0	2	2	1	2	1	4
2	0	0	4	1	0	3	4	0	0	2	1	0	2	2
Tot	1	7	7	2	6	7	15	1	6	5	2	6	6	12

Note, table 17 shows the values of the command probability table before and after normalization. Total figures lose synchronization with other parts of table after synchronization - this leaves the relative weightings correct.

TABLE 18. Test data and results: `normalize_cpt`(arguments)
Left before normalization and right after normalization

Com	Arg				Arg			
	0	**1**	**2**	**Tot**	**0**	**1**	**2**	**Tot**
0	1	5	1	7	1	4	1	6
1	3			3	3			3
2	0	2	0	2	0	1[a]	0	1

a. Note: A value cannot be normalized to less than one

Functions for prediction data preparation (Set 1)

TABLE 19. Test data and results for data preparation functions

Function	Arguments		Return
get_com_tot_freq	0		[12, 7, 1, 3]
get_com_probs	1	0	[4, 2, 0, 1]
get_arg_tot_freq	1		3
get_arg_tt	2		msj
get_arg_prob	1	0	3
get_arg_name	1	0	cae

Functions for prediction data preparation (Set 2)

TABLE 20. Test data and results for data preparation functions (set 2)

Command	Arguments	Returns	
in_arg_list	msj	1	
search_com_match	l	0	2
	le	0	
search_arg_match	[1, m]	1	
	[1, ma]		

Test Data and Results for Data extraction Module

Functions for command prediction

Test Stubs Data

```
get_com_tot_freq          [8, 4, 3, 3]

get_com_probs             Command 0 : [2, 3, 1, 0]
                          Command 1 : [5, 1, 1, 2]
                          Command 2 : [1, 0, 1, 1]

search_com_match          Command 0 : load
                          Command 1 : less
                          Command 2 : mail
```

Testing data and results

TABLE 21. Test data and results for predict_com

String	Return	Equivalent
l	-1	*No guess*
lo	0	load
m	2	mail
ma	-1	*No guess*
mai	-1	*No guess*
mail	2	mail

Functions for argument prediction

Test Stubs Data

```
get_arg_tot_freq          9

get_arg_tt                Position 0 : mike
                          Position 1 : cae
                          Position 2 : chris

get_arg_prob              Argument 0 : 4
                          Argument 1 : 2
                          Argument 2 : 3

get_arg_name              Argument 0 : mike
                          Argument 1 : jet
                          Argument 2 : cae

in_arg_list               cae : 2
                          mike : 1

search_arg_match          Argument 0 : mike
                          Argument 1 : jet
                          Argument 2 : cae
```

Testing Data and Results

TABLE 22. Test data and results for predict_arg

String	Com_Ref	Return	Equivalent
c	0	-1	*No guess*
ch	0	5	chris
ca	0	2	cae
m	0	0	mike
mi	0	-1	*No guess*
mik	0	-1	*No guess*
mike	0	0	mike

References

Specific References

[CO90] Cooper BM, “Writing technical reports”, Penguin 1990

[GO75] Goodenough JB et al, “Toward a theory of test-data selection”, IEEE Transactions Soft Eng, June 1975, pp 156-173

[IS89] Ishikawa, “User Interface models by connectionist approach”, Journal of Jap Soc for AI, Vol 4, 1989, pp 398-410

[NA92] Napier et al, “Knowledge of Command Usage in a Software Product”, Data Base, Winter 1992, pp 13-21

[RO85] Rochkind M, “Advanced Unix Programming”, Prentice-Hall 1985, pp 132-146

[ZE78] Zelkowitz MV, “Perspectives of Software Engineering”, Computing Surveys, Vol 10, June 1978, pp 197-216

General Texts

Day RA, "How to write and publish a scientific paper", Cambridge University Press, 1989

Glass G, "Unix for programmers and users", Prentice-Hall, 1993

Gorlen KE et al, "Data abstraction and object-oriented programming", Wiley, 1990

Lamb DA, "Software engineering, planning for change", Prentice-Hall, 1988

Lewie DA, "POSIX programmer's guide", O'Reilly, 1991

Lippman SB, "C++ Primer", Addison-Wesley, 1991

Livadas PE, "File structures, theory and practice", Prentice-Hall, 1990

Macro A, "Software engineering, concepts and management", Prentice-Hall, 1990

Stevens WR, "Advanced programming in the Unix environment", Addison-Wesley, 1992

Tabandah AS, "Interpreting unexpected user activity", IEA/AIE-92, pp 614-621