AIMLESS: A Programming Environment For ML

John Lewis Ophel

A thesis submitted for the degree of
Doctor of Philosophy
of The Australian National University

January 1991
Acknowledgements

I would like to thank Malcolm Newey, my supervisor. The ideas in this thesis have been developed and refined through numerous discussions with him. Thanks also to my supervisory panel, Robin Stanton, Brian Molinaro and Chris Johnson, also, along with Malcolm, have contributed advice, direction and encouragement.

Thanks to everyone else who has made the department such a good environment to work in. All the programmers, past and present, have provided assistance and encouragement. Thanks to my friends, especially James and Colin, with a special mention to Tom for all the good lunches (arguments?) and Colin for sharing an office and putting up with all the constant interruptions and assorted basketball posters for so long.

I must also thank my family - especially my parents, Maureen and Greg, Susan and Greg - for their support (both moral and financial). I hope they (and all my friends) can recover from the shock of my not being a full-time student.

Special thanks to Allisson for her love and companionship, and even more so, during the long and difficult process of examination and re-submission.

This project was supported by a Commonwealth Postgraduate Scholarship and a Sigma Data Postgraduate Award.

The candidate hereby declares that the work in this thesis is that of the candidate alone, except where acknowledgement has been made to the published work of others. It has not been submitted previously, in whole or in part, in respect of any other academic award.

J. L. Ophel
Acknowledgements

I would like to thank Malcolm Newey, my supervisor. The ideas in this thesis have been developed and refined through numerous discussions with him. Thanks also to my supervisory panel, Robin Stanton, Brian Molinari and Chris Johnson, who, along with Malcolm, have contributed advice, direction and encouragement.

Thanks to everyone else who has made the department such a good environment to work in. All the programmers, past and present, have provided assistance whenever necessary. Bev and Jeannie, as well as running the administration smoothly, have reminded everyone that there is more to life than computers. The postgraduate students have provided conversation and ideas. Thanks to Andy, Seppo, Greg, Graham and Charles, with a special mention to Rob for all the discussions (arguments?) and Colin for sharing an office and putting up with AIMLESS, my constant interruptions and assorted basketball posters for so long.

I must also thank my family - my parents, Margaret, Cathy & Greg, Susan and Greg - for their support (both moral and financial!). I hope they (and all my friends) can recover from the shock of my no longer being a full-time student.

Special thanks to Alison for her love and companionship; and even more so, during the long and difficult process of examination and re-submission.

This project was supported by a Commonwealth Postgraduate Scholarship and a Sigma Data Postgraduate Award.
And each one found me aimless,

one more year the worse for wear.

Khe Sahn Cold Chisal
Abstract

The read-eval-print loop (or simply, eval loop) is a convenient and practical base for a programming environment. The eval loop, however, has deficiencies that must be addressed when used in an environment for a modern programming language such as ML. This thesis examines these issues through the design and construction of AIMLESS, a programming environment for ML.

AIMLESS concentrates on the activity of programming in the small - the development of either a single program or one module in a possibly larger system. It overcomes deficiencies of an eval loop programming environment; it provides an editing facility, it maintains a program listing, it provides a re-testing mechanism and it has a trace mechanism.

The central abstraction of the development process used by AIMLESS is the phraselist. The phraselist consists of a script and a buffer. The script is the sequence of valid, evaluated phrases that generated the current binding environment and store. The buffer is a collection of phrases entered but not affecting the binding environment or the store.

The editing operations in AIMLESS are based on the phraselist. Phrases can be inserted, deleted and replaced on the script. Phrases can be combined into larger constructs to promote information hiding - similarly they can be articulated into individual phrases for debugging. By moving phrases onto the buffer and undoing their effects, the edit operations maintain a valid and consistent script at all times. The consistent script can then be used to derive a program listing.

The debugging facilities in AIMLESS are a re-testing mechanism and a trace mechanism. The re-test mechanism views expressions on the script as checkpoints of program behaviour. A test that refers to a function with a changed definition is automatically re-evaluated and a warning given of any changed behaviour. The trace mechanism displays a function’s arguments on call and the result of the function on evaluation. Functions are traced using a source code transformation. The transformation relies on function makestring, the implementation of which
proves non-trivial given ML’s polymorphism and compile-time type system.

The relationship between AIMLESS and the outside world in which it evaluates is discussed. The idea of the logical session, as opposed to the physical session of entering and exiting the ML system, is considered. Mechanisms exist for associating phrases with files, or alternatively, for packaging developed definitions into a program or a module and adding the program or module to file store.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 What Is A Programming Environment?</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Read-Eval-Print Loop</td>
<td>5</td>
</tr>
<tr>
<td>1.3 ML</td>
<td>6</td>
</tr>
<tr>
<td>1.4 An Eval Loop Programming Environment For ML</td>
<td>7</td>
</tr>
<tr>
<td>1.5 AIMLESS</td>
<td>11</td>
</tr>
<tr>
<td>1.6 An Outline Of The Thesis</td>
<td>11</td>
</tr>
<tr>
<td>2 A Review Of Programming Environments</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Some Programming Environments</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Discussion Of These Programming Environments</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Eval Loop Based Environments</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Discussion Of Eval Loop Based Environments</td>
<td>23</td>
</tr>
<tr>
<td>2.5 The Design Decisions For AIMLESS</td>
<td>29</td>
</tr>
<tr>
<td>3 The Editing System</td>
<td>33</td>
</tr>
<tr>
<td>3.1 The Design Of The Editing System</td>
<td>33</td>
</tr>
<tr>
<td>3.2 A More Formal Description of AIMLESS</td>
<td>50</td>
</tr>
<tr>
<td>3.3 A Summary Of AIMLESS</td>
<td>55</td>
</tr>
<tr>
<td>4 Implementing The Editing System</td>
<td>58</td>
</tr>
<tr>
<td>4.1 Evaluation In ANU ML</td>
<td>58</td>
</tr>
<tr>
<td>4.2 The ANU ML Architecture</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Implementing The Phraselist</td>
<td>65</td>
</tr>
<tr>
<td>4.4 Reduced Re-evaluation In AIMLESS</td>
<td>73</td>
</tr>
<tr>
<td>5 Debugging In AIMLESS</td>
<td>84</td>
</tr>
<tr>
<td>5.1 Makestring</td>
<td>85</td>
</tr>
<tr>
<td>5.2 A Re-test Mechanism For AIMLESS</td>
<td>94</td>
</tr>
<tr>
<td>5.3 A Trace Mechanism For AIMLESS</td>
<td>104</td>
</tr>
</tbody>
</table>
Chapter One: Introduction

This thesis examines issues associated with programming environments based on the read-eval-print loop (or simply, eval loop). The eval loop offers significant advantages as the basis of a programming environment, as can be seen in the archetypical example of such an environment, the traditional LISP implementation. LISP, however, has language features that are particularly suitable for an eval loop programming environment. These features are not common to all programming languages, especially modern ones like ML which has constructs that compel an alternative examination of eval loop issues. To do this, a programming environment for ML, AIMLESS, has been designed and built.

Chapter one provides an introduction to programming environments, the eval loop, ML and AIMLESS.

1.1. What Is A Programming Environment?

One of the fundamental issues in computer science is the development of tools to aid in the programming process (Barstow et al 1984). This process can be divided according to scale. Programming in the large is concerned with inter-module programming, programming in the small is concerned with intra-module programming. The term programming environment has been used to refer to tool sets for all such activity (Reiss 1986) but will be used in this thesis to refer only to tool sets for programming in the small.

Using this restricted definition, programming environments are concerned with developing either a single program or a particular module in a possibly larger system. When the term program is used in this thesis, it may refer to the module under development.

A programming environment is evaluated on the functionality offered by its tools, the degree of integration in the tool set and the degree of interaction available to the user.

The functionality of the tools permits a user (at least) to enter, modify, debug
Chapter 1: Introduction

and execute the program under development. To allow top-down development, the programming environment must allow incomplete programs, containing missing or invalid\(^+\) components. "In the top-down refinement of a program, the user often wishes to test selected parts of a definition before all subsidiary definitions are available" (Nikhil 1985). The missing or invalid components cannot be executed, however, and typically cause a run-time error if they are executed.

Integration in the tool set means that it is not just an ad-hoc collection of tools. This affects both the user's view of the tool set and its implementation. The tools should have a consistent approach to program development. For example, debugging is better done at the level of language constructs where program development occurs, instead of at the level of source code line numbers. The tools should also interact as appropriate. For example, an error detected by the compiler may put the user in the editor at the offending construct. In terms of implementation, an integrated tool set can have the tools sharing a common representation of the program so that tools can be implemented more efficiently. For example, the compiler and the debugger can use the same symbol table.

The interaction available in a programming environment consists of the bandwidth of communication between the user and the programming environment, the responsiveness of the environment and the ability for incremental program development.

"[Programming environments] are highly interactive in nature, promoting and exploiting a fairly high bandwidth of communication between the user and the environment" (Barstow et al 1984). This high bandwidth allows the user both to enter and to access required information efficiently. For example, graphical interfaces provide a high bandwidth of communication. The advent of workstations with high resolution displays and processor power available for communication (and not just program execution) have enabled the bandwidth to exist. With future advances, the bandwidth could be abused - the human/machine interface

\(^+\) A valid program contains no context-free or context-sensitive errors. An invalid program contains such an error.
needs to be engineered to avoid possible information overload.

The responsiveness of an environment refers both to the time taken between editing the program and being able to execute the modified program, and to the time taken to execute the program. Interpretive systems reduce the time between editing and execution at the expense of execution time. Compiling systems reduce the execution time at the expense of the translation time (the delay of the so-called edit/compile/execute cycle). Incremental compiling techniques (Schwartz et al 1984; Cardelli 1984; Reiss 1984c) offer the improved execution time of compiling systems, while minimizing the translation time by trying to limit the compiling effort to only those parts of the program affected by the most recent editing.

Incremental program development allows program fragments to be created, executed, tested and debugged. These fragments can in turn be combined and executed, tested and debugged, and so on. This style of development encourages increased interaction as the program is not edited and tested as a whole, but instead the program is developed in a series of edits and tests. “Interactive program development consists alternatingly of tests of procedures (or other program parts) and editing to correct errors discovered during the tests and/or to extend the program” (Sandewall, 1978). Development can therefore occur at the fragment level as well as at the program level.

Programming environments have emerged in response to a changed perception of the way programs are developed. Program development is increasingly recognized as an iterative process. A program is not simply written and evaluated. Instead within the current program, errors must be fixed, definitions extended and algorithms improved. Program development becomes a cycle of editing, testing and debugging. In conventional programming, the process of implementing a specification, the cycle occurs because of errors in implementation. These must be fixed and the program re-tested. In exploratory programming (Sheil 1983; Toyn 1987), the process of specifying a program by experimentation, the cycle occurs because the program changes as it is understood. Each iteration builds on the existing program structure either vertically, by the deepening of existing facilities,
or horizontally, by the addition of more facilities (Sandewall 1978).

The techniques used in exploratory programming are applicable to conventional programming. Instead of implementing the whole program and then trying to find and correct errors, it is often more practical to develop a program skeleton and build on fragments. This is especially true in light of increases in processing power that have changed the onus in program development from the effective use of machine resources to the effective use of human resources.

Programming environments support iterative development. Incomplete programs allow the program to be developed top-down, each iteration building on the program skeleton. The user can focus on the main structure and add parts as required, rather than dealing with a monolithic program. Iterative development encourages repeated cycles of moving between tools such as the editor, tester and debugger when building the program. Integrated tool sets reduce the need for mental context switches (Delisle et al 1984) when moving between tools.

The first aspect of interaction in programming environments is the increased bandwidth. Iterative development naturally creates traffic that must be handled and can be used to provide more information for the user.

The responsiveness of the system, especially the time between editing and executing, permits iterative development. If the time between editing and executing is too slow (the delay of the edit/compile/execute cycle) then the incentive to make small changes is lost. The actual quantitative measurement of 'too slow' cannot be specified, but if a small change takes long enough for the user to feel the need to wander off for a cup of coffee, then iterative development is prohibited.

Incremental program development permits iterative development at the fragment level. Once a fragment is developed it can then be combined to build the program. This increases the responsiveness of the environment as the whole program does not have to be executed to test a particular fragment.
1.2. The Read-Eval-Print Loop.

The read-eval-print loop (or simply, eval loop) is a language evaluator working in a cycle of reading a phrase, evaluating it, and printing the result. Although the eval loop is particularly suitable for expression-based languages such as LISP and ML that return a result for each expression, the eval loop is an implementation decision and not tied to any one language style. Batch (non-eval loop) versions of ML exist, for example New Jersey ML (Appel and MacQueen 1987) has a batch version. As well, eval loop implementations of languages usually thought of as batch exist, for example Pathcal, a programming environment for Pascal (Wilander 1980).

The eval loop has a binding environment and a state. The binding environment is the set of bindings defined in the eval loop. The state consists of an internal state, called the store in ML, and an external state. The store is the set of memory locations and their contents. These memory locations can have their contents side-effected (by an assignment statement). The external state consists of objects outside the eval loop such as files and interactive input/output.

A phrase is either a definition or an expression of the programming language, or a programming environment command. A definition adds a set of bindings to the binding environment. An expression does not define any bindings but evaluates using the binding environment and the state. Both definitions and expressions may side-effect the state. Some languages, such as LISP, draw no distinction between language constructs and programming environment commands, forming the basis of mono-lingual programming environments (Heering and Klint 1985).

The phrases entered in an eval loop can be viewed as either a history or a script. The history is simply the sequence of phrases entered in the eval loop. The script is a sequence of expressions and definitions resulting from the interaction so far in the eval loop. For example,

```
enter phrase: define \[ f = \lambda x.x; \]
enter phrase: edit \[ f \] to define \[ f = \lambda x.x*x; \]
```
can be viewed as the history

\[ \text{define } f = \lambda x.x; \text{, edit } f \text{ to define } f = \lambda x.x*x; \]

or the script

\[ \text{define } f = \lambda x.x*x \]

The eval loop has significant advantages for program development. It is naturally responsive as phrases are dealt with as entered. For example, a compiling eval loop need only compile the most recent phrase, previous phrases are already compiled. Such a compiling eval loop is therefore an incremental compiler. Incremental program development is also supported by the ability to define, execute and test program fragments. In contrast, a batch implementation requires a fragment to be placed in a complete program, along with test code, which is then compiled before being tested.

Incremental program development supports modular programming where a program is decomposed into sub-programs that are written before being combined to solve the final program. Incremental program development permits these sub-programs (and their sub-programs in turn) to be tested before they are combined.

1.3. ML.

ML is a modern general purpose programming language (Milner et al 1990). It has facilities to support both functional and imperative programming styles. It has static scoping, strong static typing with (parametric) polymorphism, type inferencing by the compiler, abstract data types, exception handling and modules. It is the most widespread language in this class and won the 1987 British Computer Society Technical award. Harper (1986b) and Wikstrom (1987) are introductions to ML. The growing recognition of ML can be seen in its use as an introductory programming language for computer science courses at universities around the world (for example, Cornell and Cambridge) and its use in texts on programming languages (Sethi 1989, Reade 1989, Watt 1990).
There are three major “checkpoints” in the history of standard ML. The first, (Milner 1985), gave a semi-formal description of the core ML language (basically ML without modules). A formal presentation (Harper et al 1988) included a revised version of exceptions and a proposal for modules. And finally version 4 of the standard (Milner et al 1990) is the current formal definition.

ANU ML (Cardelli et al 1991) was used in this thesis. It builds on an earlier implementation (Cardelli 1983b) and although at present it does not implement Standard ML, work continues to move it towards the standard. The main area that ANU ML differs from the standard is in its module facility. See Appendix 1 for details on the differences and a brief description of the module system in ANU ML.

Except for comments on the module facility, all discussion of ML, while directly applying to ANU ML, also applies to standard ML.

1.4. An Eval Loop Programming Environment For ML.

As stated earlier, the eval loop is a natural base for a programming environment. Indeed, given a LISP implementation on an eval loop it is difficult to distinguish the eval loop implementation and the programming environment (Sandewall 1978).

ANU ML and all other available ML implementations, RML from Rutherford Laboratory in Cambridge, SML from Edinburgh, Poly-ML (Matthews 1988a) and New Jersey ML (Appel and MacQueen 1987), are based on the eval loop. The building of a programming environment for ML may therefore appear a straightforward activity.

LISP, however, has considerable language features that can be used to offset deficiencies of the eval loop as the base of a programming environment. A language such as ML does not have these features and an environment for ML must extend the eval loop to overcome these deficiencies. The design and building of an environment for ML therefore provides a means for studying programming environments based on the eval loop.
The deficiencies of the eval loop are discussed in the following sections, along with the language features of LISP that avoid them.

1.4.1. Editing Definitions.

Most languages interacting through an eval loop (such as LISP) have dynamic outer-level scoping. Changing an existing definition is done by entering a new one; this new definition replaces the old one. Consider the system with dynamic outer-level scoping where \( f \) and \( g \) are entered

\[
\text{enter phrase: } \text{define } f = \lambda x. x; \\
\text{enter phrase: } \text{define } g = \lambda x. (f \ x);
\]

To change \( f \) the user would now enter

\[
\text{enter phrase: } \text{define } f = \lambda x. x \ast 10;
\]

and the call \( (g \ 5) \) would use the most recent definition of \( f \) in the top-level environment, and evaluate to 50.

In contrast, the basic eval loop implementation of ML cannot change existing definitions, because ML has static outer-level scoping and in the above example the expression \( (g \ 5) \) still evaluates to 5. The new definition of \( f \) does not affect the definition of \( g \). As existing definitions are not replaced by entering a new definition, a mechanism for editing existing definitions is required.

Here, the security offered by static outer-level scoping is at odds with the flexibility required to change definitions that is offered by dynamic outer-level scoping.

As well, in LISP, each declaration defines one identifier. In ML it is possible to define more than one identifier, for example, in a \texttt{val} declaration with more than one identifier on the left hand side, or in a mutually recursive \texttt{fun} declaration, or in an \texttt{abstype} declaration. It is not possible therefore to simply add a new definition for the identifier, the whole definition involving several identifiers would have to be re-entered.
1.4.2. The Text of Phrases.

The eval loop does not maintain text copies of the phrases entered. For example, in the above ML implementations, it is not possible to examine the phrase that defined an identifier. LISP avoids this problem as phrases are entered and stored as LISP data structures (S-expressions). The LISP command \texttt{pp} then can pretty-print an identifier by finding its binding (which is the phrase that defined it, as there is a mapping from identifiers to phrases) and displaying its associated data structure. Similarly, a LISP edit command can use the data structure associated with an identifier as the initial form to be altered.

1.4.3. Input Errors.

The eval loop does not maintain the text of invalid phrase that are entered, but rejected by the evaluator. The user must re-enter the phrase, either by re-typing it or by cutting and pasting an altered version of the phrase. As LISP has a simple syntax and no semantic checking (such as type checking), few phrases are rejected as invalid by a LISP system. ML on the other hand, has a sophisticated type system that catches many program errors statically, and therefore will more often reject phrases as invalid.

1.4.4. Aggregate Expressions and Declarations.

The eval loop provides the flexibility of incremental development, the user can enter expressions to test and debug them. To build a program, however, the user often wants to combine expressions into aggregate expressions or declarations; for example, to combine declarations and an expression into a \texttt{let} expression in ML to hide the local declarations of the expression, or to combine a datatype declaration and functions operating on this datatype into an \texttt{abstype} declaration. The converse operation of articulating an aggregate into individual phrases is necessary when debugging; for example, if a \texttt{let} expression does not produce the correct value, the whole expression must be decomposed so that the hidden declarations can be tested. The eval loop provides no support for aggregates. LISP implementations do not have this problem as LISP does not have such constructs in the language.
1.4.5. The Program Listing.

Normally, the eval loop does not maintain a program listing; the user must keep an external program listing, edit this listing, and then use it when entering to avoid losing text if an input error occurs. In the worst case the user must re-enter the complete listing after each change, using cut and paste technology this can be reduced to cutting changes from the external program listing and pasting them into the eval loop.

The eval loop offers the convenience of interactively creating a program 'state' (the state of the eval loop) for developing a program. This interactive convenience is negated if the user must simultaneously maintain a program listing. LISP systems typically offer the ability to save individual definitions or the whole state, but not a program listing that accurately describes the state.

1.4.6. Re-testing.

Although the eval loop provides the ability to test program fragments as they are developed, these test are not retained. If definitions are altered, the tests must be re-entered to test the program fragment. The user must remember the result of the test. This is a problem in all eval loop systems, including LISP implementations.

1.4.7. Debugging.

Incremental program development allows many errors to be detected by isolating them in a small expression. Still, debugging tools are required for a programming environment. A trace mechanism allows functions to display values on call and the result of the function. A debugger allows execution to be stepped through and values changed as required. LISP implementations are typically interpreted (in debug mode as least) and can also exploit the reflexive nature of LISP to extract run-time structures as LISP data structures when building mechanisms for tracing and debugging. A compiling eval loop (such as ML systems) must be engineered to provide debugging mechanisms.
1.5. AIMLESS.

AIMLESS is An Interactive ML Editing Support System. Originally the name of the editing component of a programming environment, it is now used to describe the whole environment. AIMLESS, as described in the following chapters, is implemented and currently runs on Sun (680x0) workstations and Vax and Pyramid computers.

The editing operations in AIMLESS allow an existing definition to be altered. The text of phrases is kept and invalid phrases can be edited. Aggregates can be combined and articulated. A program listing is maintained. The re-test mechanism views expressions as checkpoints of program behaviour. A test that contains a changed definition is automatically re-evaluated and a warning given of any changed behaviour. The trace mechanism permits functions to be traced, displaying arguments on call and the result of the function on evaluation. Mechanisms exist for packaging developed definitions into a program or a module and adding the program or module to file store.

1.6. An Outline Of The Thesis.

This chapter has given an introduction to programming environments, the eval loop and the programming environment AIMLESS. Chapter 2 outlines previous work in programming environments and then develops specific design decisions for an environment for ML. Chapter 3 is the central practical work of the thesis. It describes the editing system, the central component of AIMLESS. The implementation of the editing system is described in Chapter 4. Chapter 5 describes the debugging tools of AIMLESS; the re-testing and tracing mechanisms. The relationship between AIMLESS and the 'outside world' is then examined in Chapter 6. Finally Chapter 7 contains some conclusions of AIMLESS and the themes it explored.
Chapter Two: A Review Of Programming Environments

The previous chapter characterized programming environments as interactive, integrated tool sets for programming in the small. This chapter examines existing programming environments.

The first section reviews the programming environment landscape, outlining some programming environments which are discussed in the following section. Section three examines programming environments based on an eval loop which are then discussed in section four. Finally, section five motivates the design decisions for AIMLESS based on the previous discussion.

2.1. Some Programming Environments.

This section outlines some representative programming environments. The particular selection of programming environments does not necessarily comprise the ‘best’ environments; they were chosen as a basis for discussing programming environments design since they have all explored the issues of functionality, integration and interaction.

The environments are described only in sufficient detail to provide a framework for discussion. Later sections referring to the environments concentrate on particular features under discussion.

2.1.1. Interlisp.

The Interlisp environment was developed from 1972 at Xerox Palo Alto Research Centre based on a sequence of BBN LISP systems (Teitelman 1978; Teitelman and Masinter 1981).

The Interlisp environment was developed as an environment for exploratory programming by expert users, based on an eval loop. It encouraged users to add to the environment, giving a rich set of highly integrated, extendible tools. As discussed in chapter one, the language Interlisp is based on, LISP, offers language advantages for both the basic programming environment and for extensions.
2.1.2. The Cornell Program Synthesizer.

The Cornell Program Synthesizer (CPS) was developed at Cornell University (Teitelbaum and Reps 1981). The environment supported PL/CS, an instructional dialect of PL/I. This initial environment was further generalized to the Synthesizer Generator (Reps and Teitelbaum 1985) which could be used to generate syntax-directed editors for arbitrary languages.

CPS viewed a program as being a hierarchy of structurally nested components. This was reflected in the template editor for entering programs. Interpretable code was generated on entry and execution was based on this template model.

2.1.3. An Incremental Programming Environment.

An Incremental Programming Environment (IPE) was developed at Carnegie-Mellon University (Medina-Mora and Feiler 1981). The environment supported the language GC, a variant of C with typing and module structure.

IPE attempted to offer in a compiler based system the degree of interaction usually found in an interpretive system. It used a fully template driven editor. When the editor left a procedure template, if the procedure was valid, it was automatically compiled, linked and loaded.

2.1.4. Magpie.

The Magpie programming environment was developed at the Computer Research Laboratory of Tektronix (Delisle et al 1984; Schwartz et al 1984). Magpie, an environment for Pascal, was built on a single-user workstation with high-resolution bitmapped display with mouse.

Magpie was built on a windowing environment and inherited many ideas from CPS and IPE. The type of a window, either code browser, heap browser, event monitor or workspace, defined a mode of user interaction. Compilation of procedures occurred in the background.
2.1.5. PECAN.

The PECAN programming environment generator system was developed at Brown University (Reiss 1984a; Reiss 1984b; Reiss 1984c). The PECAN system took a specification of language syntax and semantics and generated a programming environment. Only one major environment, one for Pascal, was ever built; the specification of language semantics being complex to detail for an arbitrary language. The name PECAN will be used for this Pascal programming environment.

PECAN was a highly graphical environment with many simultaneous static and dynamic views of the program under development. Views included a syntax-directed editor, a Nassi-Shneiderman editor, a symbol table display, a data type display, a flow graph display and an execution tool.

2.1.6. Glide.


Glide was based on an eval loop. It allowed top-down programming for a language with strong static polymorphic typing and static scoping.

2.2. Discussion Of These Programming Environments.

Chapter 1 discussed programming environments as providing a basic functionality for developing (possibly incomplete) programs, being integrated and being interactive. All the environments above allowed incomplete programs, all were integrated. Interlisp (in its graphical form Interlisp-D), Magpie and PECAN were highly graphical.

Not all the programming environments were incremental. The two eval loop environments, Interlisp and Glide, were incremental. Magpie provided workspaces for evaluating expressions in the current program state. CPS, IPE and PECAN required the whole program to be executed, although after a change (say on stopping
Chapter 2: Review

at an incomplete fragment) execution could be continued at the point it stopped.

The environments all attempted to be responsive from editing up to execution. Interlisp was usually interpreted, eliminating any translation costs on editing at the expense of slower execution. Functions in Interlisp could be compiled, but this required an explicit request from the user. All the other environments used some translation to improve execution speed.

CPS, PECAN and Glide compiled an intermediate form that was interpreted, while IPE and Magpie compiled machine code that was directly executed. All these environments used incremental compilation to reduce their translation costs. Incremental compilation depends on various implementation techniques and language features. As the ML system that AIMLESS is based on uses compilation, it is useful to examine these techniques and features now in the context of the above environments. Later discussion of compilation can then refer to these points.

2.2.1. The Unit Of Change.

The unit of change is the smallest program fragment that is translated after editing steps. In a traditional programming environment, such as Berkeley Pascal (Joy et al. 1977), the unit of change is any file containing changes. In contrast, CPS and PECAN had expressions as the unit of change while IPE, Magpie and Glide had procedures as the unit of change. The size of the unit of change in a programming environment is a trade-off between the reduction in translation costs for a smaller unit of change and the overheads associated with composing units of change into larger program constructs.

As a program is developed, program fragments must be composed into larger constructs. For example, an if-then-else statement is the composition of a boolean expression and two statements. Fragments can be composed in two ways; in-line or by reference. An in-line fragment is one incorporated directly into the larger construct, while a reference in the larger construct is a pointer to the fragment.

In CPS and PECAN the units of change (expressions) were composed into larger constructs using references. For example, an if-then-else statement was (in
effect) represented as a triple \((ptr_1, ptr_2, ptr_3)\) where \(ptr_1\) was a reference to the code for the condition, \(ptr_2\) a reference to the code for the then branch, and \(ptr_3\) a reference to the code for the else branch. To change an expression in a larger construct the reference to the old expression need only be replaced by the new reference, reducing the translation costs. IPE, Magpie and Glide had expressions compiled in-line and so a change of an expression necessitated the surrounding procedure being re-compiled.

![Fig. 2.1 Direct and Indirect References: The difference between the two methods can be seen. In direct reference the address of the body of function \(f\) is used in the larger construct. In indirect reference the address of where to look for the address of function \(f\) is used.](image-url)
IPE, Magpie and Glide also had their units of change (procedures) composed using references. Magpie used direct references, IPE and Glide used indirect references. See Figure 2.1 for a schematic view of the difference between direct and indirect references. The difference between direct and indirect references comes into effect when a procedure is changed. In Magpie all occurrences of the direct reference to the changed procedure must be adjusted. In IPE and Glide the indirect pointer is adjusted to point to the changed procedure. Change using an indirect reference is shown in Figure 2.2.

![Indirect Pointer Diagram](image)

**Fig. 2.2 Change Using An Indirect Reference:** The indirection pointer is changed to point to the new code of \( f \). The code in the larger construct does not change, but now refers to the new code of \( f \).

### 2.2.2. Outer-level Scoping.

The outer-level scoping rules determine the binding of identifiers to values. For example, consider the following definitions.

\[
\begin{align*}
\text{define } f &= \lambda x. x + 1 \\
\text{define } g &= \lambda x. (f \ x) \\
\text{define } f &= \lambda x. x + 2 \\
\text{g } 5
\end{align*}
\]

The definition of \( g \) contains free variable \( f \). Evaluating \( g \ 5 \) reduces to evaluating \( f \ 5 \). Two bindings to \( f \) exist; \( f \) bound to \( \lambda x. x + 1 \) and \( f \) bound to \( \lambda x. x + 2 \). The
outer-level scoping rules determine which binding is used. Static (or textual) outer-level scoping uses the first binding to \( f, (\lambda x.x+1) \), as this binding existed when \( g \) was defined. Dynamic outer-level scoping uses the second binding to \( f, (\lambda x.x+2) \), as this is the most recent binding for \( f \).

The outer-level scoping rules are analogous to the expression scoping rules within expression evaluation for determining bindings to free variables in functional arguments. The outer-level and expression scoping rules need not be the same in a language. Interlisp had dynamic outer-level and expression scoping. ML has static outer-level and expression scoping. Scheme (Steele and Sussman 1975) and Common LISP (Steele 1984), in comparison, have dynamic outer-level scoping and static expression scoping.

In an eval loop system where definitions are added interactively, outer-level static scoping gives security against naming conflicts. Values cannot change underfoot as the accidental entering of a definition with the same name hides the previous definition, but does not affect definitions using this previous one.

An alternative description of scoping (Nikhil 1990) regards dynamic outer-level scoping as a re-organization of the outer-level definitions. In the original example, in a system with dynamic outer-level scoping, after the second definition of \( f \), the definitions would be viewed as

\[
\text{define } f = \lambda x. x+2 \\
\text{define } g = \lambda x. (f \; x)
\]

Such a description would view static outer-level scoping as not re-organizing the definitions. This alternative description is, however, a question of terminology, not a question of semantics.

Haskell (Hudak et al 1990) leaves the problem of dynamic or static outer-level scoping as a programming environment question, not part of the language itself.

Languages with outer-level dynamic scoping must determine the binding to the definition at run time. CPS and PECAN had the cost of looking-up the binding to a definition existing at run time.
Languages with static outer-level scoping are traditionally compiled with direct references determined at compile time. The use of direct references, however, is a disadvantage for incremental compilation as if a fragment is changed, all definitions referring to the changed fragment must be adjusted to refer to the changed fragment.

Static outer-level scoping is often regarded as synonymous with direct references. As pointed out by Nikhil (1985) this is not true. Glide had static outer-level scoping implemented with indirect references. Thus in Glide when a definition was edited, the resulting compilation did not have to adjust references to this definition. Glide required definitions to be in the correct context and so enforced static outer-level scoping.

### 2.2.3. Typing

Typing is a valuable mechanism in program construction for checking the correct use of components in a program. Dynamic type systems perform type-checking at run time. Static type systems, such as in ML and Pascal, perform typechecking before a fragment is executed.

Interlisp had a (very weak) dynamic type system and did not have to be concerned with typechecking as definitions changed. At run time if a type error occurred, such as taking the CAR of a number or not finding a variable on look-up, an error was reported and execution stopped.

"Static typing is firmly established as a fundamental tool in building large, highly structured and reliable software systems" (Cardelli 1987). Static typechecking gives the security of knowing that a typechecked fragment will not generate a type error at run time as well as allowing more efficient code to be compiled. For incremental compilation, however, there is the cost that when a declaration is added or is changed, all parts of the program affected by the declaration must be re-typechecked. CPS, IPE, Magpie, PECAN and Glide all had this re-

+ Following the precedent of Cardelli and Toyn, typechecking is used as one word to permit the use of re-typechecking.
typechecking overhead. Magpie and Glide had algorithms that reduced the amount of re-typechecking necessary. Still, static typing implies that a type change in a unit of change may require compilation outside the unit, possibly of the whole program.

2.3. Eval-Loop Based Environments.

Two approaches in programming environment design can be identified in the above environments. One approach sought to reduce the edit/compile/execute cycle, the other approach had the environment based on an eval loop.

CPS, IPE, Magpie and PECAN reduced the edit/compile/execute cycle - the time taken between editing and being able to execute the program. A program listing was edited and the listing (incrementally compiled) was used for execution. Magpie provided workspaces that allowed program fragments to be evaluated. This provided some of the functionality of an eval loop; editing in Magpie, however, was still done on the program listing.

Interlisp and Glide were programming environments based on an eval loop. No explicit notion of the program listing existed within the eval loop. Interlisp and Glide maintained external records of individual definitions. These were then manipulated outside the eval loop as the program listing.

Some programming environments based on the eval loop are now reviewed. The list was chosen to illustrate the spectrum of eval loop systems in programming environments. As the eval loop is an implementation decision, specific environments are cited for BASIC and LISP. Most implementations of these languages are based on the same eval loop ideas. The implementations chosen reflect local availability and knowledge rather than any specific design issues.

2.3.1. Atari BASIC.

Atari BASIC (Albrecht et al. 1979) was similar to many BASIC programming environments. The eval loop read a phrase in direct or indirect mode. Direct mode phrases were executed. Indirect mode phrases were signified by a line number before a BASIC statement and were added to the program. The program could
then be executed by the direct mode command RUN or GOTO <line number> to execute from a given line number.

The eval loop could be thought of as defining a binding environment, binding line numbers to BASIC statements. The program then was the listing defined by ordering line numbers. The binding environment could be incomplete, with invalid lines or with references to non-existent line numbers, and was edited by replacing an existing line.

2.3.2. Cardelli’s ML.

Cardelli’s ML, or CML, was an ML compiler rather than a programming environment (Cardelli 1983b). The compiler was based on an eval loop; phrases were entered and compiled. As ML has static outer-level scoping, all global variables could be typechecked and their direct references determined at compile time.

The expected mode of operation was to have text files as the listing which would be edited outside the eval loop. Either the necessary text files would be re-loaded after editing, or if cut and paste facilities were available (as in emacs or a windowing system), these would be used. No editing or debugging facilities were provided in the eval loop, its functionality being seen as sufficient.

2.3.3. Interlisp.

As described in the general overview of programming environments, Interlisp had a highly integrated and extendible set of tools aimed at the expert user. In this system, a history was maintained, recording the user’s input, a description of the side-effects of the operation and the result of the operation. Items in the history were numbered and could be undone or re-done. The history was maintained for operating in the eval loop but the binding environment was used for editing and generating the program listing. As the Interlisp LISP dialect had dynamic outer-level scoping a definition could be replaced by entering a new definition.

The Interlisp file package kept track of where objects (definitions, values) in the eval loop were saved in the file store. The user specified the various files to be used and maintained on entering a session. On demand, the file package would
retrieve objects that had not been explicitly loaded into the environment. It also maintained information on what objects needed to be saved (i.e., objects in the environment that had been edited but not updated in file store) and any object created but without a file entry. Routine Cleanup existed to do this tidying up on demand or on exiting the session.

2.3.4. Nuprl.

Nuprl (Constable et al 1986), a programming system for developing mathematical proofs, operated in an eval loop. The actual proof system is not important for this discussion. The programming system, however, used ML to code tactics for proof procedures. The development of the ML tactics in the eval loop is relevant.

The Nuprl editing system had a library, a list of objects defined by the user. An object was either a theorem, a definition (a text macro), an evaluation binding, or ML code. Each object had an associated name. An object in the library had a status: raw, bad or complete. Raw indicated unchecked text; bad indicated text with an error; complete signified a valid object.

Objects were indexed by their name, or their position relative to another object, the current pointer or the beginning or end of the library. The main editing commands included create, delete, check, move, jump and view. Command create took a name, a kind and a place, and added the object with name and type to the position specified in the library. Command delete removed the specified object. Command check compiled the object to see if it is valid in its context in the library. Command move moved an object or objects about in the library, while jump displayed the desired object. Command view put the specified object in the editor, ready for editing.

2.3.5. Glide.

As described in the general programming environment overview, Fly and Glide were programming environments for functional languages. Fly supported a language with eager evaluation based on the SECD model of computation. Glide supported a lazily evaluated language based on the combinator compilation model.
Chapter 2: Review

Glide evaluated a functional language with static outer-level scoping and strong (polymorphic) typing. Phrases entered were compiled into intermediate form and which was then interpreted. Compilation used an indirection reference for identifiers. “This allows one to edit a definition and allow other expressions automatically to receive this latest value. (This is not to be confused with ‘dynamic scoping’..)” (Nikhil 1985).

In Glide the textual bodies of definitions were stored externally, an edit command changed the external text and internally changed the indirection definition. Therefore, an external listing was maintained based on the definitions in the environment.

Glide’s typechecking algorithm reduced the effect of type errors by assuming that the type of an erroneous expression was as expected by the typechecker. An invalid expression was compiled as a call to procedure typefault. The evaluation of typefault stopped execution and reported an error.

When a definition was changed its use within other definitions had to be typechecked. As opposed to re-typechecking all definitions, clever analysis limited the amount of typechecking needed, while ensuring all affected definitions were typechecked.

2.4. Discussion Of Eval-Loop Based Environments.

Some of the issues relevant in a programming environment based on an eval loop are discussed in this section. The issues deal with changing the program under development: how changes can be made, keeping track of the program under development, and making sure that the saved program matches the changed program.

2.4.1. Edit Operations.

Editing within the eval loop (as opposed to editing an external file) can be based on the binding environment or on the script.
Chapter 2: Review

The natural unit of change when editing based on the binding environment is the declaration defining the identifier. The basic operations required for editing based on the binding environment are insert, delete and replace. Operation insert adds a binding in an appropriate context to the binding environment, delete removes a binding, and replace replaces an existing binding with a new definition.

In an eval loop with dynamic outer-level scoping such as Interlisp or Atari BASIC, entering a definition is equivalent to an insert. Entering a definition is also equivalent to a replace. The scoping rules ensure that any calls to the old definition will now use the new definition. The delete operation need only remove the binding from the environment.

In an eval loop with static outer-level scoping entering a new definition does not affect previous bindings. An insert operation must be able to specify a particular context, so that the new binding is used by existing bindings. A replace operation must be able to specify the particular instance to be replaced. The delete operation must be able to remove a binding. Any bindings that used the removed one now must be able to use bindings previously hidden. For example,

\[
\text{define } f \ x = \lambda x. x \\
\text{define } f \ x = \lambda x. x * x \\
\text{define } g \ x = \lambda x. f \ x
\]

Now if the second definition of \( f \) is removed, \( g \)'s definition should use the first definition of \( f \).

In a static outer-level scoping system with static typechecking such as CML and Glide, an insert, replace or delete creates a new context, requiring all definitions dependent on changed definitions to be re-typechecked.

In a system such as CML using direct references, an insert, replace or delete required all occurrences of changed references to be adjusted or re-compiled. In systems such as Atari BASIC and Glide with indirect references, changed definitions did not have to have their references adjusted.
Chapter 2: Review

It is not always possible to edit based on the binding environment. In ML a phrase may define more than one identifier and these may have to be edited at the same time. For example, for some expression $EXP$ the phrase $\text{val} (a, b) = EXP$; will define identifiers $a$ and $b$ and it is not possible to edit the "definition" associated with either of these two identifiers separately.

An alternative view of editing in the eval loop is to base the edit operations on the script and have the changes to the script reflected in the binding environment and state as in Nuprl. This view is particularly appropriate in a loop with static outer-level scoping where the order of definition is important. The primitive editing operation in an eval loop is to append a phrase $p$ to the script $< p_1, \ldots, p_n >$,

$$\text{append}(p, < p_1, \ldots, p_n >) \Rightarrow < p_1, \ldots, p_n, p > .$$

A more flexible range of operations would include (with $<>$ the empty script)

$$\text{reset}(< p_1, \ldots, p_n >) \Rightarrow < > .$$
$$\text{delete}(i, < p_1, \ldots, p_i-1, p_i, p_{i+1}, \ldots, p_n >) \Rightarrow < p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n > .$$
$$\text{insert}(i, p, < p_1, \ldots, p_i, p_{i+1}, \ldots, p_n >) \Rightarrow < p_1, \ldots, p_i, p, p_{i+1}, \ldots, p_n > .$$
$$\text{replace}(i, p, < p_1, \ldots, p_i, \ldots, p_n >) \Rightarrow < p_1, \ldots, p_{i-1}, p, p_{i+1}, \ldots, p_n > .$$

Nuprl had names for each phrase to be able to specify the particular phrase to be edited. Again, in static outer-level scoping systems with static typechecking, delete, insert and replace create new contexts that must be typechecked.

When editing based on the script, the natural unit of change is the phrase. The phrase, however, can sometimes be too coarse-grained (too large a unit of change). For example, an ML phrase can be large, a declaration of mutually recursive definitions; changing one definitions means changing (re-evaluating) all definitions in the declaration. See Section 7.1. for a discussion of how an individual definition in a mutually recursive definition could be edited. This would not be a problem in a language where mutually recursive definitions were broken-up by forward declarations. Other constructs such as let, local and abstype can be viewed as aggregates of smaller phrases, and if necessary, can be decomposed before editing (see Section 3.1.9).
2.4.2. The Program Image and The Program Listing.

There are two aims of programming in an eval loop; firstly, to enter phrases into the eval loop to produce a state where expressions will evaluate as desired, and secondly, to produce a record in some notation describing the phrases that produced such a state. This state of the eval loop will be known as the *program image*, the record will be known as the *program listing*.

In one-off programming only the image is necessary, say to calculate a result or to perform some task. More generally the listing is required to communicate what operations are performed by the image and how they are performed - “a computer language is not just a way of getting a computer to perform operations, but rather ... it is a novel formal medium for expressing ideas about methodology.” (Abelson and Sussman 1985).

The eval loop builds a program image which is the binding environment and the state of the eval loop. A program listing is not constructed. To construct a program listing either the user can be made to maintain an external listing, or the eval loop can provide mechanisms to generate a listing.

If the user is made to maintain the program listing, as in CML, the working cycle soon degenerates. The user can work in the eval loop developing the image, and then edit the listing, hoping to capture developments in the image. This is unacceptable as it requires the user to remember all changes. Or the user can maintain an external listing (in a file or files), edit this as required and then re-load changes and test these. In the worst case, the user would re-load the entire listing (the edit/compile/execute cycle re-visited). A better solution (and the one most often used for the current ML systems without an editing mechanism) is to use cut and paste facilities to add changes. Such editing is not, however, a solution and should be seen as a kludge to overcome the problem of maintaining a program listing. As a programming environment, the editor and the compiler do not share a common representation of the program under development and integration is not supported. It would be difficult to implement a command `whatdefn` to display the phrase defining an identifier or to avoid re-compilation when changing
a definition. Using cut and paste reduces the inconvenience of the eval loop not maintaining a program listing, but it does not provide a general editing mechanism for a programming environment. As well there are technical problems relying on cut and paste editing; users are tied to a particular editor (for example, emacs) or must have access to a windowing system, and problems can exist if primitives are re-defined. For example, if the definition of f uses primitive hd that is re-defined after its use in f, if f is altered and entered in the eval loop, it will use the new version of hd rather than the old one.+

The eval loop can provide mechanisms to generate a listing. The binding environment, or more strictly the definitions defining the bindings, may form a listing. Interlisp keeps the definitions of entries on the binding environment and uses these as the listing. Similarly Atari BASIC keeps its indirect mode phrases and uses these as the listing. This listing, however, will not include expressions side-effecting the state, so the listing may not correctly describe the image. This problem of consistency is discussed in the next section.

While the binding environment can be used to provide current values, it cannot describe how the current values came to be. For example,

\begin{verbatim}
define x = 3
define y = x+2
\end{verbatim}

The binding environment can provide the current values of x and y, 3 and 5 respectively, but not the relationship between x and y.

Even if the binding environment has mechanisms for describing values when extracting a listing, the listing may not describe the image correctly. Consider the example

\begin{verbatim}
define x = 3
define y = x+2
\end{verbatim}

+ This is not a made-up example. One of my first experiences with ML was writing an evaluator where I re-defined hd and encountered this problem.

27
Chapter 2: Review

If the mechanism extracting the listing from the image is clever enough to know that \( y \) was defined as \( x+2 \), if \( x \) has been changed by a replace to say 6, unless the image has the value of \( y \) as 8, the listing will not describe the image.

The script can be used as a listing. However, unless mechanisms exist to indicate phrases that are part of the listing and phrases that are not, the script will contain many redundant phrases. These redundant phrases reflect the editing, testing and debugging of program development. A program listing should not contain such irrelevant phrases.

If side-effects do not have to be considered, as in pure functional systems, only definitions in the script form the listing. If side-effects exist, all side-effecting phrases are part of the listing.

2.4.3. Consistency.

As discussed above, the immediate aims of programming in an eval loop are to produce a program image performing certain operations and a program listing describing the operations performed by the image. An image and a listing are \textit{consistent} if they define the same program. Intuitively, this means if the listing is entered and evaluated, it will produce an image that behaves similarly. Formally, an image and a listing are consistent if the listing entered and evaluated\(^+\) in an eval loop generates the the same binding environment and state as present in the image.

Consistency can be broken into two categories: environment and state consistency. A listing and image are \textit{environment consistent} if the listing generates the binding environment of the image. A listing and image are \textit{state consistent} if the listing generates the state of the image.

In eval loop systems consistency is a difficult problem as the image and the listing have separate identities. The previous section outlined possible methods that could be used by programming environments for extracting or maintaining a

\[^+\] This definition assumes the evaluation of the listing is deterministic (e.g., does not contain definitions that rely on specific input values).
Chapter 2: Review

listing from an image. These techniques are not always consistent. For example, in CML simply adding a definition to the image and not to the listing makes the two inconsistent. Atari BASIC is not state consistent as side-effecting direct mode phrases are not kept as part of the listing.

The argument for consistency is that the user can be sure that behaviour observed using the image is captured by the listing. The user does not have to consider two worlds, the image and the listing, but instead can develop a program image and have it described by a consistent listing.

2.5. The Design Decisions For AIMLESS.

The previous sections in this chapter have reviewed and discussed existing programming environments. This section now examines the design decisions for AIMLESS.

The first decision is to base AIMLESS on an eval loop. Given that AIMLESS is being used as a case study to examine programming environments based on the eval loop, this decision may seem obvious. It is included as an explicit design decision as a reminder that other options are possible for a programming environment for ML. Although the formal semantics of ML (Milner et al 1990) defines the language as interacting in an eval loop, the design of an ML programming environment could consider alternative approaches - for example, an environment similar to CPS, IPE, PECAN, or Magpie.

The approaches used by CPS, IPE and PECAN do not permit incremental program development and therefore were rejected. Incremental program development allows programs to be developed interactively; components can be entered, tested and altered (if necessary) before being used by other parts of the program. In contrast, in a non-incremental program development system, a component can only be tested as part of a whole program. This typically means that components are not tested individually; debugging is often the process of isolating parts of a program and then observing behaviour with a debugger to fix errors. Incremental program development, on the other hand, allows testing to be done bottom-up,
component by component, which means errors can be detected at the component level instead of when a whole program performs incorrectly.

The eval loop is preferred instead of the workspace approach of Magpie, even though it also permits incremental program development. The workspace approach separates the listing and the interactive testing system and therefore requires a "context switch" when moving between entering definitions and testing. This separation also does not allow workspace commands that manipulate the program listing, such as commands for combining definitions into larger constructs, as can be done in an eval-loop based system.

With the programming environment based on an eval loop, there are two possibilities (as discussed in Section 2.4.1.) for editing: the binding environment or the script. Editing based on the binding environment would have the flavour of "change the definition of $f$ to ..." or "do a text edit on the definition of $f$". ML permits multiple bindings in one value binding, for example in val $(a, b) = EXP$; . It is not clear what editing this definition of $a$ by itself in the above phrase would mean. For this reason, AIMLESS uses editing based on the script. Thus phrases are edited as units and in the above example neither $a$ or $b$ would be edited separately, they would only be edited as part of the whole phrase.

Editing at the phrase level ties in naturally with the phrase as the unit of entry into the eval loop. Similarly, on an error, the phrase is a sensible unit for indicating an invalid component. The drawback with using the phrase, however, is that it can be a large unit of change, affecting the turnaround time of the environment after a change. For example, a large mutually recursive declaration can contain many definitions, changing one then involves re-compiling all the definitions in the phrase. (This disadvantage is examined further in Chapter 7.)

As described in Chapter 1, the eval loop has several deficiencies as the base of a programming environment. While some languages (such as LISP) have language features that overcome these deficiencies, ML does not have these features; indeed, ML's design is counter to those required. For example, ML's static outer-level scoping is contrary to the ability to edit definitions in the eval loop.
Chapter 2: Review

The approach taken in Glide, a language with similar contrary features such as static outer-level scoping and strong typing, was to find programming environment solutions by using dynamic binding for identifiers to definitions and introducing procedure `typefault` as the value of invalid components. Such an approach could be taken for a programming environment for ML in an eval loop.

The “contrary features” of ML, however, were designed deliberately so that ML would be a language for writing secure, modular programs. The strong static type system detects all type errors at compile time. Coupled with the exception system an ML program never “crashes”; it either returns a value, loops indefinitely, or raises an exception. Static outer-level scoping gives the security that definitions cannot change underfoot, as a later definition to the same name will not change any earlier bindings. Static outer-level scoping ensures that the current program under development is well formed and can be reasoned about as a mathematical object, as in the definition of ML (Milner et al 1990). The aggregate constructs in ML encourage modularity and information hiding as local declarations are “hidden” from the rest of the program.

In an eval loop system ML encourages the writing of secure, modular programs from known, tested components. Static outer-level scoping ensures that components must be known (already defined), this is consistent with the strong static type system needing to know the type of components when they are used. Components can be tested in the eval loop before being used and can be combined into aggregates to produce modular programs.

The next design decision is therefore to reject the approach taken by Glide and instead attempt to build a programming environment that supports the model of ML as writing secure, modular programs from known, tested components.

The major consequence of this decision is that in AIMLESS, the eval loop “accepts” only valid phrases; invalid phrases are not permitted on the script. Glide allowed invalid phrases as part of the script, whereby “the programmer is free to develop the program without being inconvenienced [by type errors]” (Toyn et al 1987). If ML is to produce a secure program, type errors are not
inconveniences but indications that the program is wrong and must be corrected. It is against the spirit of ML's design to build an invalid program from invalid components. The requirement that the script be valid reduces the flexibility of the programming environment but increases the support given to the user for developing valid meaningful ML programs.

As well as being valid, the script should also be consistent. It is pointless to develop carefully a program from known, tested components within an eval loop, if a listing describing the image does not exist. Having a consistent script eliminates the error-prone task of the user manually maintaining a listing.

In summary, the four basic design decisions of AIMLESS are:

- to base the environment on the eval loop;
- to use the script for editing (in effect, to use the script as the representation of the program under development);
- to have a valid script;
- and to have a consistent script.

The design decisions of a valid and consistent script for AIMLESS as a programming environment appear restrictive. In a real sense they are. ML as a language has features designed to enhance the building of correct programs. These features are usually seen as contrary to programming environments and to be avoided. The AIMLESS design decisions recognize the problems introduced by language features such as static scoping and typechecking, but instead of avoiding them, seek to develop a programming environment that will support these features and incorporate their advantages into the environment. The challenge for AIMLESS is to solve the eval loop deficiencies described in chapter one, while working within the design constraints introduced in this section.
An editing system for ML is presented that gives the desired operations for editing a script - insert, delete and replace - while maintaining a valid and consistent script. It solves some of the deficiencies of the eval loop - editing facilities are available, the text of phrases is kept, input errors are handled, and aggregates can be combined and articulated.

The chapter first examines the design of such a system. The editing system of AIMLESS is described informally, to provide a working understanding of its operations. Although editing is described in terms of AIMLESS and ML, most of the discussion generalizes to any eval loop system that needs an editing system that maintains a valid and consistent script. A more formal description of AIMLESS is then given, followed by a summary of AIMLESS.

3.1. The Design Of The Editing System.

Some of the design issues encountered in an editing system that maintains a valid and consistent script are examined.

3.1.1. A Truncate/Reappend Editing System.

A model of editing phrases within an interactive system while maintaining a valid and consistent script is suggested by ideas presented in the programming environment COPE (Archer et al. 1984). This environment had the notion of a script, a sequence of commands, and was concerned with recovery facilities, undoing previously executed commands in the script.

Two extremes in editing a script were nominated: incremental editing and unrestricted editing. Incremental editing consists of appending commands to the script. Unrestricted editing allows for new commands to be inserted at any point in the script and for existing commands to be deleted, changed or re-ordered at will. COPE used truncate/reappend editing, a form of editing between incremental and unrestricted editing.
In a truncate/reappend system, the script is supplemented with an auxiliary script \( R \). The operations that characterize truncate/reappend editing (in the COPE notation), with script \( S = c_1, \ldots, c_n \) and auxiliary script \( R = r_1, \ldots, r_p \), producing a new script \( S' \) and auxiliary script \( R' \), are

\[
\text{append}(c, S, R) \Rightarrow S' = c_1, \ldots, c_n, c \quad R' = R
\]

\[
\text{remove}(S, R) \Rightarrow S' = S \quad R' = r_1, \ldots, r_{p-1}
\]

\[
\text{truncate}(S, R) \Rightarrow S' = c_1, \ldots, c_{n-1} \quad R' = r_1, \ldots, r_p, c_n
\]

\[
\text{reappend}(S, R) \Rightarrow S' = c_1, \ldots, c_n, r_p \quad R' = r_1, \ldots, r_{p-1}
\]

The operations truncate with an empty script and remove or reappend with an empty auxiliary script are illegal. Recovery is necessary and implicit on a truncate operation, the effects of the phrase moved from the script to the auxiliary script must be undone.

The edit operations insert, delete and replace can be implemented in a truncate/reappend editing system. To insert a phrase, phrases are truncated from the script to the auxiliary script, the new phrase appended, and the truncated phrases re-appended. To delete a phrase, phrases are truncated from the script to the auxiliary script, the phrase removed, and the truncated phrases re-appended. To replace a phrase, phrases are truncated from the script to the auxiliary script, the phrase removed, its replacement added, and the truncated phrases re-appended. If only valid phrases can be appended and re-appended, the script will always be valid. If truncated phrases are undone, the script will always be consistent.

The model of truncate/reappend editing can be applied to eval loop editing (with a change to a more convenient terminology). A phraselist consists of the script and a buffer (the auxiliary script). With the script \( < p_1, \ldots, p_i > \) and the buffer \( < p_{i+1}, \ldots, p_n > \) the phraselist (PL) will be represented as \( < p_1, \ldots, p_i | p_{i+1}, \ldots, p_n > \). The script represents phrases evaluated and affecting the top-level binding environment and store. The buffer contains phrases entered but not evaluated and
Chapter 3: The Editing System

therefore not affecting the top-level binding environment or store. The edit oper­
ations are

\[ \text{append}(p, PL) \Rightarrow PL'=< p_1, \ldots, p_i, p[p_{i+1}, \ldots, p_n] > \]

\[ \text{del}(PL) \Rightarrow PL'=< p_1, \ldots, p_{i+1}, \ldots, p_n > \]

\[ \text{pop}(PL) \Rightarrow PL'=< p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n > \]

\[ \text{push}(PL) \Rightarrow PL'=< p_1, \ldots, p_i, p_{i+1}, \ldots, p_n > \]

The operation \text{del} is used for removing extraneous phrases, \text{pop} and \text{push} are so
named to capture the idea of the script being a stack of phrases. Both \text{append}
and \text{push} assume that the phrase added to the script is valid. Thus the script is
always valid. The treatment of invalid phrases will be discussed in Section 3.1.5.

Some comments on this system of editing.

- The \text{append} operation corresponds to entering and evaluating a phrase in
the eval loop. The \text{append} command is implicit, any phrase entered that is not
an editing command, is a phrase to be appended.

- The \text{pop} operation, like the truncate operation, requires the effects of the
popped phrase to be undone. This ensures the script is always consistent. Undoing
the effects of a phrase is discussed in the next section.

- The \text{push} operation requires, at least logically, the pushed phrase to be re-
evaluated (compiled and executed).

- The replace operation is now: pop until phrase to be altered is on the buffer,
delete the phrase, append the new version and then push phrases that were popped
by this operation.

3.1.2. Undoing Phrases On The Script.

The major technical difficulty in a truncate/reappend editing system is the
undo mechanism associated with truncate. The COPE system explored methods
for providing an undo mechanism: simulating undo by re-running the new script
again from the initial state, saving state information for each entry in the script, and checkpoints which allow block truncates (truncate up to checkpoints only). The general conclusion was that an undo mechanism can be implemented but the operation is non-trivial because of a possibly complex state.

The pop in ML editing has two components that it must ‘undo’: the binding environment and the state. For example if phrase val x = 10; is popped, the environment must be restored to its previous bindings before the phrase was evaluated. If phrase x := 1; is popped, the state (the ML store) must be adjusted so that variable x has its previous reference value. External side-effects (such as writing to file store) caused by a phrase are outside the control of the editing system and are not undone on a pop. The script is therefore binding environment and store consistent but not external world consistent (see Section 3.3 for further comments).

The semantics of Standard ML presented in Section 3 of the Standard ML Report (Milner 1985) is expressed in terms of the generation of environments. Milner et al (1990) use a similar approach. A phrase $p_i$ evaluated in the top-level binding environment $ENV_{i-1}$ returns a binding environment $E_i$ and the new top-level binding environment $ENV_i$ is $E_i$ appended with $ENV_{i-1}$ so that bindings in $E_i$ supersede those in $ENV_{i-1}$. The operations append and push can be seen in these terms; adding new bindings to the existing binding environment. The operation pop has a natural interpretation as the inverse operation. pop removes those new bindings from the current binding environment, leaving the previous binding environment. This is a good model but two features are not explained: fixity changes and weak type instantiations.

This model fails to take into account changes in fixity; a phrase can change an identifier’s fixity (and if infix, its precedence and associativity). A pop must reset identifiers to their previous fixities. The above model can be used by including fixity information in the binding environment. The binding environment then has three components: the value, type and fixity environments.

The model also fails to describe weak type instantiation. The phrase val x
= ref [] ; adds a binding to the top-level binding environment of variable \( x \) with the type \( '_a \) list ref. The weak type variable \( '_a \) will be instantiated by a later phrase. For example, \( x := [1] ; \) will instantiate the type of \( x \) to int list ref in the existing top-level binding environment. The undo mechanism must undo weak type instantiations on a pop. In the example, popping the phrase \( x := [1] ; \) must reset the type of \( x \) to \( '_a \) list ref. Weak types were permitted at the top-level in (Milner 1985) and so were implemented in ANU ML and undone on a pop. As weak types are now illegal at the top-level, weak types cannot be instantiated and therefore do not have to be undone.

Store side-effects can be undone by re-building the store existing when the popped phrase was originally evaluated. As ML is primarily a functional language, the number of store side-effects caused by a phrase should not be large and the operation of re-building the store should not be prohibitive. The editing system need only maintain the previous values of altered reference objects and re-establish these on a pop. Note that only reference objects in the top-level binding environment need have side-effects undone. Temporary reference objects do not have to be considered unless they are an alias to a reference object in the top-level binding environment. For instance, in the following example, two temporary reference objects, count and result, are defined. A phrase that uses factorial will side-effect both count and result, but if the phrase is undone on a pop, the side-effects to these temporary references do not have to be undone.

```ml
fun factorial n = 
  let
    val count = ref n
    and result = ref 1
  in
    (while !count <> 0 do
      (result := !count * !result;
       count := !count - 1);
     !result)
  end;
```

+ 99.7% of objects created when New Jersey ML, written in ML, compiles itself are immutable (Tolmach and Appel).
Chapter 3: The Editing System

If `result`, however, was an alias defined as `val result = x` where `x` was a reference object in the top-level binding environment, then the value of `x` would have to be reset on a `pop`.

3.1.3. Textual Versus Syntax-Directed Append.

Phrases appended in an editing system can be entered either as textual objects or as syntactic objects. Phrases entered as textual objects are simply entered as character strings which are parsed before analysis and compilation. A syntax-directed append would be based on a syntax-directed editor and would enter only valid phrases.

Both textual and syntax-directed editing have advantages and disadvantages (Waters 1982). Textual editing was chosen for AIMLESS, mainly because the research focus was on inter-phrase operations rather than intra-phrase operations. The existing textual entry in Cardelli’s ML was used.

3.1.4. Saving The Format Of A Textual Append.

One advantage of a textual append is that the user can format the phrase as desired on entry. The formatting is achieved by adding syntactically unnecessary white space (blanks, tabs and end of line characters) to the phrase. An editing system with a textual append should keep this format for display.

The mechanisms for input from a terminal affect the textual form. Typically terminals buffer text input till an end of line character is entered. Any white space entered in the body of the phrase is retained as part of the textual format of the appended phrase. White space before the start of a phrase, however, should be ignored. Consider,

```
- val x = 10; val y = 20;
```

In such a case the white space between the phrases (in the above example, the blank between ';' and 'v') should be ignored. The reading loop is therefore

```
skip any white space
while not end of phrase do
```
if input line is empty
then read next line
else scan current line

In ML, a phrase is terminated by either a semi-colon or the start of the next phrase (which always commences with a recognizable keyword). Thus when reading the current phrase it is possible to read the start of the next phrase. With semi-colons this problem is avoided by adjusting the lexical analyser to not read the following character as lookahead. This is possible as the semi-colon is a reserved word in ML and cannot be used as a character in an identifier. When the start of the next phrase terminates the current phrase, the ANU ML scanner has read the next token (a keyword) and the next character (to determine the end of the keyword). These must be removed from the text associated with the current phrase, and remembered as text for the next phrase. As well, any trailing white space from the current phrase must be removed from the text.

The AIMLESS commands are always terminated by a semi-colon, to avoid any look-ahead problems with nested file input.

Comments are often regarded as white space by scanners. An editing system should keep comments. Obviously comments within a phrase belong within that phrase. Comments outside a phrase may refer to either the previous or the following phrase. In AIMLESS, such comments are associated with the following phrase.

This treatment has some effects that are not immediately apparent, for example the input

\[
\begin{aligned}
\text{val } x &= 20; \\
\text{val } y &= 30;
\end{aligned}
\]

is broken up as two ML phrases, \(\text{val } x = 20;\) and \((\text{* } x \text{ is the number of widgets } \ast) \text{ val } y = 30;\), as the first phrase is terminated by the semi-colon. In this case the comment following the phrase must be included before the final semi-colon; i.e.,

\[
\begin{aligned}
\text{val } x &= 20 \quad (\text{* } x \text{ is the number of widgets } \ast); \text{ val } y = 30;
\end{aligned}
\]
val y = 30;

and the comment is now correctly associated with the definition of x. If the first phrase is not terminated by a semi-colon, as in

val x = 20 (* x is the number of widgets *)
val y = 30

the comment is part of the first phrase, as this phrase is terminated by the keyword val of the second phrase.

The problem of comments appearing between phrases occurs in any eval loop system with a textual append.

3.1.5. Dealing with Invalid Phrases.

Within a truncate/reappend editing system phrases may be invalid when appended or pushed. It is inconvenient for invalid phrases to be simply discarded, as is done by many eval loop systems. Instead they should be saved. Phrases with context-free errors can then have these amended (by intra-phrase editing), and invalid phrases with context-sensitive errors can be either amended or simply added to the script when appropriate. For example, phrase val x == 10; has a context-free error and requires intra-phrase editing. In an empty script, the phrase val x = f 5; is invalid. The phrase, however, will be valid when the script defines function f: int → 'a and then can be added to the script.

Invalid phrases are most appropriately saved on the buffer which then contains both valid and invalid phrases. The definitions of append and push become, with the phraselist \( PL = < p_1, ..., p_i | p_{i+1}, ..., p_n > \).

append(\( p, PL \)) ⇒

if \( p \) is valid, \( PL' = < p_1, ..., p_i, p | p_{i+1}, ..., p_n > \)

if \( p \) is invalid, \( PL' = < p_1, ..., p_i, p | p_{i+1}, ..., p_n > \)

push(\( PL \)) ⇒

if \( p_{i+1} \) is valid, \( PL' = < p_1, ..., p_i, p_{i+1} | p_{i+2}, ..., p_n > \)
Chapter 3: The Editing System

if $p_{i+1}$ is invalid, $PL' = <p_1, \ldots, p_i, p_{i+1}, \ldots, p_n>$

The edit operation is, with the phraselist $PL = <p_1, \ldots, p_i, p_{i+1}, \ldots, p_n>$,

$\text{edit}(PL) \Rightarrow$

if $p'$ is valid, $PL' = <p_1, \ldots, p_i, p', p_{i+2}, \ldots, p_n>$

if $p'$ is invalid, $PL' = <p_1, \ldots, p_i, p', p_{i+2}, \ldots, p_n>$

where $p' = \text{EDIT}(p_{i+1})$. EDIT($p$) is some intra-phrase edit of phrase $p$.

The EDIT operation manipulates the textual representation of the phrase at the top of the buffer and then does an append. This operation must be done by a text editor because some phrases are not representable as parse trees. Note that as the user can change the edited phrase to any text, this can be an AIMLESS command.

The discussion so far has assumed that phrases are clearly delimited units. This is not the case with a textual append; if input contains a syntactic error the evaluator cannot always recognize the end of the phrase, the evaluator simply stops reading an invalid phrase at some point. ML does not have a sufficiently simple syntax to have the end of phrase detected by the parser. It would be possible to delimit phrases by introducing a unique character as a phrase terminator. This approach was not adopted as it introduces problems with the phrase terminator in strings and comments, and causes deviation from the Standard.

Input read by the evaluator and incrementally parsed has three sources: from standard input, from a file (on a use command) and on pushing an edited phrase. Entry from standard input is broken into lines; if the incremental parser has not finished parsing the current phrase, the next line is read. Entry from a file or pushing an edited phrase is equivalent and will be treated as entry from a file; an incomplete phrase when end of file is encountered is invalid. An AIMLESS phrase consists of a sequence of characters read from input; from standard input the phrases consists of the lines read (if the AIMLESS phrase does not contain a complete AIMLESS command or ML phrase, the user is prompted for another line to add to the current AIMLESS phrase); from a file, the AIMLESS phrase
Chapter 3: The Editing System

is the whole file. The evaluator works in the cycle of reading from the current AIMLESS phrase, the next AIMLESS command or ML phrase, and evaluating it. The approach adopted was if the AIMLESS phrase is invalid (not a valid command or valid ML phrase) it is in error and placed on the buffer.

For example, with an empty phraselist, if the user enters the AIMLESS phrase `val x == 10; val z = true;` (either as one line from standard input or in any number of lines in a file), it is in error and placed on the buffer. The resulting phraselist would be `< |val x == 10; val z = true;>`. Note that `val x == 10; val z = true;` is one entry on the buffer.

For nested AIMLESS phrases (e.g., a use within a use, or a push command within an AIMLESS phrase), on an error, each AIMLESS phrase is concatenated and placed on the buffer as one phrase. The phrases are concatenated in the order in which they would have been read by the parser. Flushing input in this manner has the advantage that now the user can either edit the offending text or change existing definitions or add missing definitions, and do a push which continues loading the phrases specified by the original command.

For example, with a phraselist `< |val x == 10; val y = 22;>`, if the user enters `push; val z = true;`, when the AIMLESS phrase `val x == 10;` is pushed and is in error, it and the rest of the original AIMLESS phrase (`val z = true;`) are concatenated and placed on the buffer as one phrase. The resulting phraselist would be `< |val x == 10; val z = true; val y = 22;>`. Note that `val x == 10; val z = true;` is one entry on the buffer.

An ML phrase with a context-sensitive error could be separated from the rest of an AIMLESS phrase. For example, when appending the AIMLESS phrase `val x = y; val z = 3;`, where `y` is not defined, the definition `val x = y;` could be separated from the rest of input `val z = 3;`. These two objects could be placed as two entries on the buffer. There seemed to be no advantage in dividing the AIMLESS phrase in such cases; it must be edited or it must be pushed after other bindings are added to the environment. In either case the single AIMLESS phrase on the buffer is sufficient.

42
Chapter 3: The Editing System

It is interesting to compare this approach with other eval loop systems. Interlisp uses the simple syntax of LISP S-expressions and reads until either a constant is read or the number of left parentheses is balanced by the number of right parentheses. The Smalltalk system (Goldberg 1984) has a special menu command, doit, to read a phrase. The menu command can be viewed as a special phrase terminator. On receiving a doit command the evaluator can treat whatever is entered as the phrase. Even if the phrase contains an error, the whole phrase has been specified, and can be treated as a complete unit.

The Smalltalk doit could be used for ML. Input would not be entered line by line but instead would be entered into the system on a separate input channel which would not incrementally parse the phrase. A command issued in the eval loop would then read the current text on the input channel as the next phrase. On an error it would read to the end of this current phrase. This approach loses the interactive feel of the eval loop where the input is incrementally parsed. It would, however, avoid the complications that the normal eval loop system has with text input.

3.1.6. Exceptions And The Phraselist.

Exceptions are a fundamental construct of ML and many other high-level languages and must be dealt with appropriately in an editing system.

Top-level exceptions occur in two cases. The first is in the evaluation of a top-level expression, the other is while evaluating an expression in the course of evaluating a definition.

A top-level exception raised in evaluating a top-level expression is regarded as the observed behaviour of a test and not as an outright failure. For example, the user, on defining a function \( f \) designed to raise an exception on application to \( \text{nil} \), should test \( f \ \text{nil} \); at the top-level to see that \( f \) does raise the desired exception. Now \( f \) can be used in situations where the exception is trapped and appropriate error handling implemented. A top-level expression raising a top-level exception is placed on the script.
Chapter 3: The Editing System

A valid definition defines a set of bindings in the environment. A definition that raises a top-level exception (for example, \texttt{val x = 1 div 0;}) is a failure, no bindings are made. A definition raising a top-level exception is therefore an error. As with invalid phrases, the rest of the input is concatenated to the phrase raising the exception and the resulting text placed on the buffer.

This separation is not perfect. A top-level expression may raise a top-level exception for reasons other than as a test of behaviour; system errors, system limits and run-time interrupts may raise the exception. For example, an infinite loop \texttt{fun f x = f x;} , when tested by \texttt{f 0;} will typically be interrupted by a system interrupt and an interrupt exception will be raised. Such system exceptions seem to be more appropriately regarded as errors, with the test placed on the buffer. As at present system exceptions can be trapped by user code, then for consistency, the tests are regarded as not being in error and are placed on the script. If in the future, system exceptions are treated differently from normal exceptions, the buffer may be a more appropriate destination for tests raising system exceptions.

3.1.7. Extending The Primitive Editing Operations.

The operations \texttt{append, pop, push, del} and \texttt{edit} are primitives for an editing system that can insert and delete phrases on a script while maintaining a valid and consistent script. It would be desirable to increase the functionality of the editing system while retaining these properties.

The user should be able to pop, push or delete a series of nominated phrases with one command. Also there should be a command to replace a phrase on the script and a command to select phrases on the buffer. These operations require a method of nominating phrases in the phraselist. A phrase on the script can be nominated by any of the identifiers that it defines. The phrase \texttt{val (i,j,k) = (10,20,\texttt{true});} could be nominated by \texttt{i}, \texttt{j} or \texttt{k}. Phrases on the buffer, however, may be invalid and therefore cannot be specified in such a manner. In a system with a good graphical interface, phrase positions could be nominated by a pointing device such as a mouse. Unfortunately, pointing device technology cannot yet be
Multiple `pop`, `push` and `del` commands take an integer argument. For example, `pop k` moves the top `k` entries off the script. `push k` attempts to push the first `k` entries on the buffer, stopping if a pushed phrase is in error. For convenience, the argument `all` means all phrases on the script or buffer as appropriate. For example, `del all` would delete all phrases on the buffer. Finally if no integer argument is given the default value is 1 (i.e., `pop`, `push` or delete one phrase).

The `edit` command is also extended to do a textual edit of the first `k` phrases on the buffer.

Command `alter` takes an identifier and replaces the most recent phrase defining the identifier as a value identifier on the script with an edited version. `alter` pops phrases on the script until the specified phrase is on the buffer, does an `edit`, and then attempts to push all phrases popped by this command back onto the script. As value and type identifiers have different name spaces, two commands are necessary: `alter` for value identifiers, `altertype` for type identifiers.

Phrases on the buffer can be placed at the head of the buffer by a `move k` command that moves the `k`th phrase on the buffer to the head.

Multiple `pushes` can cause some problems with error recovery. Consider if the simple error recovery scheme of Section 3.1.5 is used when the line `push 2; val z = 3;` is entered at the terminal with the phraselist `< |val x == 10; ,val y = 20; |`. AIMLESS phrase `val x == 10;` is pushed and is in error, so `val x == 10;` and the rest of input, `val z = 3;`, are concatenated and placed on the buffer as one phrase. The resulting phraselist is `< |val x == 10; val z = 3; ,val y = 20; |`. It would be more intuitive if `val z = 3;` appeared after `val y = 20;` on the buffer as if the `push 2;` had worked, it would have been appended after `val y = 20;`. If an error occurs during a multiple push, the offending phrase, a `push` command for any remaining phrases, and the rest of input is placed on

---

+ A more literal translation of this statement would be that I still have a vt100 terminal on my desk.
the buffer as one phrase. In the above example, the resulting phraselist would be
< |val x == 10; push 1; val z = 3; val y = 20; >. In effect, a push n; is
translated into push; push (n-1); for n greater than one. This introduced push
is particularly useful if an alter command stops when a phrase it is returning to
the script is in error. The user can then fix this phrase and then phrases moved
onto the buffer by the alter command are returned to the script by the introduced
push.

3.1.8. Executing Incomplete Program Fragments.

An eval loop implementation of ML provides the ability to execute complete
program fragments. To allow top-down development, it must be possible to exe­
cute incomplete program fragments (i.e., those containing missing or invalid com­
ponents).

As only phrases on the script are executed, the execution of phrases with
invalid components contradicts the design of the script as containing only valid
phrases. Top-down development, however, does not require the ability to execute
invalid fragments, it requires the ability to have missing components in program
fragments. In ML this corresponds to having references to functions that are not
yet defined.

Missing components are usually treated as invalid by compilers. A clear dis­
tinction though can be made between the conscious decision to delay defining a
function but mentioning the function in other phrases, and having phrases anal­
ysed as invalid because of an undefined reference (through misspelling or oversight)
or any other error. A mechanism that explicitly postponed a function’s definition
would permit top-down development but still keep the script valid.

The defer command in AIMLESS takes an identifier and an optional type
specification and adds a temporary function definition for this identifier. If no type
is specified the function is given the most general type 'a -> 'b. The identifier
can then be used in later phrases which compile and can be executed. Applying
a deferred function at run time, however, generates an error and execution stops.
Deferred functions, like all other phrases on the script, can be replaced by popping them onto the buffer and using the edit command or by using the alter command.

The alternative to the defer command would be to introduce an uncatchable exception Undefined. The user could then enter

```plaintext
- fun f (_:int):bool = raise Undefined;
> val f = fn: int -> bool
```

to defer function f. This is exactly what the defer command does more conveniently, without having to introduce uncatchable exceptions into ML (it introduces uncatchable exceptions into the implementation).

The defer command is a compromise that allows top-down development by permitting incomplete program fragments to be executed but requires the user to state explicitly that a function's definition has been postponed.

Note that constants cannot be deferred. It was felt that, unlike functions, users understand constants during program development and do not need to defer them. If constant a must be deferred, it can be defined as a deferred function defer a : unit -> 'a; and a () used wherever a must appear.

3.1.9. Combining and Articulating Aggregates.

The eval loop allows for incremental development of a program as fragments can be entered and tested separately. Once the individual fragments have been tested they usually should be combined into aggregates (such as ML's let, local and abstype declarations). Aggregates exist in a language to support information hiding and its associated desirable properties (Watt 1990). For example, a datatype declaration and operations over this datatype should be combined into an abstype declaration. Conversely, when debugging it is sometimes necessary to articulate an aggregate so that "hidden" components, such as a local declaration, can be seen and tested.

The editing facilities of AIMLESS provide the basic mechanisms needed to implement the combining and articulating of aggregates. The user can manually
construct an aggregate by popping phrases onto the buffer, editing all relevant phrases, and adding the necessary syntactic sugar. The complementary operation would remove syntactic sugar. It would be more convenient for these operations to be supported by the programming environment.

Once the user has decided to build an aggregate, the major problem is determining which phrases should be combined into the aggregate. It would be convenient if the system could do this automatically. For example, given \texttt{rev'}, a tail recursive version of \texttt{reverse}, and \texttt{reverse}, a call to \texttt{rev'}

\begin{verbatim}
fun rev' [] L = L |
  rev' (a::b) L = rev' b (a::L);
fun reverse L = rev' L [];
\end{verbatim}

it would be good programming practice to construct the \texttt{local} declaration

\begin{verbatim}
local
  fun rev' [] L = L |
    rev' (a::b) L = rev' b (a::L)
in
  fun reverse L = rev' L []
end;
\end{verbatim}

A dependency analysis could make local all declarations of free variables in the exported declaration. In the above example, the free variable set of \texttt{reverse} is \{\texttt{rev'}\}. The free variable set of \texttt{rev'} is empty as \texttt{::} is a primitive operator. Such free variable dependency analysis is often, however, too general. For example

\begin{verbatim}
val a = [1,2,3,4];
reverse a;
\end{verbatim}

It may, if \texttt{a} is only used in expression \texttt{reverse a};, be desirable to make a let expression

\begin{verbatim}
let
  val a = [1,2,3,4]
in
  reverse a
end;
\end{verbatim}
In free variable dependency analysis, the definition of `reverse` would also be made into a local declaration, hiding it from the rest of the program.

In an incremental system, automatic dependency analysis cannot know what the rest of the program will access (as the rest of the program does not yet exist). Only the user can know how phrases will be used. Therefore the solution adopted was to have an interactive command `make` that asks the user to select phrases to include or not include in the aggregate.

Command `make` takes as an argument `let`, `local`, `abstype`, `assert` or `module`. Assertion construction is discussed in Chapter 5, module construction in Chapter 6. The `make` command moves back through the script, prompting the user for the desired action. For each phrase the user can either stop at this point, add this phrase to the aggregate, or not add this phrase to the aggregate (put it on the buffer instead). The aggregate is built and pushed onto the script. If it is in error for any reason (such as the user failed to include a necessary declaration) the phrase is placed on the buffer. For `let` expressions, the most recent phrase on the script is the exported expression. For `local` declarations, the user is prompted for exported declarations (as there may be more than one).

The combining of an aggregate can lead to values but not their types being exported out of an aggregate. For example, a datatype declaration and a constructor evaluate at the top-level but in a `let` expression such as

```plaintext
let
    datatype a = b | c
  in
    b
end;
```

the value `b` is exported without its type definition. Such behaviour was illegal (Milner 1984) but is now legal (Milner 1990).

The articulation operation, `unmake`, is more straight-forward. It takes an argument `let`, `local`, `abstype` or `module`. If the most recent phrase on the script is such an aggregate, it is moved onto the buffer, its syntactic sugar removed and
its component phrases pushed onto the script.

The **make** and **unmake** operations change the text of their component phrases. For example, the **make let** adds indentation to component phrases, **unmake** removes leading white space.

### 3.1.10. The AIMLESS System.

See Appendix 2 for a user’s guide to AIMLESS and Appendix 3 for an example AIMLESS session.

### 3.2. A More Formal Description of AIMLESS.

The AIMLESS operations described informally above can be (and should be) described more formally. The following description is based on the model of evaluation presented in Section 3 of the Standard ML Report (Milner 1985) and Section 8 of the formal definition of Standard ML (Milner et al 1990).

As well as the phraselist, the AIMLESS system has a binding environment sequence, a sequence of binding environments. The sequence is represented by \(< E_1, ..., E_i >\). Each binding environment is composed of a value environment, a type environment and a fixity environment. A value environment binds values to variables and to value constructors, a type environment binds generative and non-generative types to their definitions, and a fixity environment maintains fixity information of variables. \( E + E' \) denotes the operation of extending binding environment \( E \) by adding the bindings in \( E' \).

The system also has a store sequence, a sequence of stores. The sequence is represented by \(< S_1, ..., S_i >\). Each store associates values to references, which are themselves values.

The initial state of the AIMLESS system is \((< E_0 >, < S_0 >, < | >)\). \( E_0 \) is the initial ML binding environment (consisting of system libraries), \( S_0 \) the initial store and \(< | >\) the empty phraselist.

The top-level evaluation of a valid ML phrase will be denoted by the function **eval**. Function **eval** takes a valid ML phrase, a binding environment and a store.
and either returns an exception and an associated value, and a new store, or returns a new binding environment and a new store.

\[
eval(p, E, S) \implies (\text{exception}(\text{value}), S') \text{ or } (E', S').
\]

Note that when a phrase evaluates to an exception, the binding environment cannot be altered. If weak type instantiation is allowed at the top-level (see Section 3.1.2) then the definition of \( \text{eval} \) must be changed to

\[
eval(p, E, S) \implies (\text{exception}(\text{value}), E', S') \text{ or } (E', S').
\]

to allow for possibility of the environment changing with weak type instantiation.

It is possible for the store to be altered when a top-level exception is raised, for example in the evaluation of the phrase \( (x:=2; \ 1 \text{ div } 0) \); the first expression alters the store, the second raises an exception of division by zero.

The behaviour of the non-display commands of AIMLESS are now described. The base functions are defined in English, the main functions, \texttt{aim}, \texttt{do\_aim\_command} and \texttt{aim\_command}, are defined by pseudo-ML code.

An AIMLESS phrase is any textual object. Function \texttt{empty} takes an AIMLESS phrase and returns true if the phrase is empty, false otherwise.

Function \texttt{AIMLESS\_command} takes an AIMLESS phrase and returns true if it is a legal command, false otherwise.

Function \texttt{valid} takes an AIMLESS phrase and a binding environment and returns true if the phrase is a valid ML phrase in the environment, false otherwise. That is, \texttt{valid} is true if the phrase compiles, false otherwise. Function \texttt{is\_expression} takes a valid ML phrase and returns true if it is a top-level expression, false otherwise.

Function \texttt{AIMLESS\_valid} takes an AIMLESS phrase, \( p \), and a binding environment, \( E \), and returns true if either \texttt{AIMLESS\_command}(\( p \)) or \texttt{valid}(\( p, E \)), false otherwise.
Function **separate_phrase** takes an AIMLESS phrase, a binding environment \( E \) and returns two strings \( \text{first} \) and \( \text{rest} \). If the AIMLESS phrase is all white space, \( \text{first} \) and \( \text{rest} \) are empty. If the AIMLESS phrase is invalid, \( \text{first} \) is the whole phrase and \( \text{rest} \) is again empty. Otherwise \( \text{first} \) is a valid AIMLESS command or ML phrase and \( \text{rest} \) is the remainder of the AIMLESS phrase.

Status codes are defined as datatype status = ok | not_ok;

Function **aim** takes an AIMLESS phrase, a binding environment sequence, a store sequence and a phraselist, and evaluates AIMLESS commands and ML phrases in the AIMLESS phrase. An ML style definition of function **aim** is given below, followed by an English description.

\[
\text{aim}(p, ES \text{ as } <E_0, \ldots, E_i>, S \text{ as } <S_0, \ldots, S_i>, PL \text{ as } <p_1, \ldots, p_i|p_{i+1}, \ldots, p_n>) =
\]

\[
\begin{align*}
\text{let} & \\
\text{val } & (\text{first}, \text{rest}) = \text{separate_phrase}(p, E_i) \\
\text{in} & \\
\text{if } & \text{empty(first)} \text{ then } (\text{ok}, ES, S, PL) \\
& \text{else if } \text{AIMLESS_command(first)} \\
& \text{then } \text{do_aim_command(first, rest, ES, S, PL)} \\
& \text{else if } \text{not_valid(first, E_i)} \\
& \text{then } (\text{not_ok}, ES, S, <p_1, \ldots, p_i|p, p_{i+1}, \ldots, p_n>) \\
& \text{else if } \text{eval(first, E_i, S_i) } \Rightarrow (\text{exception(value)}, S') \\
& \text{then } \text{is_expression(first)} \\
& \text{then } \text{aim(rest, } <E_0, \ldots, E_i, E_i>, <S_0, \ldots, S_i, S'_i>, \\
& \text{< } p_1, \ldots, p_i, \text{first}|p_{i+1}, \ldots, p_n >) \\
& \text{else } (\text{not_ok}, ES, S, <p_1, \ldots, p_i|p, p_{i+1}, \ldots, p_n>) \\
& \text{else } (* \text{eval(first, E_i, S_i) } \Rightarrow (E', S') *) \\
& \text{(display_new_bindings(E', E_i);)} \\
& \text{aim(rest, } <E_1, \ldots, E_i, E'_i>, <S_1, \ldots, S_i, S'_i>, \\
& \text{< } p_1, \ldots, p_i, \text{first}|p_{i+1}, \ldots, p_n >))
\end{align*}
\]

Description: Function **aim** extracts the first AIMLESS command or ML phrase in the AIMLESS phrase and processes it. If it is empty no action is taken. If it is an AIMLESS command then the command is executed. If it is not a valid ML phrase, the whole AIMLESS phrase is placed on the buffer. Otherwise the ML phrase is evaluated. If an exception is raised and the ML phrase is an expression, the phrase is placed on the script and the rest of the AIMLESS phrase processed.
Chapter 3: The Editing System

If an exception is raised and the ML phrase is not an expression, the whole AIMLESS phrase is placed on the buffer. Any store side-effects done by the phrase are undone to keep the script consistent. This is in contrast to the definition of standard ML (Milner et al 1990) where in Section 8 (point 195) it is stated “An evaluation which yields an exception .. does not nullify side-effects on the state which may have occurred before the exception was raised.” Such a definition is intended to make implementing ML easier, unfortunately it spoils the model of a consistent script and so is not followed. Otherwise the ML phrase has successfully evaluated, the new bindings or the result of the phrase are displayed, the phrase is placed on the script and the rest of the AIMLESS phrase processed with the new environment and store.

Function `do_aim_command` takes an AIMLESS command, an AIMLESS phrase, an environment sequence, a store sequence and a phraselist, and evaluates the command. If the command is successful, the AIMLESS phrase is evaluated, otherwise it is appended to the phrase on the buffer (the phrase that caused the error).

```plaintext
Function `do_aim_command`:

```do_aim_command(command, rest, ES, S, P)
  as <p1, ..., pi, pi+1, ..., pn >
  =
  let
    val (status, E', S', P' as <p1, ..., pj, pj+1, ..., pm >)
    = aim_command(command, ES, S, P)
  in
    if (status=ok) then aim(rest, E', S', P')
    else (* attach rest to end of phrase on buffer *)
      (not_ok, E', S', <p1, ..., pj+1, rest, ..., pm >)
  end
```

Function `aim_command` executes the AIMLESS commands. The behaviour of commands `pop` and `push` are shown below, see Appendix 4 for a description of the complete set of AIMLESS commands.

3.2.1. Pop.

```plaintext
Function `aim_command`:

```aim_command(pop k, <E0, ..., Ek>, <S0, ..., Si>, <p1, ..., pi, pi+1, ..., pn >)
  =
  if (k=i)
    then (ok, <E0>, <S0>, <p1, ..., pi, pi+1, ..., pn >)
  else (* attach rest to end of phrase on buffer *)
      (not_ok, E', S', <p1, ..., pj+1, rest, ..., pm >)
```
Chapter 3: The Editing System

else (ok,< E₀,...,E_i-k >,< S₀,...,S_i-k >,
      < p₁,...,p_i-k |p_i-k+1,...,p_n >)

Description: The pop command either moves all the script onto the buffer or moves the top k phrases onto the buffer. The binding environment entry for each popped phrase is removed from the binding environment sequence. The store entry for each popped phrase is removed from the store sequence.

3.2.2. Push.

aim_command(push _i;ES,S,PL as < p₁,...,p_i >) = (ok,ES,S,PL);
aim_command(push 1;ES,S,< p₁,...,p_i|p_i+1,...,p_n >) =
aim(p_i+1,ES,S,< p₁,...,p_i|p_i+2,...,p_n >);
aim_command(push k;ES,S,< p₁,...,p_i|p_i+1,...,p_n >) =
aim(p_i+1; push (k-1);ES,S,< p₁,...,p_i|p_i+2,...,p_n >)

Description: The first push command clause is the case where there are no phrases on the buffer. The status code ok is returned with the existing environment sequence, store sequence and phraselist. The next clause pushes the top phrase on the buffer. The final clause handles the multiple case. The top phrase on the buffer is concatenated with push (k-1); and then evaluated. This handles the introduced push discussed in Section 3.1.7.

3.2.3. The Eval Loop.

The semantics of function separate_phrase must be slightly altered to describe the eval loop. Now function separate_phrase does not return an empty phrase for first if reading from the terminal. When reading from the terminal, separate_phrase reads lines of input until it has either a complete phrase or a phrase with a context-free error. It then separates the phrase as defined. When reading from a file (on a use command) or when a phrase is pushed or edited, separate_phrase returns empty when the phrase is finished.

The initial call is aim(nil,< E₀ >,< S₀ >,< | >), with input being read from the terminal.
3.3. A Summary Of AIMLESS.

AIMLESS overcomes many of the deficiencies of the eval loop as the base of a programming environment outlined in Chapter 1. Phrases can be edited, the text of phrases is maintained, with erroneous input placed on the buffer for later repair. Aggregates can be combined and articulated as desired. Re-testing and debugging mechanisms can be built based on the phraselit model of the editing system (see Chapter 5) and the program listing is maintained by the phraselit and can be extracted (see Chapter 6).

Phrases can now be edited - AIMLESS gives the desired functionality of an editing system for an eval loop script; phrases can be inserted, deleted and replaced. The program under development, the script, is always valid. The script is binding environment and store consistent.

The editing system presents changing a definition as the process of restoring the script to the state existing when the phrase was first entered and continuing with a new definition. The script (and hence the program) is restored to its existing state by undoing the effects of past actions when phrases are popped. The phrases popped onto the buffer to change a definition represent the rest of the script that was entered after the original definition.

A central concept in AIMLESS is undoing a phrase on a pop. The current system undoes ‘internal’ effects - bindings added to the binding environment and store side-effects. External side-effects are not undone.

It can be argued that, for completeness, external side-effects should also be undone. When a phrase that changed a file was popped, the file would be restored to its previous value. There are implementation problems with undoing external side-effects. It may not be possible to undo output, such as output to a hard copy device. The cost of saving changes to the external world may also be prohibitive. As well as these implementation problems is the basic premise that the external world is not part of AIMLESS. Internal objects are created and managed and undone within AIMLESS. External objects have a life outside of the eval loop and
so are seen as being outside the control of AIMLESS. If a side-effect on an object needs to be undone on a pop, the object must be an internal object.

AIMLESS maintains the text of phrases. This is used to display the script and buffer, and as text for both the edit command and the display commands (like whatdefn). It is also used by the program listing described in Chapter 6. The textual entry of phrases complicates the description of AIMLESS - white space must be skipped between ML phrases, an AIMLESS phrase may consist of more than one AIMLESS command or ML phrase, and trailing text must be dealt with on an error. When working with AIMLESS, however, these concerns do not affect the user unless the user chooses to have more than one command or ML phrase on an input line. The textual entry gives the advantage of the user formatting the phrase as desired. The entry of phrases terminated by the following phrase is correctly handled.

Input errors are placed on the buffer and can be edited. This is a particularly convenient feature, and one missed when using other eval loop systems. On an error in a system where invalid phrases are not available to be edited, the user must re-type the phrase, or cut and paste parts of the phrase while editing the phrase, or maintain a file which is edited and then paste the new phrase into the eval loop. In AIMLESS, if the invalid phrase must be edited, the user simply enters edit; and can change the text. If the invalid phrase refers to identifiers not yet defined, these can be added or deferred and the phrase pushed.

Aggregates can be built using the make command and decomposed using the unmake command. Programs can then be built up from tested components that are packaged appropriately to maximize information hiding and data abstraction. The development of programs from tested components is encouraged by the make command. The unmake command gives the user the ability to open up an aggregate to test components. This also encourages the building of programs by components. If the user knows that it is simple and convenient to articulate an aggregate, then the user will not wait until the final stages of program development to combine aggregates; aggregates can be constructed as soon as their
components are available. To give the user complete control over selection of component phrases in a construction, the system asks the user for the desired action phrase on the script. Clearly there exists potential for automating part, if not all, of this process. The textual formatting of aggregates from components also presents opportunities for more flexible mechanisms.

A weakness of AIMLESS is the amount of re-evaluation required when a phrase ‘early’ in the phraselist is altered. All later phrases must be re-evaluated when pushed, degrading the responsiveness of AIMLESS. Phrases on the script, however, may not be affected by a change, simply their position on the script requires them to be re-evaluated. Section 4.4 outlines techniques that could be implemented to reduce this unnecessary cost.
Chapter Four: Implementing The Editing System

Chapter 4 describes adding AIMLESS to ANU ML. The chapter consists of four sections. The process of evaluation in ANU ML is briefly outlined in section one. Section two describes the ANU ML architecture relevant to implementing the editing system. The next section is the core of the chapter and describes the implementation of the phraselist. The last section then sketches some techniques for reducing the cost of re-evaluation in AIMLESS.

While the description of this implementation of AIMLESS is based on ANU ML, most discussion is of wider interest; applying generally to either eval loop systems or ML implementations. Where the implementation of AIMLESS relies on aspects of the design of ANU ML that are not found in other ML systems, alternative approaches are outlined.

As ANU ML is implemented in Pascal and C, AIMLESS is also written in these languages. The code outlines given in the chapter are presented in a pseudo-Pascal and reflect the actual implementation.

4.1. Evaluation In ANU ML.

Valid ML phrases entered in the eval loop are compiled and then executed. Compilation involves three phases: parsing, analysis and code generation. Parsing takes the text of the current phrase and a fixity environment and produces an abstract syntax tree (AST) and alters the fixity environment if appropriate.

Analysis takes the AST and the current binding environment, performs type-checking, and produces an expanded AST and a new binding environment with the bindings defined by the phrase appended to the current environment. The expanded AST has any wildcards in records expanded and resolves overloaded operators. For example, with the definitions

```pascal
- type person = {name: string, age: int};

- fun select_name ({name=x,...}:person) = x;
```

analysis effectively expands the definition of select_name to
fun select_name ({age=_, name=x}: person) = x;

The code generator works in two stages. It uses the expanded AST and the current binding environment (to calculate the position of identifiers) to generate intermediate Functional Abstract Machine (FAM) code (Cardelli 1983a) which is assembled into the object code of the particular target machine. This object code is then executed. Note that value definitions and function definitions produce code that is executed to produce the value of the binding. The evaluation process is schematically shown in Figure 4.1.

4.2. The ANU ML Architecture.

Parts of the ANU ML architecture relevant to implementing AIMLESS are described. First, the abstract machine and the binding environment are outlined. Then some modifications necessary to the implementation of the eval loop needed by AIMLESS are presented.

4.2.1. The FAM And The Binding Environment.

Parts of the FAM relevant to implementing the editing system are discussed. See Cardelli (1983a) for a complete description.

The FAM has a data heap, storing structured and variable length data objects. The data objects are stored in cells. The cells have various formats, such as record cells, variant cells, reference cells and closure cells. Record cells are n-tuples of cells. Variant cells are tagged cells, with a tag field and a field for the body, used to represent disjoint union types. Reference cells have a field for the value of reference objects. Closure cells are function cells consisting of a pointer to a text cell (the body of the function) and a set of cells for global variables of the text. Each field of every cell in the heap contains either a constant or a pointer to another cell in the heap. Constants are limited in range so they can be distinguished from pointers.

The data heap is partitioned into data pages. Each data page contains cells of one particular format. The format information of each data page is used by the garbage collector.
Fig. 4.1 Evaluation in ANU ML: The data structures are shown in square boxes, the evaluation phases in rounded boxes. The result of the evaluation is the final value, the new fixity environment, the new binding environment and the new store.

The FAM has an argument stack. FAM data objects are stored on the argument stack as either constant or pointer values. The argument stack stores all
Chapter 4: Implementation

temporary values as well as values bound in the top-level environment. The argument pointer, AP, points to the top of the argument stack. The environment pointer, EP, also points to the argument stack but points to the top of the values bound in the top-level environment. Except during execution, the argument and environment pointers point to the top of the argument stack. During execution, the argument pointer will change as temporary values are used. See Figure 4.2 for a schematic view of the argument stack.

![Argument Stack Diagram]

Fig. 4.2 The Argument Stack: Temporary values are placed on the top of the stack and are pointed to by AP. Values bound in the top-level binding environment are pointed to by EP.

Consider the evaluation of a phrase of the form val <ide> = <expression>. To evaluate the phrase, <expression> must be executed. Temporary values will be added to and removed from the top of the argument stack as <expression> is executed. Once executed, the remaining value above EP on the stack is the value to be bound to <ide>. This is reflected by moving EP to point to the top of the stack (to AP).

The FAM contains no mechanism for storing the type or scope information necessary for analysis. The top-level binding environment stores value, function and type information on identifiers bound by top-level definitions. This information is used during analysis to determine typing and scoping. The environment
is represented as a linked list of values. As new identifiers are bound by top-level
definitions they are added to the top of the environment.

Representing the top-level environment as a linked list is clearly inefficient. As the size of the environment grows, so does the average search time for an identifier (searching for a primitive requires searching through the entire linked list environment). Most ML systems use a hash table (or series of hash tables) to represent the environment so that the average search time for an identifier is constant. ANU ML should implement hash tables to improve its performance that degrades badly as the environment grows.

The binding environment and the argument stack are closely related. The binding environment contains information on value, function and type definitions for use at analyse time, while the argument stack contains the values of the value and function definitions for use at run time. The binding environment and the argument stack below the EP are therefore synchronized. The $n^{th}$ most recent entry in the binding environment (ignoring type definitions) appears at an offset of $n$ from the EP in the argument stack. See Figure 4.3 for an example.

The information on the position of identifiers used by the code generator is a temporary binding environment (a copy of the current top-level binding environment) that is modified to keep track of temporary values on the stack during evaluation.

4.2.2. Modifying The Eval Loop.

To implement AIMLESS the reading of input and handling of exceptions in the eval loop had to be altered.

4.2.2.1. Reading The Current Phrase.

The reading of input by the evaluator is the most immediate experience the user has with a programming environment. Because of this direct impact, reading input cannot be dismissed as a minor issue but must be well engineered. Problems with textual append, push and edit commands have been discussed in Sections
Phraselist =

<val x = 101, type ('a,'b) myrecord = {first:'a, second:'b},
val y = true, val z = 'a'>

Fig. 4.3 Schematic view of argument stack and binding environment:
The binding for z appears at offset 0 from the top of the argument stack and
the binding environment. The binding for x appears at offset 2 from the top of
the argument stack and the binding environment (ignoring the type definition of
myrecord).

3.1.4 and 3.1.5.

Input is entered either by standard input or by reading a file with the use
command. A file stack stores the stack of input files corresponding to nested uses;
an empty file stack indicates that input is to be read from standard input. A call
to use a file has that file’s descriptor put on the file stack and read till end of file
or an error. On an error the file stack is reset to empty and the current input line
flushed.

Scanning in ANU ML is done using routine ScanChar to read the next charac­
ter. ScanChar returns the next character from the appropriate input. It also
prints the system prompt when reading from standard input and the last input line has just been exhausted.

Having standard input for entering phrases and as input for other operations can cause problems in an eval loop system. Consider the AIMLESS phrase
\[
\text{input}(\text{std.in},3); \text{val } x = 20;\]
entered as standard input. On executing the \texttt{input} function, part of the rest of the phrase would be read. Consider the phrase
\[
\text{edit}; \text{val } x = 10;\]
entered on standard input. On editing the phrase on top of the buffer the rest of standard input, \texttt{val } x = 10; is taken as input to the editor.

To avoid the overloaded use of standard input for entering AIMLESS phrases and as input for AIMLESS commands and ML phrases, an internal buffer was introduced. Now \texttt{ScanChar}, if it has finished the last line, reads the next line into the internal buffer. \texttt{ScanChar} then reads the internal buffer as required till end of line. Standard input can be used by the \texttt{input} function or the editor without affecting the entered phrase.

To accumulate the text of a phrase, as each character is read by \texttt{ScanChar} it is placed in data structure \texttt{PhraseInput}. The text saved in \texttt{PhraseInput} is later moved onto the phraselist. White space before the start of a phrase is ignored.

As discussed in Section 3.1.4 any look-ahead used to determine the end of a phrase must not be included as the text of the phrase. If look-ahead was required (no semi-colon terminating the phrase) then the look-ahead text was removed from \texttt{PhraseInput} and stored as the start of the text for the next phrase. AIMLESS commands are always terminated by a semi-colon, avoiding the problem of nested input look-ahead.

On recognizing an error, either a syntax error or a definition raising an exception, the AIMLESS phrase must be placed on the buffer. Each file on the file stack is flushed into \texttt{PhraseInput} in turn. Next, the internal buffer is flushed into \texttt{PhraseInput}. If any AIMLESS phrase has look-ahead, this must precede its remaining text. Similarly if an AIMLESS phrase is in the midst of a multiple \texttt{push} then an introduced \texttt{push} is inserted before the rest of its text. \texttt{PhraseInput} is then moved onto the buffer. This scheme obeys the method of flushing described
4.2.2.2. Handling Exceptions.

Cardelli’s ML system had a simple mechanism for handling top-level (i.e., untrapped) exceptions. The exception was displayed and the system reset. The existing system state was discarded as it was not required. AIMLESS, however, needs top-level exceptions to be handled more gracefully. More specifically, AIMLESS needs to be able to raise a top-level exception but continue from the existing system state. For example, on doing a multiple \textbf{push}, if the first phrase is an expression such as \texttt{hd \{\}}; which raises a top-level exception, AIMLESS must deal with the exception and then continue with the \textbf{push}.

A routine \texttt{ExecuteOnce} was built that evaluated the next AIMLESS command or ML phrase (using \texttt{ScanChar} and the file stack to determine input) and returned a status code. The codes are \textit{successful evaluation}, \textit{soft exception}, \textit{hard exception} and \textit{pre-runtime error}. A \textit{soft exception} is a top-level exception raised by an expression (and hence regarded as valid) while a \textit{hard exception} is a top-level exception raised in a definition (and hence regarded as invalid). A \textit{pre-runtime error} is a syntax or type error.

Now AIMLESS calls \texttt{ExecuteOnce} to evaluate the next phrase. As it is read, the text is stored in \texttt{Phraselnput}. If the phrase evaluates it is placed on the script, and status \textit{successful evaluation} returned. If the phrase is not valid, the input is flushed, the phrase is placed on the buffer, control returns to \texttt{ExecuteOnce} with status \textit{pre-runtime error}. If the phrase raises a top-level exception, the phrase is placed on the script (\textit{soft exception}) or the phrase with flushed input is placed on the buffer (\textit{hard exception}), and control then returns to \texttt{ExecuteOnce} with the status code.

4.3. Implementing The Phraselist.

The script and the buffer are the primary data structures of AIMLESS. The two are represented as separate linked lists, each of type \texttt{PhraseEntryRefT}. Type \texttt{PhraseEntryRefT} is defined as

\begin{verbatim}
Chapter 4: Implementation

in Section 3.1.5.

4.2.2.2. Handling Exceptions.

Cardelli’s ML system had a simple mechanism for handling top-level (i.e., untrapped) exceptions. The exception was displayed and the system reset. The existing system state was discarded as it was not required. AIMLESS, however, needs top-level exceptions to be handled more gracefully. More specifically, AIMLESS needs to be able to raise a top-level exception but continue from the existing system state. For example, on doing a multiple \textbf{push}, if the first phrase is an expression such as \texttt{hd \{\}}; which raises a top-level exception, AIMLESS must deal with the exception and then continue with the \textbf{push}.

A routine \texttt{ExecuteOnce} was built that evaluated the next AIMLESS command or ML phrase (using \texttt{ScanChar} and the file stack to determine input) and returned a status code. The codes are \textit{successful evaluation}, \textit{soft exception}, \textit{hard exception} and \textit{pre-runtime error}. A \textit{soft exception} is a top-level exception raised by an expression (and hence regarded as valid) while a \textit{hard exception} is a top-level exception raised in a definition (and hence regarded as invalid). A \textit{pre-runtime error} is a syntax or type error.

Now AIMLESS calls \texttt{ExecuteOnce} to evaluate the next phrase. As it is read, the text is stored in \texttt{Phraselnput}. If the phrase evaluates it is placed on the script, and status \textit{successful evaluation} returned. If the phrase is not valid, the input is flushed, the phrase is placed on the buffer, control returns to \texttt{ExecuteOnce} with status \textit{pre-runtime error}. If the phrase raises a top-level exception, the phrase is placed on the script (\textit{soft exception}) or the phrase with flushed input is placed on the buffer (\textit{hard exception}), and control then returns to \texttt{ExecuteOnce} with the status code.

4.3. Implementing The Phraselist.

The script and the buffer are the primary data structures of AIMLESS. The two are represented as separate linked lists, each of type \texttt{PhraseEntryRefT}. Type \texttt{PhraseEntryRefT} is defined as

\end{verbatim}
Rest and Prev make the structure a doubly linked list for easy manipulation.

Str stores the raw textual form of the phrase.

Body stores the abstract syntax tree representation of the phrase. The AST is not strictly necessary as it can always be generated from the raw textual form. It is kept to reduce the cost of reparsing when it is required (e.g., for determining if a phrase is an expression, for tracing operations discussed in Chapter 5).

OldFixityEnv points to the fixity environment existing at the time that the phrase was parsed.

IsItExpression is true if the phrase defines identifier it, either as a top-level expression or as a definition.

DependsOnIt is true if identifier it is a free variable in the phrase.

HowMany and ArrayHowMany are the number of store side-effects that the phrase caused.

AggregateFlag is used in aggregate construction to indicate phrases in the aggregate (and whether local or exported) or not in the aggregate.

StartScript points to the start of the script, Script to the top of the script, where new phrases are added. Buffer points to the head of the buffer where new phrases are added, EndBuffer points to the end of the buffer.
Chapter 4: Implementation

The fields of the script and the buffer are explained by describing the operations that they support.

4.3.1. Undoing The Binding Environment.

To undo the binding environment, the entries on the binding environment added by each phrase need to be known. A phrase on the script may add more than one entry on the top-level binding environment. For example phrase `val x = 1 and y = 2;` will add two entries to the binding environment, one for `x` and one for `y`. To synchronize the binding environment and the script an additional pointer, `Phrase`, was added to each entry in the binding environment. Each entry points to the phrase defining it. A schematic view of the top-level binding environment and the script is shown in Figure 4.4. The binding environment sequence abstraction used in Section 3.2 is therefore implemented by a single binding environment with place holders indicating which bindings correspond to which phrase.

---

Fig. 4.4 View of top-level binding environment and script: The phrase `val x = 1 and y = 2;` defines two entries on the binding environment. The script and the top-level binding environment are stacks; items are added to and removed from the top of both structures.
Chapter 4: Implementation

With this information, undoing the binding environment of the top phrase on the script is simple - all entries on the binding environment pointing to the phrase are removed.

If the top-level environment is implemented as a hash table, each phrase on the phraselist would have to keep track of all entries it added to the hash table (it could keep an explicit pointer to each entry in the table, or re-calculate each entry as necessary). To undo these, it would be necessary to remove them from the hash table. In a hash table with overflow chaining this is straightforward. It is also simple in a hash table with collision chaining as deletes occur in the reverse order to inserts (if the deletes in each individual phrase are done in reverse order to their inserts).

4.3.2. Undoing The Fixity Environment.

When a fixity directive is popped from the script, its effect on the fixity environment must be undone. Consider,

- fun myadd (x,y) = (x:int) + y;
- infix myadd;
- 3 rnyadd 4;
- pop 2;

Function myadd must now have a fixity of nonfix and must be used in prefix form, as in myadd(3,4).

The ANU ML system maintains a fixity environment for use by the lexical analyser to determine the fixity, precedence and associativity of identifiers. To undo the most recent fixity directive, the corresponding most recent entry on the fixity environment must be removed and the fixity of the identifier restored to its previous value. This mechanism was used by the parser to undo fixities defined in scope, for example an infix definition in a let expression.

To undo fixities on a pop, each phrase on the script maintains OldFixityEnv, a pointer to the fixity environment existing when the phrase is entered. When a phrase is popped, to undo the fixity environment, all entries on the fixity envi-
environment above OldFixityEnv are removed and all specified identifiers have their fixity values restored.

4.3.3. Undoing Weak Type Instantiations.

As mentioned in Section 3.1.2, weak type instantiations have become illegal at the top level in ML, and so they no longer have to be undone. This section explains how it was once done and thus how it could be done in similar situations.

Undoing weak type instantiations is very similar to undoing fixity directives. The information required was not maintained by the ANU ML system and so had to be added.

A list of top-level weak types, WeakObjectList, is kept. When a phrase successfully evaluates, it maintains a pointer, WeakObjectPtr, to the existing WeakObjectList. This list is then scanned to see if any of the uninstantiated weak types were instantiated by this phrase. Any weak types now instantiated are recorded on the phrase’s ChangedWeak list. Next, any uninstantiated weak types in the types of the entries that the phrase adds to the top-level binding environment are added to WeakObjectList.

Undoing the weak type instantiations is simple. All weak types on the phrase’s ChangedWeak list are reset to uninstantiated. The WeakObjectList is set to the value of the phrase’s WeakObjectPtr.

4.3.4. Undoing Store Side-effects.

There are two data objects in ANU ML that can be side-effected: reference objects and arrays. Reference objects are side-effected by the assignment operator. Arrays are not standard (but widely supported) and can be thought of as an efficient implementation of an abstract data type that can be defined in standard ML. Arrays are side-effected by the update operator.

The top-level store side-effects of a phrase must be undone when the phrase is popped. For example, with the phraselist <val x = ref 0; x := 10; | >, a pop moves phrase x := 10; on to the buffer and must also undo the assignment.
so that \( !x \) (the de-referenced value of \( x \)) is 0. Note that side-effects to temporary reference objects (Section 3.1.2) do not have to be undone.

The method used was to save changes made to the store by each phrase. When a phrase is then popped, the changes are undone. The store sequence abstraction of Section 3.2 is therefore implemented by a single store with change information. Alternative approaches exist. For example, the store could have checkpoints and to undo a store side-effect, the store would be set back to the most recent checkpoint and the script would be played forward up to the desired point. On the assumption that there are not many reference objects in an ML program (0.3% of objects produced when New Jersey ML compiles can be side-effected (Tolmach and Appel 1990)) and because an arbitrary expression can take arbitrarily long to evaluate, the saving changes method was preferred. See (Archer et al 1984) for a discussion of undoing side-effects in a system.

4.3.4.1. Undoing Reference Objects.

The basic assumptions in designing the reference object undo mechanism were that because of ML's functional character there would not be many reference objects and that assignment, when used, would be used for efficiency and should not be slowed down. This meant that it is preferable to check relevant reference objects after each phrase was evaluated rather than slow down assignment by making it raise a flag whenever a side-effect occurred. As assignment is one machine instruction, making it raise a flag at run time would slow it down by at least a factor of three (load one, store in flag, do assignment).

The FAM reference cell was extended from being just a pointer field, \( \text{At} \), to having a previous value, \( \text{Previous} \). \( \text{At} \) is the current value of the reference object, \( \text{Previous} \) the value of the reference object before execution of the current phrase. If, after execution of a phrase, the \( \text{At} \) and \( \text{Previous} \) fields of a reference object are different then the phrase side-effected this object.

A new FAM data structure, \( \text{ChangedStack} \), was added to save store changes. \( \text{ChangedStack} \) is a stack of \((\text{Where}, \text{What})\) pairs. \( \text{Where} \) is the location of a changed
reference object and What its previous value.

The basic loop is simple. Before a phrase is executed all At fields are equal to their Previous fields. The phrase is executed. If the phrase is valid (is added to the script) all reference objects are checked to find ones where At does not equal Previous. If At does not equal Previous then this reference object has been side-effected by the phrase. The (position,Previous) pair is placed on ChangedStack and Previous is set to At. The number of changed reference objects placed on ChangedStack is stored in the HowMany field of the phrase (of type PhraseEntry) on the script.

If the phrase is invalid (is added to the buffer) then all reference objects have their At fields set to Previous. This correctly undoes a phrase such as val x = (y := 10; 1 div 0); which causes a side-effect (to y) that must be undone.

When adding a new reference object it is tempting to initialize Previous to the new value and continue execution. Such a scheme, however, would result in side-effects to temporary reference objects being stored as changes, unnecessarily using up space on ChangedStack and in heap space. To avoid this problem NewRefList stores all declarations made by a phrase. Before reference objects are checked for changes, all reference objects on NewRefList have their Previous set to At and are not seen as changes.

When a phrase is popped, the number of side-effects caused by the phrase (HowMany) are popped off ChangedStack and the Where locations reset to their What values. The store is now the same as when the phrase was originally evaluated.

All references are checked after each phrase is evaluated. Temporary references are garbage collected periodically and so do not accumulate. It would be possible to examine only top-level references for changes, but at the extra cost required to determine the top-level references for each phrase.
4.3.4.2. Undoing Arrays.

Undoing arrays is similar to undoing reference objects except that the basic design assumptions are different. First, arrays are typically large, so even if there were only a few arrays there would be many elements to compare. Second, raising a flag on an array update operation is not a large overhead, adding one C instruction to the ten already required to calculate array positions and insert the new value. So, rather than searching all array elements for changed elements between phrases, the update operation raises a flag indicating that the array has a changed element, and between phrases, an array is only checked if its flag indicates that an element has changed.

The FAM array cell changed from (Lowerbound, Size, Array elements) to (Lowerbound, Size, Update, Previous, Array elements). Update is the flag indicating that the array has an element changed by the current phrase. The update operation sets the flag to true whenever it side-effects an array. Because of array sizes, instead of storing the Previous field for each element in the array, the array has a pointer to an array of Previous values.

The individual element changes are stored on the ArrayChangedStack and field ArrayHowMany on the script is the number of array elements changed by the phrase. Thus if a large array has one element changed, only that one element is kept as a change.

It is difficult to provide experimental evidence to support the decisions taken for undoing side-effects. The operation of finding and saving changed references is fast and is unlikely to effect the performance of the system. The evaluation of quite small ML phrases, such as 3 + 4; requires the execution of thousands of Pascal and C statements (which translates to tens of thousands of machine instructions). On the other hand, checking 1000 references (a huge number given ML's functional character) would take 3000 C instructions in a tight loop (which would translate directly into machine code instructions).

See Appendix 5 for the implementation of the individual AIMLESS editing
4.4. Reduced Re-evaluation In AIMLESS.

In AIMLESS each push operation notionally involves re-evaluation of the pushed phrase. The re-evaluation consists of compilation and execution and is described as an apparent operation as in many cases some, or even all, of the re-evaluation need not be repeated. Phrases popped onto the buffer could store the effect of re-evaluation and, in appropriate circumstances, use these on being pushed. Consider the phraselist `<fun g x = x + 1; fun f x = x + 2; | >`. If function g is altered, function f is popped and then pushed. When pushed, f does not need to be re-analysed as g is not used by f, and indeed the closure bound to f can be re-used, avoiding any re-evaluation cost.

The re-evaluation algorithms have a strong bearing on the performance of AIMLESS as a programming environment. Unnecessary re-evaluation represents a decrease in the responsiveness of the system between editing and being able to execute.

4.4.1. Compilation Costs.

Compilation costs for ML differ from more traditional languages where parsing, particularly lexical analysis, usually dominates compilation costs (Waite 1986). Tests were done on ANU ML using files written to check the compiler for correctness. These files consisted of short phrases with a wide range of constructs. The tests indicated the breakdown of costs as parsing (11%), analysis (62%) and code generation (27%). The code generation further broke down to 3% for generating the intermediate FAM code and 24% for assembling the object code from that. The higher than usual cost for analysis reflects the additional work done by an inferencing typechecker and the inherent inefficiency of a non-flat binding environment (where the time to access a primitive identifier is proportional to the number of entries in the environment).

4.4.2. Re-evaluation And Dependencies.

When a phrase is pushed the amount of re-evaluation that is necessary de-
Chapter 4: Implementation

Depends on changes within the system since the phrase was originally evaluated. The changes within the system range over the fixity environment, the binding environment, locations on the argument stack and the store. The dependencies are summarized in Table 4.1 and should be compared to Figure 4.1.

<table>
<thead>
<tr>
<th>phase</th>
<th>data structure produced</th>
<th>depends on</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsing</td>
<td>AST fixity environment</td>
<td>fixity environment</td>
</tr>
<tr>
<td>analysis</td>
<td>expanded AST binding environment</td>
<td>types</td>
</tr>
<tr>
<td>code generation</td>
<td>code</td>
<td>locations in stack overloaded operators</td>
</tr>
<tr>
<td>execution</td>
<td>value store</td>
<td>values, store</td>
</tr>
</tbody>
</table>

Table 4.1 The phase, data structure and dependency relationships: For example, parsing produces an AST and a fixity environment using information in the fixity environment.

Changes within the system affect the free variables of a phrase. A free variable (one not bound in the phrase) refers to a binding in the top-level binding environment defined by either an earlier phrase or the initial binding environment. The free variables in a phrase consist of both value identifiers and type identifiers. For example, the phrase `val (x:int) = (f 10) + ((g:a) 7);` would have free value identifiers `{f,g}` and free type identifiers `{int,a}`. As the value and type namespaces are separate the two free variable lists must be separate.

If a phrase avoids a phase of re-evaluation it must still have the same effect on the system. For example, a phrase that avoids parsing must still produce an abstract syntax tree and change any fixities as would happen if the phrase was
Chapter 4: Implementation

It is worth noting some peculiarities of ANU ML at this point. Firstly, ANU ML parses constructors as identifiers and alters the abstract syntax tree during the analysis phase. Other systems recognize constructors during the parse phase. In such systems, a change in constructor status would alter the abstract syntax tree produced. Secondly, FAM code uses stack offsets to access variables in the outer-level environment. The FAM could treat these variables as literals and compile them directly into the code. The code generation phase would not then depend on locations on the stack, but on the values of the variables themselves.

A phrase need not be re-parsed on a push if the fixity environment values of the free variables in the phrase are unchanged from when the phrase was originally parsed. Only value identifiers can have their fixity properties changed, so type identifiers need not be considered.

Consider

- \texttt{fun \ f (x,y) = (x:int) + y};
- \texttt{f (3,5)};
- \texttt{pop};
- \texttt{infix f};
- \texttt{push};

Here phrase \texttt{f (3,5)}; must be re-parsed, as \texttt{f} has changed from nonfix to infix. Note changes of precedence and associativity are changes to the fixity environment.

The big saving on re-evaluation costs occurs when the types of the free variables of the pushed phrase are unchanged. Consider the phraselist \texttt{<fun f x = x + 1; fun g x = (f x) * 2;>}. Now if \texttt{f} is altered to \texttt{fun f x = x + 2}; function \texttt{g}'s type will remain as \texttt{int \rightarrow int} as the free value identifier \texttt{f} has the same type and the free type identifier \texttt{int} has not changed. Phrases popped would need to store the information added to the binding environment by the analysis phase and the information added to expand the abstract syntax tree.

The Glide project included work on reducing the cost of re-typechecking for polymorphically typed languages similar to ML. This allowed functions to be en-
tered and altered in any order while retaining outer-level static scoping. As the environment was interpretive, the main cost of re-evaluation when a definition was changed was the typechecking of definitions depending on the changed definition. Although the Glide environment is quite different to that of AIMLESS, the goal of only having to re-typecheck dependent definitions is common.

Type identifiers are not used by the typechecker by name but by their timestamp, a unique integer assigned by the analyser. Phrases with type definitions popped must therefore retain their previous timestamps, and on pushing, if the type definition has not changed, then the type identifier must re-use its old timestamp. Otherwise the system would allocate a new timestamp when the type definition was pushed and all type identifiers pushed would be regarded as changes.

As the FAM code uses stack offsets the position of the free value identifiers in a phrase must be the same to avoid re-generating code on a push. If the FAM treated free variables as literals, then the height of the stack would not affect the FAM code. Overloaded operators must also have the same type. For example, if function f in the phrase val x = (f 10) + (f 10); is changed from \text{int} \rightarrow \text{int} to \text{int} \rightarrow \text{real} the code for x must be changed from an integer addition to a real addition. To avoid re-generating code, the final object code must be stored with each popped phrase.

Finally some phrases need not be executed on a push. A function definition phrase has the same closure if its free value variables (the globals of the closure) have not changed. Both top-level expressions and value definitions need not be re-executed if all free variables have the same value and the phrase contains no side-effects. If the expression contains store side-effects then these must be simulated. If the expression contains external side-effects then the expression should be re-executed as external side-effects are regarded as being outside the control of AIMLESS.

Note that most ML systems, but not ANU ML, use generational garbage collectors. Such garbage collectors make assumptions about the order of references within objects on the heap, newer objects can only pointer to older ones. If objects
in the run-time system can be changed so that this ordering assumption no longer holds, then these objects must be treated in the same manner as reference objects (Matthews 1988b).

The next sections explore some schemes for reducing re-evaluation costs using the basic dependency ideas presented in this section.

### 4.4.3. Avoiding Re-parsing.

Re-parsing a pushed phrase can be avoided by using the AST generated when the phrase was parsed. That AST can be re-used if none of the identifiers in the phrase have had their fixity changed. This improvement has been implemented.

The fixity environment before the phrase was last parsed is already stored on the script as OldFixityEnv so that fixities could be undone on a pop. On a push, each free value variable in the phrase, is compared in the current fixity environment and OldFixityEnv. If each has the same fixity properties the AST can be re-used.

If the AST is re-used, any changes to the fixity environment that the phrase should make (when being parsed) must be performed. To do this, additions to the fixity environment by a phrase when it is originally parsed are stored (as the fixity environment entries between the previous fixity environment, OldFixityEnv, and the new fixity environment, NewFixityEnv). When the phrase is popped its changes to the fixity environment are undone as previously. When the phrase is pushed (and its AST re-used) the current fixity environment is adjusted by adding the fixity changes of the phrase.

It was assumed that fixity information will generally be the default (prefix) and therefore searching through fixity environments will be quick. If this assumption was not seen to hold, each free value variable’s fixity could be stored explicitly to avoid look-ups.

When an AST is saved on the buffer, a check must be made to ensure that the AST and the text correspond. The text will not correspond with the AST
on an append (or a push) if the phrase fails and remaining input text is flushed onto the end of the failed phrase’s text. For example, consider the case where phrase \texttt{val x = y; val z = 20;} is appended to an empty script. The \texttt{val x = y;} phrase is read, and an AST generated, but as \texttt{y} is not defined, the phrase fails. The phrase and all remaining input text is flushed and placed on the buffer. The text of the phrase on the buffer is therefore \texttt{val x = y; val z = 20;} but the AST represents \texttt{val x = y;}: The AST does not correspond to the text and cannot be re-used. This problem is simply detected and handled; if any text is flushed on a phrase being put on the buffer, the AST is not saved.

The AST stored on the phraselist is not actually the AST but the expanded version after analysis. The expanded AST cannot always be re-used. For instance

- \texttt{type person = \{address:string, age:int, name:string\};}

- \texttt{fun select_age = ({age=x, \ldots }:person) = x;}

After analysis, the expanded AST will be

- \texttt{fun select_age = ({address=_, age=x, name=_}:person) = x;}

If the definition of \texttt{person} is changed to

- \texttt{type person = \{age:int, name:string\};}

then the expanded version cannot be re-used. A similar problem occurs when an overloaded operator is resolved. There are two solutions to this problem. Either the phraselist could keep a copy of the unexpanded AST, or expanded ASTs could be noted and not re-used. As most ASTs will not be expanded, the second option was used, avoiding the need to make a copy when putting the AST on the phraselist.

While re-using the AST reduces the re-evaluation costs when pushing a phrase, the cost of saving the AST is expensive (on simple tests the memory usage of the system increased by 25%), and therefore may be prohibitive for a system with tight memory limits.
4.4.4. Implementing Re-evaluation Ideas.

Although parsing costs are avoided for re-evaluation, the real costs in re-compilation are in analysis and code generation. As well, the re-execution of expressions may be computationally expensive. Techniques for avoiding these costs have not been implemented and so are only sketched out and briefly discussed. The techniques were not implemented as they are not essential to AIMLESS; they do not alter its functionality and while they will improve the responsiveness of AIMLESS, this has not been found to be unacceptable in practice.

When implementing techniques to avoid re-evaluation a balance must be kept between the cost to analyse and to store information to avoid re-evaluation against the cost of the re-evaluation. The costs depend on how the system is used and many unknowns exist when analysing the performance of a programming system. These factors include the experience of the programmers using the system (their experience - with the system, with programming generally, and with the language used), the domain of problems being solved (their size and complexity) and the style of development (prototyping vs structured development). These in turn prompt other questions: what is the average position of the phrase on the phraselist to be edited, how many dependencies has an average edited phrase, and what sort of change is an average edit - deleting a phrase, inserting a phrase, or altering a phrase? And if an alteration, what sort of alteration - minor (e.g., changing a comment), a change in value, or a change in type? No analysis has been performed on AIMLESS to determine these factors that determine the balance in trade-offs for avoiding re-evaluation. Such analysis should be performed before the techniques described below are implemented.

Two algorithms are given, and then some further ideas. To simplify the discussion changed type definitions will not be considered. To include changed types the machinery for changed identifiers must be duplicated.

Only popped phrases on the buffer can reduce re-evaluation costs when eventually pushed. If a phrase is on the buffer because of an error, pushing it has the same effect in the algorithms below as appending a new phrase.
4.4.4.1. Algorithm 1.

Let Changed be a set of identifiers. The basic idea of the algorithm is that all identifiers changed by a pop, push or append are added to Changed, and if a pushed phrase has a free identifier in Changed then it must be re-evaluated.

Popped phrases store a Defines set, a Frees set and BindHeight. Defines is the set of identifiers defined by the phrase, Frees is the set of free identifiers in the phrase, and BindHeight is the height of the argument stack when the phrase was originally evaluated. As well, the phrase must store binding environment information, consisting of the bindings added to the environment and any weak type information altered by the phrase. The code to be executed and the value(s) produced by the code are also stored.

When a phrase is popped, Changed has Defines added to it. The phrase stores all the information described above.

On an append (or the push of a phrase on the buffer because of an error), if the buffer is not empty, the identifiers defined by the phrase (if it is valid) are added to Changed. If the buffer is empty, no phrases on the buffer are affected by the append.

On a push, if one of Frees is in Changed then the phrase must be re-evaluated and Defines added to Changed. Else if the argument stack height has changed, the binding environment is adjusted, the code is re-generated and re-executed, and Defines added to Changed. Else if the phrase causes a side-effect, the binding environment is adjusted, the code re-executed and Defines added to Changed. Otherwise, the phrase does not cause a side-effect, so the binding environment is adjusted, the old value(s) used and Defines removed from Changed.

A del has no effect on Changed.

An edit is treated as a del followed by an append.
4.4.4.2. Algorithm 2.

Treating an edit as a del followed by an append loses the essence of most edits, that of adjusting a definition rather than adding a totally different definition. Algorithm 2 recognizes an edit as a change.

The Changed set now consists of identifier and type-of-change pairs. A change can either be a type, position or value change. An identifier with a type change must be completely re-evaluated. An identifier with a position change has the same type and so can avoid re-analysis but needs the code to be re-generated and re-executed. An identifier with a value change must be re-executed only.

A phrase's Defines set now is a set of (identifier, type, position, value) tuples.

On a pop, Changed has each identifier in Defines added as a type change.

On an append, if the buffer is not empty, Changed has each identifier defined added as a type change.

On a push, the type of change determines the re-evaluation necessary. If one of Frees is in Changed as a type change then the phrase must be re-evaluated and Defines added to Changed as type changes. Else if one of Frees is in Changed as a position change, or the height of the argument stack is different, then the binding environment is adjusted, the code re-generated and re-executed, and Defines added to Changed as position changes. Else if one of Frees is in Changed as a value change then the binding environment is adjusted and the code re-executed, and Defines added to Changed as value changes. Finally, if the phrase has side-effects it must be re-executed and Defines added to Changes as value changes. If the phrase does not have side-effects the binding environment is adjusted, the value can be re-used and Defines removed from Changed.

A del again has no effect on Changed.

On an edit, the old Defines is compared with the identifiers defined to add to Changed. New or deleted identifiers are type changes. If the edited phrase is invalid, the previous information of the popped phrase is retained.
4.4.4.3. *Even More Improvements.*

Even more improvements are possible, bearing in mind that the cost of implementing them (in terms of time and space) may outweigh their savings.

A function definition that is changed, but still has the same type, requires all dependent phrases to have their code re-generated as direct references are used in the FAM. This could be avoided. The function definition generates a closure consisting of text, a pointer to the body of the function, and a list of globals. If the globals list was changed to a pointer to a globals list then the closure could act as an indirect reference. Changing the function would involve changing the text and changing the globals but using the same closure.

The change hierarchy of *type* and *position* is not necessary. A phrase with a changed type that is still valid (and has no frees with changed positions or values) can re-use its code. The only exception to this is if the re-typechecking changes an overloaded operator. If for example, an integer operation is changed to a real operation, as the code has different routines for integer and real operation, the code must be altered according. Detecting changes in overloaded operators is difficult. If this scheme is used, a new change type *new* must be introduced. A *new* change, such as an append, implies a type and a position change.

A phrase on the buffer with a free identifier is dependent on that identifier. An identifier only changes (is added to Changed) when a phrase on the buffer is dependent on it. Thus appends when the buffer is empty are not recorded as changes. Currently, however, if there is a phrase on the buffer, even if none of its free identifiers are affected by an append, all identifiers defined by an append are regarded as changes. The Changed set will soon contain irrelevant entries that force unnecessary re-evaluation. Keeping track of free identifiers on the buffer can reduce the number of entries on Changed.

Glide used existing type information to reduce the cost of re-typechecking. The existing type information, the *inferred* type of each sub-expression, and the *expected* type of the expression containing the sub-expression, were used to restrict
the effect of a type change. The algorithms used in Glide produced dramatic reductions in the cost of re-typechecking (up to 50%). The re-typechecking in Glide, however, related to the system of loading definitions into the system. The use of existing type information in AIMLESS would produce some improvement but would be unlikely to produce such a large improvement.
Errors inevitably occur during programming, either in the program specifications or in implementing the specifications. Programs must be debugged - “the problem of program debugging is present in any programming or specification language used to communicate with the computer” (Shapiro 1983). Tools for debugging aim to aid in the removal of bugs from a program. This chapter describes the debugging tools of AIMLESS.

Debugging consists of three activities: noticing incorrect behaviour by the program, locating the bug(s), and fixing the bug(s). Testing is an input/output based form of debugging (Seviora 1987) where the behaviour is observed on the output produced for the input given. If the observed behaviour deviates from the expected or required behaviour, an error has been noticed. Tracing is one of the most popular techniques for locating bugs. The control flow in particular sections of the program is observed in order to isolate sections of the program where the actual behaviour does not match expected behaviour.

The design of ML reduces the occurrence of errors. Structured programming is encouraged by appropriate constructs. Static scoping (both outer-level and expression) provides security against accidental naming conflicts. Strong typing prevents program composition errors - “typechecking, combined with the clean semantic structure of ML, eliminates a large proportion of bugs at compile-time … (when I now occasionally program in untyped languages, I find myself thinking, over and over, ‘the ML typechecker would have trapped this bug …’)” (Cardelli 1984).

Errors do occur in ML programs, however, and a programming environment for ML must provide techniques for debugging. The eval loop that ANU ML is based on provides a powerful debugging mechanism - “the simple fact that ML is interactive [implemented in an eval loop] already provides a debugging environment far superior to any debugging system for batch-compiled languages” (Cardelli 1984). Program fragments can be evaluated to provide feedback on the behaviour of definitions. Definitions can be easily changed and re-tested without a
lengthy edit/compile/execute cycle. Still, more sophisticated debugging tools are needed for a programming environment.

This chapter describes two tools developed for debugging in AIMLESS. The first is an automatic re-testing mechanism that builds on the phraselist model of editing. The second is a tracing mechanism for locating bugs in an expression. Before the two tools are outlined, the implementation of function `makestring` in ML is presented, as it is essential to both.

5.1. Makestring.

In Section 5.2 of “The Standard ML Core Language”, Milner (1985) notes: “Two multi-typed functions are included as quick debugging aids. The function `print : ty \rightarrow ty` is an identity function, which as a side-effect prints its argument exactly as it would be printed at top-level. ... The function `makestring : ty \rightarrow string` is similar ... it returns as a string what `print` would produce on the screen”. This definition is deliberately vague, encouraging each ML system to implement as appropriate. This section describes the implementation of `makestring` for ANU ML. Function `makestring` is no longer included as part of the ML standard, presumably because of the difficulty in implementing it.

5.1.1. The Print Representation.

The print representation, how an expression is displayed at the top-level, is well defined for types such as integers, booleans and records with standard normal forms. Lists have a special print form, the list read as `::(1,::(2,nil))` is displayed as `[1,2]`. Closures are displayed by the string “fn”.

Strings are displayed in quotes with appropriate escape sequences. For example, ANU ML has the following behaviour:

```
- makestring "what";
  "\"what\"" : string
```

This is in contrast to Wikstrom (1987) where the above expression returns "what", as top-level strings are not enclosed in quotes. Not enclosing top-level strings in
quotes is an ad-hoc treatment of the print representation of strings and so was not followed.

User-defined types are displayed in full form. A print method for each type (similar to ideas in object oriented programming) could be used by `makestring` so that a user-defined type could have a special print form just as the system defined ML list has a print form.

The print representation of objects in an abstract data type are hidden. In ANU ML such objects are printed as `'-'` at the top-level. If the internal representation is hidden, however, then when a function in the abstype definition is traced, the abstract data type will not be displayed. This was found to be inconvenient when debugging. Currently `makestring` displays the internal representation. The type system prevents the internal representation being used outside the abstype definition so the security of abstype definitions is maintained.

ANU ML has arrays as a primitive data type. At the top-level they are printed as abstract data types. To increase the debugging value of `makestring`, arrays have their lowerbound and elements displayed.

5.1.2. Why Makestring Is Difficult To Implement.

At first glance, it may appear that implementing `makestring` should be easy; after all, the eval loop print routine does this when displaying the result of evaluating an expression. The print representation of an expression, however, cannot be generated without having associated type information. As the types of all top-level expressions are bound in the top-level binding environment, their values can be printed. Function `makestring`, however, must be able to generate the print representation of temporary values that do not have type information in the top-level binding environment.

The problem of lack of type information could be avoided if the type of the argument to `makestring` was known at compile time. This is not always the case.

These problems are now examined in turn.
5.1.2.1. The Run-time System And Type Information.

The run-time systems of all current ML implementations have no explicit type information (other than the ability to distinguish ref objects from other data objects for implementing equality). This is a feature of languages with strong static typechecking as the run time need do no typechecking and yet be sure of the correct typing of all expressions. For example, records are typically implemented as tuples without labels, as their labels are just type information and do not affect the run-time behaviour. A record such as \{john=3,fred=true\} is sorted on its labels (as fields in a record may appear in any order) to \{fred=true,john=3\} and is represented as (true,3). The print representation of this tuple cannot be generated unless the label information has been saved for this purpose.

5.1.2.2. Polymorphic Arguments To Makestring.

Although not having sufficient run-time type information prevents the print representation of an arbitrary expression from being generated, an alternative scheme could be used if the type of the argument to \texttt{makestring} was always known at compile time. For example Pascal also has a static typechecking system, but a Pascal \texttt{makestring} could be implemented by compiling in type information of the argument necessary for generating the print representation. ML has polymorphic typing, however, and the type of an expression that \texttt{makestring} is applied to may not be known at compile time. The type of the expression is not instantiated till later use. For example,

```ml
> fun f x = makestring x;
- val f = fn: 'a -> string
```

It is not till a call to \texttt{f} that the type of argument \texttt{x} is instantiated. This is not just a pathological case; if \texttt{makestring} is to be used to print a polymorphic argument when tracing, it will not know the type of the argument till a later instantiation.

5.1.3. Other Implementations.

All current implementations have problems implementing \texttt{makestring}. New Jersey ML, SML (from Edinburgh) and RML (Rutherford Laboratory) all treat
**makestring** as an overloaded function and restrict its domain to integers, reals, booleans and strings (RML only). As ML has very restricted rules for overloaded identifiers - their particular instance must be determined at compile time from the context in which they appear - this approach is not suitable for a general **makestring**.

A debugger for New Jersey ML (Tolmach and Appel 1990) effectively implements **makestring** to display a variable's run-time value. A variable's type is determined by examining the types of the variables of the function in which it appears (this in turn may require a recursive application of the algorithm to determine the types of these variables) and re-running the typechecking process. This algorithm relies on the New Jersey implementation which has continuations of previous states of the compiler available at run time.

FLIC (Peyton Jones 1988), a functional language intermediate code for compiling functional languages, uses a similar scheme as the FAM for implementing datatypes and will also have problems implementing **makestring** for a polymorphic language.

Miranda^+ (Turner 1985) has a similar function **show** for displaying the print value of an expression. Miranda, however, prohibits using **show** in a function definition if the type of the argument to **show** cannot be determined from context. This prohibits the use of **show** polymorphically, allowing the compiler to know at compile time **show**'s run-time use.

### 5.1.4. An Implementation Of Makestring.

Simply stated, implementing **makestring** is a matter of having run-time type information available when **makestring** is applied. Some possibilities are discussed before the actual implementation is described.

#### 5.1.4.1. Instantiating At Run-time.

One approach to implementing **makestring** would be to provide **makestring** at run time with any type information not available at compile time. For example,

^+ *Miranda is a registered trademark of Research Software Ltd.*
- fun f x = makestring (x,3);
> val f = fn: 'a -> string

From typechecking this phrase, it is known that makestring will have a tuple as an argument, the first field of some type 'a, the second field of type int. The variable 'a will be unified when f is applied. Now if the application of f gives the information that its free type variable ('a) is instantiated to some type, makestring can use this to print its argument.

A stack of instantiated values can be defined so that on application a function pushes all free type variables and their instantiation on the stack. The stack is popped when the application is complete.

This solution has problems with curried functions. For example,

- fun f x y = makestring (x,y);
> val f = fn: 'a -> (b -> string)

- val g = f true;
> val g = fn: 'a -> string

Now the problem to solve is, how is the instantiation of f's first free type variable ('a) to bool maintained so that on application of g, the two arguments needed to instantiate makestring are available? [This is the continuation information available to New Jersey ML mentioned in Section 5.1.3.]

This approach is attractive as it involves minimal change to the abstract machine, and is a direct solution to the lack of type information introduced by polymorphic typing. Unfortunately an elegant algorithm implementing this approach could not be devised.

5.1.4.2. Maintaining Run-time Type Information.

A more direct approach is to keep run-time types for all FAM objects on the argument stack. This can be implemented by keeping a type associated with each entry on the argument stack. The type can exist either on a separate but synchronized type stack or as part of a tuple entry of value and type on the
argument stack. The type is the type’s timestamp, a unique key identifying all
types, system and user-defined. This approach runs into difficulties as modules
with user-defined types and closures that use these types will have timestamps
that bear no relationship to the current type table. Even if these difficulties are
solved, this approach is very heavy handed as it implements a full run-time type
system for the FAM. Such a run-time type system is not necessary; for example,
\texttt{makestring} does not need to know the type of a closure to generate its print
representation.

5.1.4.3. \textit{Makestring As An Overloaded Function.}

As mentioned in Section 5.1.3, other ML implementations treat \texttt{makestring}
as an overloaded function. This is \textit{ad-hoc} polymorphism, a \texttt{makestring} instance
exists for integers, a different one for reals, and so on. Due to ML’s restricted rules
for overloaded identifiers, this approach does not generalize.

Recent work (Ophel and Cormack 1991) has suggested a more general over­
loading scheme for ML. This scheme uses implicit parameters (Cormack and
Wright 1990) and would permit a more general \texttt{makestring}. For example,

\begin{verbatim}
- fun f x = makestring (x,3);
> val f = [makestring: 'a -> string] 'a -> string
\end{verbatim}

where the implicit parameter, \texttt{makestring : 'a -> string} must be instanti­
atied
when \texttt{f} is used.

Datatype declarations would now implicitly add a \texttt{makestring} instance, as
they would already do for other system overloaded functions.

This approach is very promising but remains an extension to ML. Given the
advantages of the re-testing and tracing mechanisms of AIMLESS that require a
general \texttt{makestring}, implicit parameters must be explored for possible inclusion
into ML.
5.1.4.4. The Solution Adopted.

The solution adopted was to add information to the FAM (run-time) representation so that the print representation could be generated.

The FAM already contains some implicit type information used by the garbage collector. As described in Section 4.2.1, the FAM organizes cells into pages. The garbage collector can tell the type of cells in a page and the number of fields in any cell (with fields). For example, the garbage collector can determine a page contains record cells, and by inspecting a record cell, can tell the number of fields it contains.

This information is not enough by itself to generate the print representation of all FAM objects. For example, ML tupled pairs and lists both have the same format, record cells of length 2. Using the implicit type information, it can be determined an object is a record cell of length 2, but without further type information it is impossible to tell if the object is a tupled pair or list. Similarly, the Cardelli's FAM represents small integers, booleans and user-defined constants as FAM constants; the FAM constant 0 represents unity (type unit), 0 (type int), false (type bool) or any user-defined type with a constant first entry. See Cardelli (1984) for details of the efficient representation scheme with constants and transparents used for user-defined types.

The additional information added to the FAM representation was therefore needed to provide type information necessary to print an object (such as the labels of a record) and to distinguish values with equivalent FAM representations.

As described above, ML objects with different print representations but equivalent FAM representations occur in two places in the FAM. To implement \texttt{makestring}, the FAM representations of these ML objects had to be made disjoint.

User-defined types are now all represented in heap space; the more efficient representations are discarded. A variant cell now has as well as its tag field and field for the body of the variant, a field indicating if the variant is a constant (and
hence the body field should be ignored) and a print field with a representation of the constructor. The print representation of a variant cell pointer is defined as, the print field (if a constant), otherwise the print field followed by the print representation of the body.

Records and tuples must be distinguished, with records having label information. The solution was simply to add an extra field to records and tuples, nil for tuples, a pointer to a record cell of labels for records. ML lists and records both use record cells and must also be distinguished. Lists occupy record cells of length 2. Single field records, such as \{john=3\} also occupy record cells of length 2. Single field tuples do not exist in ML, an expression with brackets around it is just that, a bracketed expression and not a tuple of length 1. A record cell of length 2, with its second field either nil or a pointer to a record of length 2, is a list, otherwise it is a singleton record (as its second field will be a record cell of length 1 with the record label). Lists therefore still have Cardelli’s efficient representation. User-defined types will have the less efficient variant cell representation.

With user-defined constants now represented by variant cells, constant entries on the stack are either integers or ML constants such as unit, true, false and nil. The ML constants can be represented by new values; false by 0, true by 1, unit by 2 and nil by 3. The representation of boolean values takes advantage of logical branching operations in the compiled code that have false as 0, true as 1. A representation for integers must therefore be found that does not use 0, 1, 2 or 3. One solution is to make integers into pointers to integer cells. The drawback with all integers as pointers is that many expressions use small integers as intermediate values which will consume memory quickly, causing more frequent garbage collection, adversely affecting performance. The solution employed for integers was to exploit unused high-order bits in the target machines. The target machines have 32 bit words but only 28 bits will be used for addressing. The top two bits are already used by the FAM for garbage collection and import/export mechanisms for modules. The integer representation uses the next two bits, an integer is represented with a bias of $2^{29}$. The conversion from ML integer $n$, to FAM integer $m$, is simply $m := n + 2^{29}$. Thus integers in the range -268435456 ..
268435455 are represented. FAM objects can be recognized as integers if either their third or fourth high bit is set (but not both).

5.1.4.5. *Some Comments On This Implementation.*

This implementation of `makestring`, using format information supplemented where necessary, imposes overheads on the compiled code. The FAM code takes extra instructions to build ML objects. Records require an extra instruction for each field (for each label) and an extra instruction for the pointer to the label record. Tuples require an extra instruction to add their nil pointer. Lists are not affected by `makestring`. While system constants are also unaffected, user-defined types require extra instructions to represent constants as variants and instructions for their constant and print fields. Integers have the minor overhead of masking top order bits, but now have a wider range than previously. The run-time system also consumes extra space for these extra fields, this is noticeable representing user-defined constants as variant cells instead of directly on the stack. The execution time is slower as this extra information must be manipulated at run time and pattern matching of constants and transparents is now done by the slower pattern matching of variant cells.

Some simple tests indicate that code dealing with constants is 45% larger and 17% slower to execute, code with transparents is 25% larger and 36% slower (see Appendix 6 for details). As a wide body of ML programs has not been analysed (such a body is still being written by the ML community) it is not clear how often constants and transparents appear in ML programs. Again, it is emphasised that all the ML primitives, lists, booleans, numbers, are efficiently implemented.

The ANU ML compiler does not have debugging and production modes (as the Berkeley Pascal compiler (Joy et al 1977) has with its `-g` option to produce code with symbolic information for use with a debugger). The run-time typing approach to implementing `makestring` could be used in a system with modes; `makestring` could only appear in the debugging mode with the large overhead of full run-time typing, the production mode would not allow `makestring` and would have no run-time typing. Although having a production mode means the
Chapter 5: Debugging

final code may be more efficient, the computing folk-lore abounds with stories of bugs in production mode code that disappear when debugging mode is used. Also, the need to re-compile fragments to debug them is cumbersome. For these reasons, the implementation of `makestring` adopted, while representing some overhead, is an appropriate solution.

### 5.2. A Re-test Mechanism For AIMLESS

Testing is the usual method of noticing errors. The observed behaviour is compared with the anticipated behaviour, and if it is different, an error in the program has been noticed. Programming environments based on the eval loop allow program fragments to be tested as the program is built. Expressions are entered, not to form part of the program, but to observe and test current program behaviour. Tests in such programming environments are normally lost during development. If definitions used by a test are changed, the user must remember to re-evaluate the test to check program behaviour. "The test cases represent a valuable investment that, in this environment [an incremental program development system], disappears after the testing has been completed. Whenever the program has to be tested again (e.g., after correcting an error or making an improvement), the test cases will have to be re-invented" (Myers 1979).

The editing system of AIMLESS avoids the problem of losing tests during development by saving them on the script. Valid test expressions on the script must be popped and then pushed to alter definitions they depend on. The results of tests, however, are lost. The AIMLESS re-test mechanism overcomes this problem by storing the values of tests moved onto the buffer. When a test is moved back onto the script, it is re-evaluated, and the old and the new values compared. The `assert` command provides a mechanism to define the expected behaviour of a test which is useful in providing program documentation.

Tests are still lost when the development session is finished. See Section 5.2.7.2 for comments on keeping tests for the lifetime of a program (i.e., through maintenance).
5.2.1. What Is A Test?

The re-test mechanism regards all expressions as tests, whether they are part of the program or not.

In the eval loop, a test’s behaviour is described by any interactive input, the result of the test and any interactive output. Note that an eval loop test does not describe any store side-effects of the test, unless these are returned as part of the result of the test or are displayed as interactive output.

The AIMLESS re-test mechanism uses the eval loop model of a test, without considering any interactive input or output (as they are considered outside the AIMLESS system). See Section 5.2.7.2 for comments on using state information in tests.

5.2.2. Changed Tests.

A test is popped from the script onto the buffer when earlier phrases on the script are to be edited. The value of the test may have changed when the test is pushed back onto the script. The re-test mechanism must inform the user when a test has changed its behaviour. A change in test behaviour is not necessarily an error, the previous value may have been incorrect and the new value the desired result. Tests therefore display a warning message but do not fail. Pushing a changed test still succeeds, the test is placed on the script.

As the re-test mechanism uses the eval loop model of a test (i.e., the result of evaluating the expression), the comparison mechanism used to compare tests is based on `makestring`; two test values are regarded as the same if they have the same `makestring` form. Thus all closures are the same as they have the `makestring` form "fn". For example, with the phrase `val x = [t1,t1];`, the test expression `hd x`; evaluates to a closure (the function `t1`). If the phrase is changed to `val x = [reverse,reverse];`, the test again evaluates to a closure, albeit a closure with different behaviour. The comparison mechanism would regard the test as having the same value. While this may appear a problem, it is not. In terms of observable behaviour the test `hd x`; displays the same representation.
“fn” of the closure to the user on each evaluation. Any fault lies not with the comparison mechanism but in the test itself. A more illuminating test would be 

\( (\text{hd } x) [1,2,3,4] \).

Tests can raise exceptions (see Section 3.1.6). The comparison mechanism must be able to compare values and exceptions. A test with a previous value and a current exception (or vice versa) has clearly changed. The comparison between two exceptions is straightforward, the exceptions and their values must be compared.

5.2.3. Types And Tests.

A test with a value will also have a type associated with the value. The type is often useful to the comparison. For example, the test \( f \) on the phrase \( \text{fun } f \ x = x + 1; \) being changed to \( \text{fun } f \ x = \text{hd } x; \) could warn that the test has changed from type \( \text{int} \rightarrow \text{int} \) to \( \text{a list} \rightarrow \text{a}. \)

The current re-test mechanism saves the textual form of the type of the old value of the test, and uses this when comparing old and new values.

5.2.4. The Re-test Mechanism.

The AIMLESS re-test mechanism uses expressions on the script as checkpoints in the development process. A test’s value may change from when it is moved onto the buffer, while the script is edited, to when it is pushed back onto the script. The previous and current values are compared using their print representations. A warning is given of a changed test, displaying the previous and the current values. The \textbf{push}, while displaying a warning, still succeeds. An example demonstrates this process.

```
- fun abs (n:int) = n;
> val abs = fn: int -> int
- abs 0;
  0 : int
- abs 3;
  3 : int
```
The user has defined an incorrect absolute function abs and tested it on 0 and 3, and observed the above behaviour.

- abs 3;
  3 : int

The test for 3, however, is in error. The user then enters alter abs; and edits the phrase to fun abs (n:int) = -n. The three tests, abs 0;, abs 3; and abs 3; are pushed and re-evaluated. The value of the first test is unchanged but the new values of the next two tests are both different from their previous values. The following warning messages are displayed.

***************
test value changed
Test Phrase: abs 3;
Old: 3 : int
New: -3 : int
***************
***************
test value changed
Test Phrase: abs 3;
Old: -3 : int
New: 3 : int
***************

The user is informed that the test abs 3; no longer evaluates to the same value. Thus, even though test abs 3; returns the desired result, the user sees that the new version of abs is still not correct.

The re-test mechanism handles tests raising exceptions. For example,

- fun first str = extract(str,1,1);
> val first = fn : string -> string
  - first "abcd"
    "a"
  - first ""
  Unhandled exception Extract
The user defines function `first` to return the first character in a string. The function works for a non-empty string but raises an exception on an empty string. The user alters the definition of `first` to:

```
fun first "" = "" | first str = extract(str,1,1);
```

The two tests, `first "abcd"`; and `first ""`; are pushed and re-evaluated. The value of the first test is unchanged, the value of the second test is changed and the following warning displayed.

```
******************************
test value changed
Test Phrase: first "";
Old: Unhandled exception Extract
New: "": string
******************************
```

The user can observe that the previous incorrect behaviour of the test has been rectified.

5.2.5. Assertions.

The re-test mechanism provides a means of using tests as checkpoints. The `assert` command provides a mechanism for defining the expected behaviour of a test. The syntax of the assert command is

```
assert_command ::= assert <expression>;
```

The expression must be of type boolean. The semantics of an assert is straightforward. The expression is evaluated, if it is true, the assertion is added to the script, otherwise the phrase fails and is placed on the buffer. Thus `assert` and re-test equality have different semantics. Originally `assert` used re-test equality but users were more familiar with other systems that used equality, and asked for `assert` to use normal equality.

The `assert` command is an ML declaration so that it can appear within aggregates (unlike AIMLESS commands). For example,

```
datatype a = b | c;
fun f b = 0 | f c = 1;
```
assert \( f \ b = 0 \);

can be abstracted to (using the make abstype command)

\[
\begin{align*}
\text{abstype} & \ a = b \mid c \\
\text{with} & \\
\text{fun} & \ f \ b = 0 \mid f \ c = 1; \\
\text{assert} & \ f \ b = 0 \\
\end{align*}
\]

This also means that (as with other ML declarations) that any unhandled exceptions are caught at the top-level. It is not possible therefore to write assert \( \text{hd} [] = \text{raise} \ \text{Hd;} \), this would have to be written assert \( (\text{hd} [\] \text{handle} \ \text{Hd} => 0) = 0; \).

The phrase assert \( (x:3) = () \); is a valid assertion. Obviously the intention of assertions (and indeed tests) is to provide feedback to the user on the program behaviour, not to alter the state, and such constructs, though legal, are not good programming practice.

The AIMLESS command make assert converts the last phrase on the script from a test into an assert. A test \( t \) that has the value \( v \) is converted into assert \( t = v; \). The conversion fails if equality cannot be applied to the test and its value (e.g., if the test returns a closure).

An example based on the abs example for tests demonstrates assert.

- fun abs \( (n:\text{int}) = n; \) \\
  > val abs = fn \: \text{int} \to \text{int} \\
  - abs 0; \\
  - abs 3; \\
  - make assert;

The user has defined an incorrect absolute function abs and tested it on 0 and 3, and observed the above behaviour. The behaviour of abs 3; has been converted into an assertion.
The test for `~3`, however, is in error. The user then enters `alter abs;` and edits the phrase to `fun abs (n:int) = ~n`. The test `abs 0;` is pushed and re-evaluated. Its value is unchanged. The assertion is then pushed and re-evaluated. The assertion does not hold, an error is reported and the pushes of the `alter` command stop.

```
Assert Failed
assert abs 3 = 3;

- show all;

fun abs (x:int) = ~x;
abs 0;
```

5.2.6. Implementing The Re-test Mechanism.

With the mechanisms for manipulating the phraselist and evaluating phrases developed for the editing system, and using the print representation for comparison, the re-test mechanism is simple to implement.

The type `PhraseEntryT` (each phrase on the phraselist is of this type) has three fields added. Field `Value`, a string, is the textual form of the test value. If `Value` is nil, the phrase does not have a value. Field `ValueType`, a string, is the textual form of the type of `Value`. Field `Exception`, a string, stores the value of the exception raised by the test. If `Exception` is nil, the phrase did not raise an exception.

On a successful `append`, if the phrase raises an exception it must be a test, as tests are the only phrases raising exceptions that are placed on the script. The value of the exception is stored in field `Exception`. 

100
On a **pop**, if the phrase is a test and has a value on the argument stack, **Value** is set to the **makestring** form of this value and **ValueType** the textual form of the value's type. Otherwise **Value** is set to nil.

On a **push**, if the phrase is a test, then its previous result, either **Value** and **ValueType** or **Exception**, is stored. The test is then pushed. If it successfully evaluates then the new result (value or exception) is compared with the old result. A warning message is printed if the old and new results are not the same. If the test does not successfully evaluate the old result is discarded; there is usually a type error and the old result is no longer useful.

**Assert** was implemented by adding the **Assert** operation to the FAM. The **Assert** operation takes the top entry on the argument stack and tests it. If it is true, no action occurs. If it is false, the assertion is displayed and the phrase is placed on the buffer. Assertions do not return a value. Command **make assert** is implemented by popping the test onto the buffer and generating an **assert** command using the body of the test and the print representation of its value. The generated **assert** command is then pushed.

5.2.7. **Comments.**

5.2.7.1. **Related Work.**

The role of testing a developed system in the software lifecycle is well established (Birrell and Ould 1985). The role of testing during the programming phase is not so well established.

Hamlet (1977) developed a system for adding tests to code. When the code was compiled the tests were evaluated and, if in error, produced a compile-time error. Tests were specified for a function as integer pairs \((x, y)\) where \(x\) was the integer input to the function and \(y\) the integer result. While the domain of tests was restricted (integers only) and the tests had to be explicitly added by the user, the system had the basic idea of having tests to check changed (re-compiled) code.

The use of tests within the eval loop as checkpoints in program behaviour was examined in the Program Testing Assistant (Chapman 1982). The PTA was
a LISP system operating in an eval loop. After evaluation, expressions could be explicitly marked as tests, and functions that the test depended on could be nominated. If a function was modified, dependent tests were re-evaluated and warnings given if results had changed. A test saved the terminal input stream and the state for re-evaluation. The state consisted of the internal state filtered to specify parts that should have the same value as when the test was originally evaluated. The value and the terminal output stream were saved as the test result. Results could be compared, either as equal, set equal or isomorphic. The PTA suggested many basic ideas used in the AIMLESS re-test mechanism. The main improvement is that the PTA relied on the user to nominate explicitly tests, dependencies and the state for re-evaluation. In a LISP system with side-effecting operations, this may be the only reasonable way to manage the state. In AIMLESS the re-test mechanism is integrated with the editing system to provide valuable, but unobtrusive, support.

Tests are observations of program behaviour and assertions define the required behaviour of a given test. Assertions will normally be instances of the axioms and theorems that define the behaviour of the function under test. The use of axioms in program specification is discussed by Sannella (1986).

5.2.7.2. Future Work.

Testing in AIMLESS could be improved by a mechanism for automatically generating test cases. See Section 7.4.16 of Birrell and Ould (1985) for a discussion of test data selection.

While tests are valuable in the program development phase, they have greater potential for use during program maintenance. Changes could be checked using existing tests and results instead of requiring tests to be re-designed and their results correctly anticipated. This motivation was a consideration in Hamlet (1977). See Section 6.5 for comments on adding tests to the ‘finished’ program.

The current version of the re-test mechanism has tests re-evaluated whenever they are pushed. As outlined in the re-evaluation algorithms in Section 4.4, phrases
do not necessarily need to be re-evaluated on a push. Tests need only be re-evaluated if a dependency has changed in a way that can affect the result of the test, or in the case of tests using the state, if the state has changed.

The PTA allowed both an input stream and a subset of the internal state to be associated with a test. In AIMLESS tests that use the external state may return a different value due to the external side-effects of phrases not being correctly undone. It would be possible to build mechanisms in AIMLESS to store the interactive input to an expression for use when re-evaluating. Similarly the interactive output could be stored (as in the PTA) as part of the result, and used when comparing the result of re-evaluating a test. This has not been implemented because interactive input/output is part of the external state, and, as discussed in Section 3.3, regarded as outside the control of AIMLESS.

The re-test mechanism uses the model of the eval loop test. Store side-effects are not tested for change unless they effect the result of the test. The user must include an explicit test to check store side-effects. It would be possible to test for store side-effects using the store changes maintained to undo store side-effects on a pop (see Section 4.3.4). The original changes and the re-test changes would be compared and the user notified on any differences.

It is interesting to consider the possibility of a language with input using a lazily evaluated stream of characters. Such an input stream is equivalent to a string, except the input stream may be defined during the evaluation of the expression. Once the expression is evaluated and the input stream completely defined (for that expression) the expression could be re-evaluated with the string equivalent. For example, expression \( E \) takes an input stream as an argument. The evaluation of \( E(keyboard) \); where keyboard is standard input, will return a result. During the evaluation the characters \( c_1, c_2, \ldots, c_n \) will be read in from standard input. The expression could be re-evaluated as

```plaintext
let
  keyboard = "'c_1, c_2, \ldots, c_n'"
in  E(keyboard)  end
```
and, except for external state considerations, would return the same result. This would be a cleaner story than the programming environment storing interactive input for use when re-evaluating tests.

5.3. A Trace Mechanism For AIMLESS.

Tracing is one of the most popular techniques employed in the location phase of debugging. A tracing mechanism in a functional system allows the user to specify functions to be traced. On the application of a traced function the values of the arguments are displayed, and when the function body has been evaluated, the result is displayed. The design and implementation of the trace mechanism for AIMLESS is described.

5.3.1. The Basic Functionality.

Tracing is turned on by the `trace <ide-list>` command, turned off by the `untrace <ide-list>` command. Each function in the identifier list argument is traced.

A traced function displays the value of its argument on application and the result of the application.

```
- fun f x = (x:int) * x;
> val f = fn: int -> int
- trace f;
- f (3+2);
Enter f: 5
Exit f: 25
  25 : int
- untrace f;
- f (7+8);
  225 : int
```
Curried definitions display trace information on evaluation of the traced function body and display all arguments to the function.

``` ML
- fun f x y = (x:int) + y;
> val f = fn: int -> (int -> int)

- trace f;

- val g = f 7;
> val g = fn: int -> int

- g (9 * 2);
Enter f: 7 18
Exit f: 25
 25: int
```

Mutually recursive functions can be traced separately. Functions defined in local and abstype declarations can be traced. The internal functions (such as g in the example below) are hidden and cannot be traced.

``` ML
- local
  fun g x = x
  in
    fun f x = g x
  end;
> val f = fn: 'a -> 'a

- trace f;

- f 7;
Enter f: 7
Exit f: 7
 7 : int
```

### 5.3.2. Method.

The trace schema is a source code transformation of the function definition. The original function is replaced by a new function of the same name. The new function displays the arguments and then defines the original function locally with a new name, before binding the original function application to a temporary variable. If this application raises an exception, the exception is trapped, reported, and then re-raised. If the application successfully evaluates, the result is displayed and then returned.
Chapter 5: Debugging

A function definition is transformed from

\[
\text{fun ide arg}_{11} \ldots \text{arg}_{1n} = \text{exp}_1 \\
\vdots \quad \text{ide arg}_{m1} \ldots \text{arg}_{mn} = \text{exp}_m;
\]

to the definition

\[
\text{fun ide' arg}_{11} \ldots \text{arg}_{1n} = \text{exp}_1 \\
\vdots \\
\text{ide' arg}_{m1} \ldots \text{arg}_{mn} = \text{exp}_m
\]

in

\[
\text{val x = (ide' x}_{11} \ldots x_{n}) \text{ handle v =>} \\
(\text{output (std_out, "Exit ide : " makestring(v)"\n");} \\
\text{raise v})
\]

end

in

\[
(\text{output (std_out,"Exit ide : " makestring(x)"\n");} \\
x)
\]

end);

where ide' and the various \(x'_i\) are unique identifiers generated so as not to interfere with free identifiers in any \(\text{exp}_j\).

The behaviour, that is the result, of the original function and the traced function should be identical, given that \text{output, std_out, } ^{\wedge} \text{ and makestring behave as expected.}

5.3.3. Problems.

5.3.3.1. Changing The Traced Function.

Transforming the original function definition is a form of editing the function definition. That is, the original function definition must be replaced by the transformed definition.
Chapter 5: Debugging

The original function definition is represented in the FAM by a pointer to a closure. The closure cannot just be replaced by a traced version as all later definitions using the original definition contain references to the original closure. This is the problem encountered in editing direct references, see Section 2.2.1.

The closure, however, consists of a pointer to the text of the function and a list of globals used in the function. If the globals are unchanged, the pointer to the original text can be substituted by a pointer to traced text. This is equivalent to editing by indirection but with the knowledge that the edited definition (the traced definition) has the same functional behaviour as the original definition. That is, for the same inputs the traced function either returns the same result as the original function or raises the same exception as the original function. The problems introduced by editing the script, those of an invalid or inconsistent script, do not occur. A schematic view of altering the text of a traced function is shown in Figure 5.1.

![Diagram of function f and its closure]

**Fig. 5.1 Tracing Function f:** The text of f is changed to a traced version while retaining the same closure.

The only new free variables in the traced version of the function definition are `std_out`, `output`, `^` and `makestring`. If these four are FAM primitives and
implemented directly in FAM code rather than as calls to definitions on the ar­
gument stack then the traced definition has the same globals as its old definition.
Before tracing, a check must be performed to ensure that `std_out`, `output`, `^
and `makestring` have not been re-defined; the transformation is refused if they have.
Note that if the FAM treated free variables as literals rather than globals than
this would not be a problem.

An alternative scheme would be to leave space in every traceable closure for
trace globals. Identifiers `std_out`, `output`, `^
and `makestring` could then be
redefined by the user and used by the trace transformation.

In a system with a generational garbage collector, changing the function text
with a traced form would mean older closures could have pointers to new objects.
Closures would therefore have to treated as reference objects (see Section 4.4.2).

5.3.3.2. Tracing Imported Functions.

Function definitions imported as module bindings do not have their definitions
available to be transformed for tracing. It would be desirable to be able to trace
imported functions. A mechanism was designed to allow this.

The command `importtrace <module>;` is equivalent to an `import` com-
mand except all functions imported then have a

\[
\text{val } \langle\text{ide}\rangle = \text{fn } x' = \langle\text{ide}\rangle x';
\]

definition added to the script. The `\langle\text{ide}\rangle` inside the val definition refers to the
`\langle\text{ide}\rangle` imported from the module (as `val` definitions are not recursive definitions).
A call to a function imported by `importtrace` therefore goes through this tempo-
rary definition, but behaves as if imported by `import`. The temporary definition,
however, can be traced. The transformation is simpler than for `fun` definitions.
A temporary definition is transformed from

```plaintext
val ide = fn PAT => EXP;
```

to the definition

```plaintext
val ide = fn a' =>
  (output (std_out,"Enter ide : " ^ makestring(a') ^ "\n");
  let
    val x = ((fn PAT => EXP) a') handle v =>
      (output (std_out,"Exit ide : " ^ makestring(v) ^ "\n");
       raise v)
  in
    (output (std_out,"Exit ide : " ^ makestring(x) ^ "\n");
    x)
  end);
```

A module defined as

```plaintext
module List
body
  fun length [] = 0 | length (_::tl) = 1 + length tl
end;
```

will behave as follows when imported in trace mode:

- `importtrace List;`
- `module /users/jlo/src/lib`
- `val length = fn: 'a list -> int`
- `val length = fn: 'b list -> int`
- `length [1,2,3] + length [4,5,6];`
  `6 : int`
- `trace length;`
- `length [1,2,3] + length [4,5,6];`  
Enter length: [1,2,3]
Enter length: [4,5,6]  
Exit length: 3  
Exit length: 3  
6 : int
Note that the command `importtrace` imports the standard library and function `length`, and then adds a temporary definition. This is why the binding to `length` is shown twice. When function `length` is traced its arguments and result are shown only when the imported function is called. The internal calls to `length` inside the module are not traced.

5.3.3.3. Tracing Functional Forms.

As shown in the previous section, definitions of the form `val <ide> = PAT => EXP`; can be traced. Not all `val` definitions have a suitable form for tracing. Consider

```plaintext
- val function_list = [plus 1,plus 2,plus 3];
  > val function_list = [fn,fn,fn] : (int -> int) list

- val [f,g] = tl function_list;
  > val f = fn : int -> int
  | val g = fn : int -> int
```

Functions f and g cannot be traced by source code transformation as they do not have a suitable definition.

Any functional form that does not have a suitable definition, can be traced by the user explicitly adding a temporary definition as is done by `importtrace`.

5.3.4. Implementing Trace, Untrace and Importtrace.

See Appendix 7 for the algorithms for implementing commands `trace`, `untrace` and `importtrace`.
5.3.5. Some Comments.

The basic trace schema displays only limited information of function application. No attempt has been made to display the level of recursion in a function call, nor to indent nested calls, such as in the example below.

```ml
- fun length [] = 0 | length (_::tl) = 1 + length tl;
> val length = fn: 'a list -> int
- trace length;

- length [true,false,true];
  1: Enter length : [true,false,true]
  2: Enter length : [false,true]
  3: Enter length : [true]
  4: Enter length : []
  4: Exit length : 0
  3: Exit length : 1
  2: Exit length : 2
  1: Exit length : 3
```

To implement the display of level of recursion a variable must be defined and updated on each entry and exit.

As ML has pattern matching in function definitions, it may be appropriate to display the bindings made on application.

```ml
- fun s (x,(3,y,"hello"),z) = x + y + z + 1;
*** warning: pattern is not exhaustive ***
> val s = fn: (int * (int * int * string) * int) -> int
- trace s;

- s (7,(3,4,"hello"),10);
  Enter s : (7,(3,4,"hello"),10)  x=7 y=4 z=10
  Exit s : 22
    22 : int
```

ML also has clausal function definitions. The user may wish to know which clause was taken and why previous clauses failed. Both pattern matching information and clause information require transformations of the expression associated
with each clause, not the function definition itself. Such information, although extending the trace mechanism, is not essential for tracing, and so has not yet been implemented.

Tracing is just one operation of debuggers (or steppers) that allow control flow to be traced, and run-time values to be observed, and even altered, during evaluation. The complex nature of state changes in imperative languages means information is needed, not just on the program’s control flow, but also on the program’s state. For example, dbx and descendants (Adams and Muchnick 1986) allowed breakpoints to be set at particular source lines or procedures in Pascal and C programs. Evaluation stopped on reaching a breakpoint, the program state could then be examined and altered. Evaluation could either continue to the next breakpoint or be stepped through line by line. More powerful debuggers, such as Interlisp allowed run-time structures (e.g., the call stack) to be manipulated. Zstep (Lieberman 1984) allowed steps to be undone, allowing for a step to be done, an error noticed, and then the step re-done, perhaps decomposed to provide better feedback on the location of the error.

The need for an elaborate debugger in a functional system is questionable. It can be argued that tracing lacks the functionality of the debugger to ‘correct’ or modify evaluation on the fly. In a debugger, an error located after lengthy evaluation can be corrected and evaluation continued to locate further errors. Unusual cases can be tested by setting values during evaluation rather than being required to set up a lengthy test to achieve the same result. Such program development style is poor and not methodical, and is not encouraged by AIMLESS. It is also unnecessary in functional systems - “the debugger was seldom used since most of the time the tracer was needed to determine a suitable place for a breakpoint, and could also indicate the error in many situations” (Koopman 1987).

As has been mentioned earlier in the chapter, a debugger exists for New Jersey ML (Tolmach and Appel 1990). Currently the system does not allow modification of values during a debugging run, though there are plans to allow reference objects to be changed.
Tracing is a natural tool for call-by-value functions. A function with lazily evaluated arguments may not be able to display arguments during a trace. ML avoids this problem - “ML does not use lazy evaluation; it calls by value. This was decided for no other reason than our inability to see the consequences of lazy evaluation for debugging” (Milner 1983).
Chapter 6: The Outside World

Chapter Six: AIMLESS And The Outside World.

Chapters 3, 4 and 5 have described the programming environment AIMLESS. The environment has been presented in terms of producing a program, but, other than for external side-effects, there has been no mention of the ‘outside world’, i.e., the computing environment outside the AIMLESS development session. The relationship between AIMLESS and the outside world includes the issues of

- extracting the program listing
- the lifetime of the development session (logical session) in relation to entering and exiting the ML system (physical sessions),
- the lifetime of data structures and programs created in a logical session,
- maintaining a program once it is ‘finished’.

The first section examines the program listing. The lifetime of logical sessions is discussed in section two. An alternative form of logical sessions is examined in section three. Section four describes the saving of data structures and programs as modules and the form modules take in the outside world. The final section examines mechanisms for aiding in the maintenance of ‘finished’ programs in the outside world.

6.1. The Program Listing.

The script can be used as the basis of the program listing (see Section 2.4.2). The script, however, will contain redundant phrases, such as tests, reflecting program development that are not needed by the listing to describe the image. In a side-effect free system where the image is the binding environment, expressions on the script do not add to the binding environment and do not have to be included in the listing. In a system with side-effects, expressions causing side-effects affect the image and must be included in the listing.

In ML, expressions are converted into definitions by the evaluator. An expression $EXP;$ is converted into $\text{val it } = EXP;$. Identifier it can then be used as
an intermediate value. Only expressions affect it, definitions (unless they explicitly bind to it) do not affect the value of it.

Identifier it can be used by definitions. For example, in the phraselist `<3 + 4; val a = it;| >`. The listing for this script cannot just be `val a = it;` as it will not be bound.

It is tempting to avoid this complication by treating it as a shorthand mechanism. Either it can be treated as a shorthand mechanism for value substitution or for textual substitution. For example,

```
- fun f x = x + 1;
  > val f = fn: int -> int

- f 7;
  8 : int

- val a = it;
  > val a = 8 : int
```

The phrase `val a = it;` is either shorthand for `val a = 8;` (value substitution) or for `val a = f 7;` (textual substitution).

If it is bound to a functional value, there is no printable value that can be substituted for it. The program listing requires a printable form for definitions, ruling out value substitution.

If it is treated as a textual substitution, the listing and image may not be consistent. For example,

```
- f 7;
  8 : int

- fun f x = x + 10;
  > val f = fn: int -> int

- val a = it;
  > val a = 8 : int
```

Now a will have a definition of f 7 but the value 8. As the script must be consistent, textual substitution is unsuitable.
Chapter 6: The Outside World

The solution adopted was to retain all relevant it bindings as part of the listing. Now the listing contains definitions, side-effecting expressions and any expression used by the program listing. For the phraselist <3 + 4; val a = it; | > the program listing would be 3 + 4; val a = it;.

An alternative view of expressions in the listing is to regard redundant it declarations a particular instance of the more general problem of redundant declarations in a listing. A mechanism then would then take a list of the “real” program listing declarations and retain only relevant phrases for these declarations in the listing. The solution of removing redundant it bindings is a simple and convenient (for the user) version of this more general mechanism.

See Appendix 5 for the implementation of show program;

6.2. Logical Sessions.

A logical session is an interaction between a user and the ML system which results in a completed program or module. A logical session may consist of an arbitrary number of physical sessions (entering and exiting the ML system). Finishing a physical session has no significance to the program under development and should have no significance to the logical session. It is inconvenient to force a logical session to have to fit into a physical session.

When a physical session is finished, the logical session should be suspended until it is continued by a later physical session. APL (Iverson 1962) had workspaces. A user worked within an active workspace that could be saved by the finishing physical session and re-started by another physical session. Interlisp had rollout and rollin which respectively saved the current image and loaded it again. Similarly Smalltalk (Goldberg 1984) had snapshots, system images that could be saved and loaded. Poly/ML (Matthews 1988a) had databases that were the heap of an ML session and could be saved and loaded.

The suspension of logical sessions for AIMLESS are discussed in the next four sub-sections: from a simple saving of some information, to a reconstruction of the session, to an image dump, to a more sophisticated mechanism.
Chapter 6: The Outside World

The suspension of logical sessions has not been fully implemented for AIM-LESS and only exists at the save all level. The difficulties of implementation and uncertainty about the preferred form, have delayed implementation.

6.2.1. Save All.

A form of suspension is simply to save the textual form of phrases on the script and on the buffer. Command save all takes a string, a file name, and writes phrases on the script and then the buffer to the file. The file can then be read (by the use command) as the first command of a later physical session.

Command save all can only be considered a crude way to connect physical sessions for several reasons. The format of the script and the buffer may be lost when read by the later physical session. Valid phrases on the buffer may move onto the script, while all of the buffer after the first error will be flushed as one phrase onto the buffer. Phrases on the script with interactive or file store input are re-executed and may evaluate differently if the input is different. Phrases on the script because of an interrupt may evaluate differently or loop until again interrupted. All trace information is lost - all functions are untraced. The values of tests on the buffer are also lost. Finally, continuing a logical session now has the cost of re-evaluating the phrases saved by save all.

For flexibility, command save script writes only the script to its specified file, while save program writes only the program.

6.2.2. Re-constructing The Session.

The save all command does not maintain the format of the buffer, does not save the values of tests, and does not save trace information. This information can be saved by command savesession which then can be used to re-construct the script.

Command save session would save the textual form of each phrase on the buffer, the textual values of tests, any input read by phrases on the script and information to indicate if a phrase has traced identifiers. The phrases on the script would then be saved textually. Command load session would re-create
Chapter 6: The Outside World

the buffer using the textual information and then re-construct the script by re-
evaluating the appropriate phrases using their saved input. As with command 
**save all** this approach has the overhead of re-evaluating the script.

6.2.3. Image Dump.

The simplest method of suspending logical sessions is just dumping the image 
(to file store) and re-starting this image at a later stage.

The image dump is expensive, requiring megabytes of memory. The evaluator 
is included in the dump. Unnecessary information is also saved. For example, ANU 
ML has a copying garbage collector, hence half of data space is not in use (except 
during a garbage collection). An image dump would still save this unused half of 
data space.

The image dump is expensive, but if such memory is available, it is a practical 
method of implementing logical sessions.

6.2.4. Saving AIMLESS.

A more efficient approach is to save only the necessary parts of the logical 
session. The initial call to AIMLESS specifies the logical session to be called. 
On finishing the current physical session, the logical session is saved, ready to be 
continued.

The logical session consists of the FAM state, the evaluator's data structures 
and the mechanisms for maintaining the phraselist. The FAM state is basically 
the argument stack and everything on it. Following pointers, as is done by the 
garbage collector, covers the necessary parts of the heap. The data structures of the 
evaluator include the identifier table (storing fixity information) and the binding 
environment. The phraselist mechanisms need the text, abstract syntax tree, 
trace information, fixity information, weak type information and it information of 
phrases and the results of tests on the buffer to be saved. As well, ChangedStack 
must be saved.
The logical session information must be translated into a file store representation and translated back when the session is continued. Shared objects and complex structures (such as the AST) make translation difficult.

6.3. An Alternative To Logical Sessions.

Logical sessions assume that on finishing a physical session the current state of the logical session should be saved. An alternative view has that on finishing a physical session all new or changed definitions should be saved in file store for later use. The Interlisp **CleanUp** routine provides such a facility. On a call to **CleanUp**, or on finishing the session (which calls **CleanUp**), all definitions read from file store and altered have their new form written out to their associated file. A new definition can have a file associated to it, or the user is prompted for one by **CleanUp**.

The model of files with definitions being updated by physical sessions does not fit in with the paradigm suggested by AIMLESS, that of a logical session developing a module. **CleanUp** is implemented in AIMLESS for the convenience and flexibility of users.

**CleanUp** takes an argument **script**, **buffer**, **all** or **program** and writes out all phrases specified by the argument to their associated files. A phrase is associated with a file if it is loaded from that file (by a **use** command). Any phrases not associated with a file (entered in the eval loop) are prompted for a file and retain this association. An optional second argument **prompt** prompts all phrases, associated or not, and is therefore useful for re-organizing the association of phrases to files.

Nested **uses** cause problems. Consider file A with phrases A1; **use** "B"; A2; and file B with phrases B1; B2;. After A is read (by a **use**) the phraselist (assuming all phrases are valid) will be <A1;,B1;,B2;,A2;>. As the phrase **use** "B"; must be retained as part of file A, it cannot be treated simply as textual substitution and discarded. To avoid this, it is recorded as a dummy phrase on the phraselist and so is not be lost on **CleanUp**. The phraselist will be <A1;,use
Chapter 6: The Outside World

"B", B1; B2; A2; | >. A dummy phrase can be moved on the phraselist as any other phrase, but has no effect when popped or pushed (so that it does not re-load its file when pushed). If edited, it is treated as a new phrase.

Interlisp has dynamic scoping, re-defining a function is equivalent to replacing the previous function definition. As ML has static scoping, re-defining a function is a new and separate definition. A file that is read twice will therefore have repeated phrases on the phraselist. As CleanUp simply writes all phrases (even if repeated) associated with a file, phrases will appear twice in the cleaned up file.

It is possible to have invalid orderings; for example, with the phraselist <A1; B1; A2; B2; I > where A1 and A2 are associated with file A and B1 and B2 are associated with file B. As with repeated phrases CleanUp simply writes all phrases to their associated file, even though using file A and then file B will create a different phraselist.

6.4. The Lifetime Of ML Objects.

Once a program has been developed in a logical session it must be saved in the outside world so it can be used in the development of other ML programs and modules. It is also useful for ML data structures to be able to outlive the logical session creating them so that they too can be used by other logical sessions. One method of saving data structures is for the user to write the data structure out to file store and then read the structure when required. This is inconvenient as the translation from ML data structure to file format can be difficult and time-consuming and is also not type-safe, as the translation routines can coerce objects.

As pointed out by Harper (1986a) the module is the natural unit of persistence for ML both for programs and data structures. A module, both in ANU ML and standard ML, must by definition be self-contained; all bindings used by a module must either be defined in the module itself, or be imported from other modules. For example, with the script <val y = 10; fun f x = x + y; | >, to put f in module X, the definition of y would also have to be in the module. The module would have the form
Chapter 6: The Outside World

```ocaml
module X
body
  val y = 10;
  fun f x = x + y
end;
```

As a module is a self-contained unit it can be imported by any other module. As an AIMLESS logical session is consistent, a module based on the listing derived from the script will correctly describe the operations of the program image.

The ANU ML module system is implemented in file store. A module M has two files; M.sp and M.im. The M.sp file is the interface (specification) and is added to the binding environment of a session importing the module. The M.im file is the image which is translated by ANU ML import/export translation routines. A third file, M.ml, often appears and is the listing used to create the .im file.

To support the model of AIMLESS as a programming environment for the building of a module, command `make module` was developed. This command is an instance of aggregate construction discussed in section 3.1.9. Command `make module` places all specified program phrases from a session into a given module M and produces M.ml, M.sp and M.im files. The module is then imported back into the session.

An example clarifies the operation of command `make module`. With the phraselist `<fun f x = x + 1; f 7; fun g x = f x;|` the following interaction occurs:

```
- make module;
Module? test

fun f x = x + 1;
include (y/n)?:y

fun g x = f x;
include (y/n)?:y

Compilation being attempted
 [use "test.ml"]
```
Chapter 6: The Outside World

> module /users/jlo/src/lib
| val f = fn: int -> int
| val g = fn: int -> int

Command **make module** first prompts for the name of the module (here the module is *test*). A prompt is issued for each phrase that is part of the program. The user has included the definition of f and of g. The module listing is written to file test.ml. This file is compiled, producing files test.im and test.sp. The module is then imported back into the session. Any phrase not included in the module, either by the user not including it or because the phrase is not part of the program, is placed on the buffer. In this example f 7; would be placed on the buffer.

In effect, make module pops phrases onto the buffer, moves all phrases to go in the module to the top of the buffer, edits these into a module, writes a copy of the module listing to file store, compiles the module and then imports the module. The interface is not particularly sophisticated, but is appropriate for an operation used infrequently.

If the module created is invalid because the user has not included a necessary definition, an error is reported and the module placed on the buffer.

6.5. Maintaining ‘Finished’ Programs.

A ‘finished’ program is never really finished - once a program has been developed (in a logical session) it must be maintained for the rest of its life. Maintenance, which includes bug fixes and extensions, has been estimated at around 70% of the cost of software (Ramamoorthy et al 1984). Although maintenance programming is the concern of programming in the large, as mentioned in Section 5.2.7.2 tests developed in logical sessions can be retained as part of the program listing as asserts and used to aid in maintenance programming.

Asserts do not have to kept as part of the program listing. A module could consist of an interface, an image and assertions on the module’s behaviour. An alternative approach is to treat assertions as comments and store them in a fourth file, say M.as. This file would contain assertion information that could be used in
maintenance programming. Finally, a module and its logical session could be kept as the ‘finished’ program. When a module is then maintained, its logical session could be used to re-use tests and assertions.

7.1. AIMLESS As A Programming Environment

The literature suggests that a programming environment should be judged on the functionality of its tools set, the integration of these tools set, and the interaction offered to the user by these tools. It is now appropriate to evaluate AIMLESS on these grounds.

- AIMLESS has the functionality required of a programming environment. Programs can be edited, the phrase being used as the unit of change. Intra-phrase editing is done using the operations insert, delete and replace; intra-phrase editing is done textually. Aggregates can be combined and articulated. Incomplete programs can be executed, the mechanism for which (the status command) is more disciplined than in many other environments. This does not, however, affect the functionality of evaluating incomplete programs. The trace mechanism adds to the basic functionality.

- AIMLESS is integrated in that the system and all tools work together within the edit loop. The main abstraction of AIMLESS is the paradigm and all operations are based on this. The integration of the editing system and the test system means that checks are automatically built into the editing process.

- AIMLESS does not have a high bandwidth of communication. As discussed in Section 7.4, a graphical interface should be built.

- If editing is done within the last five phrases on the script then AIMLESS responds quickly. If a phrase early in the script is altered, all later phrases must be (logically) reevaluated. The algorithm described in Section 6.4 indicates...
Chapter Seven: Conclusions

Although this thesis has primarily examined issues related with programming environments based on the eval loop, the work can also be seen more generally as a programming environment *per se* and more specifically as a programming environment for ML. AIMLESS is discussed in relation to each of these three perspectives. Possibilities for future work on AIMLESS are then outlined.

7.1. AIMLESS As A Programming Environment.

The literature suggests that a programming environment should be evaluated on the functionality of its tool set, the integration within this set, and the interaction offered to the user by these tools. It is now appropriate to evaluate AIMLESS on these grounds.

- AIMLESS has the functionality required of a programming environment. Programs can be edited, the phrase being used as the unit of change. Interphrase editing is done using the operations insert, delete and replace; intra-phrase editing is done textually. Aggregates can be combined and articulated. Incomplete programs can be executed, the mechanism for which (the *defer* command) is more disciplined than in many other environments. This does not, however, affect the functionality of evaluating incomplete programs. The trace mechanism adds to the basic functionality.

- AIMLESS is integrated in that the system and all tools work together within the eval loop. The main abstraction of AIMLESS is the phraselist and all operations are based on this. The integration of the editing system and the re-test system means that checkpoints are automatically built into the editing process.

- AIMLESS does not have a high bandwidth of communication. As discussed in Section 7.4, a graphical interface should be built.

- If editing is done within the last few phrases on the script then AIMLESS responds quickly. If a phrase early in the script is altered, all later phrases must be (logically) re-evaluated. The algorithms described in Section 4.4 minimize
unnecessary computation. When a function in a large mutually recursive function
definition must be changed, the whole definition must be re-evaluated. This is
no worse than a cut and paste system that must also have the whole phrase re­
evaluated, but is not ideal. The algorithms in Section 4.4 can be adapted to apply
to one definition within a mutually recursive definition; for example, if the type
of a changed definition does not change, then other definitions in the mutually
recursive block do not have to be typechecked or re-evaluated. AIMLESS uses
the model of the phrase as the unit of change and the phraselist as a sequence of
phrases; to change individual definitions in a larger definition the model would have
to be extended to the phraselist as a sequence of phrases made up of definitions.
The user then could ask to edit one identifier's definition, even if this appeared
in a large mutually recursive definition. If the type of identifier was changed, the
phrase would still have to re-evaluated, and if the change caused an error the
whole phrase would be placed on the buffer. Note that some individual definitions
cannot be extracted from a phrase, for example the definition of a in the phrase
\[ \text{val } (a, b) = \text{EXP}; \]
as discussed in section 2.4.1.

- One of the big advantages of AIMLESS is its good execution time. Static
typechecking and the use of direct references slow down the turnaround time from
editing to executing, but make the code produced by AIMLESS for execution both
fast and compact. Most programming environments have concentrated on the
turnaround responsiveness, often using interpretive systems. While turnaround
responsiveness is important for iterative development, it is also important to be
able to make changes and to test them. Slow, unresponsive execution frustrates
and discourages the testing of changes.

- AIMLESS supports incremental program development; fragments can be
defined and tested by themselves. The automatic re-test mechanism builds on the
model of defining and testing fragments. These fragments can then be combined
into aggregates.

7.2. AIMLESS As A Programming Environment In An Eval Loop.

Chapter 1 described deficiencies of the eval loop that had to be addressed
by AIMLESS: mechanisms for editing definitions, storing and manipulating the text of phrases, handling input errors, dealing with aggregate expressions and declarations, maintaining the program listing, re-testing, and debugging. Chapter 2 described the design decisions of AIMLESS: it is based on the eval loop, editing is based on the script, the script is always valid, and the script is always consistent with the image. The challenge was to provide solutions to the deficiencies of the eval loop within the constraints of the above design decisions.

The design decisions were based on the observation that a natural method of program development in an eval loop is the construction of new program fragments from known, tested components. A local declaration, for example, is built from its hidden declarations and its exported declarations, which can be defined and tested separately, before being composed. ML is representative of modern programming languages that provide a set of language constructs for building secure, modular abstractions for problem solving. In particular, ML’s static typechecking detects many errors at compile time (while polymorphism and type inference reduce any inconvenience introduced by typechecking). Although static outer-level scoping is more efficient when compiling, its principal role in ML is to ensure good semantic properties in ML programs. Static outer-level scoping prevents definitions from changing underfoot, as a later definition to the same name will not change any earlier bindings. Aggregate constructs further emphasize modularity and information hiding.

The design decisions of AIMLESS are markedly different from Glide which dealt with equivalent language constructs. While the Glide environment addressed the problem of “how can a conventional programming environment be developed for a language like ML?”, AIMLESS addresses the problem “what style of environment (in an eval loop) would suit a language like ML?”.

To work within the design decisions AIMLESS uses the phraselist. The script represents the program under development, it is always valid, and always consistent with the image. Phrases can be edited by manipulating the phraselist (by pops and pushes), the text of phrases is stored as part of their representation. Input
Chapter 7: Conclusions

errors are placed on the buffer where they can be dealt with as required. The program listing is maintained using the script (less tests) and the text associated with those phrases. Aggregates can be combined and articulated. The re-testing mechanism is integrated into the phraselist and remembers both tests and their (eval loop) effects. A user is warned of tests with changed behaviour after an edit. The assert declaration allows for the specification of test behaviour. The trace mechanism aids in the location of errors when debugging.

The design decisions of AIMLESS, however, are based on the design decisions of ML. The question can then be asked - are such decisions appropriate for any language in an eval loop?

Perhaps the most controversial aspect of ML with respect to the eval loop is static outer-level scoping. The security of static outer-level scoping is fundamental to the concept of developing programs from known, tested fragments. With dynamic outer-level scoping definitions can change, so unless fragments are thoroughly tested after a change and contain no missing fragments, they cannot be regarded as known and tested. As well, as discussed in section 2.4.2, with dynamic outer-level scoping the script may not be consistent with the image.

If an editing system like AIMLESS is not available, dynamic outer-level scoping is more flexible than static outer-level scoping for editing definitions, as demonstrated by LISP implementations in an eval loop. It is sometimes claimed that dynamic outer-level scoping is also needed to change flags, such as a debug information flag (see discussion in smil mailing list, April 1990). This can be avoided by using reference objects that can be side-effected to be changed. Indeed there is a strong relationship between dynamic outer-level scoping and assignment. Scheme defines the binding of values in the top-level environment in terms of assignment (dynamic outer-level scoping) but the binding of lambda variables in terms of the $\lambda$-calculus (static expression scoping).

If any system is to actively support the development of secure, modular programs, the system must ensure that after a change all changed definitions are re-evaluated and all dependent tests are re-tested. A dynamic outer-level scoping
can do this by maintaining and using dependency information to keep the script consistent and re-tested as necessary. A system like AIMLESS with static outer-level scoping naturally supports these aims, and by maintaining the equivalent dependency information could avoid most unnecessary re-evaluation.

Systems with dynamic outer-level scoping do not place emphasis on the support of secure, modular programs; such systems encourage flexibility and support rapid prototyping. Static scoping and static typechecking are seen as unacceptable constraints in these systems (Sandewall 1978; Sheil 1983), although Glide demonstrated that static typechecking and rapid prototyping are not mutually exclusive. If a rapid prototyping system is preferred over one with security and modularity, then the approach of AIMLESS is not appropriate. In a rapid prototyping system the user has the freedom to change, but also must take responsibility for the behaviour of the system (by executing “sufficient” tests to ensure that run-time errors do not occur and correct results are produced). AIMLESS, on the other hand, encourages the building of secure, modular programs with the expense that after a change phrases must be re-evaluated. This overhead, however, is incurred ensuring safety and security within the system and keeping the script valid and consistent.

An alternative view rejects the idea that the eval loop develops a program listing (Nikhil 1990). The eval loop is simply used to develop a program image. Such a system does not have a consistent program listing and would more appropriately be configured as a Magpie workspace, where a program listing is maintained separately but expressions and temporary definitions could be entered and tested in the workspace. A workspace system loses the convenience of the eval loop when developing a program interactively. For example, aggregate combination and articulation would not be possible in the workspace.

Static outer-level scoping imposes a partial ordering; AIMLESS uses the script and its associated total ordering. As an abstraction for program development, the total ordering of the script is equivalent to the total ordering of a program listing. In terms of implementation, the total ordering of the script implies unnecessary
Chapter 7: Conclusions

re-evaluation on a **push**. Section 4.4. discussed algorithms for avoiding this re-evaluation. It would be interesting to explore alternative partially ordered representations of the program under development. The partial order would reflect the dependency relationships and could be used as a tool for analysing the modularity of a program when constructing aggregates.

It is interesting to think of AIMLESS in other programming domains. For example, consider a graphics system for composing images of objects in a scene. Here the partial order relates to the overlapping of images. To describe the screen (the program image), the script (the program listing) must correctly order the display of the images. The undo operation in such a system would remove an image from the display (as images would be seen as part of the system, unlike input/output in AIMLESS). If an image is re-defined it is necessary to undo images up to the altered one, change it and re-do the undone images. Again, techniques can be used to avoid unnecessary re-evaluation.

### 7.3. AIMLESS As A Programming Environment For ML.

As well as examining eval loop issues, AIMLESS has dealt with issues relating to an environment for ML. It is appropriate to ask whether AIMLESS is a better programming environment than existing ML systems.

All existing ML systems are also based on the eval loop and therefore support incremental program development, but normally use cut and paste facilities for editing. Other than recent work on a debugger (Tolmach and Appel 1990) no mechanisms exist for debugging.

As discussed in Section 2.4.2, cut and paste facilities are an attempt to overcome problems encountered when editing ML programs. Cut and paste editing represents an improvement over re-loading from files with the `use` command. To edit an existing definition, that definition is changed in the file containing it (to maintain the program listing) and it is pasted, along with all other dependent phrases, into the eval loop. Since the text of phrases is not kept in the eval loop, command `whatdefn` is not easily implemented; to find the text of a definition, its
file must be searched. Invalid phrases are discarded by the eval loop, but can then be edited in its file and re-loaded. If an invalid phrase is not kept in a file, the text must be cut and pasted and changed to make the phrase valid. In contrast, in AIMLESS the phrase is placed on the buffer and can then be edited. In a cut and paste system, when a phrase is edited and dependent phrases are then pasted into the eval loop, if an error occurs, many of the following phrases in the paste will also be in error. The first error must be fixed and phrases pasted again into the eval loop. In AIMLESS, if an error occurs on a push, the user can edit the invalid phrase on the buffer and the push is continued by the introduced push. In existing systems, aggregates cannot be manipulated as in AIMLESS. There are also technical problems dependent on windowing systems and with the ordering of definitions, as discussed in Section 2.4.2.

AIMLESS builds on its editing facilities to provide mechanisms not available in other ML systems. As the script is consistent, the program listing is maintained by AIMLESS. This can then be packaged into a module by make module or associated with files by cleanup. The AIMLESS re-test mechanism aids in managing changed tests after editing. The trace mechanism helps locate errors.

On its editing system alone, AIMLESS is more convenient and flexible than existing ML systems. With the additional advantages of the mechanisms built on top of the editing system, AIMLESS is a better programming environment for ML than existing systems.

AIMLESS is not the only possible design for a programming environment for ML. The first design decision for AIMLESS was to base the environment on the eval loop. As an expression language with static outer-level scoping, ML is suited to the eval loop which is a convenient base for a programming environment. An environment for ML could use the workspace model, but would lose the flexibility of defining, testing and building programs within the eval loop of AIMLESS.

The other design decisions - using the script, keeping the script valid and consistent - could be rejected by other ML environments. These environments would have a similar flavour to Glide. Static outer-level scoping would be used, but with
dynamic binding (indirect references) so that definitions could be changed. After a change, typechecking would be performed to detect invalid phrases, these would be converted to raise a run-time error if executed. The script would contain invalid phrases and would not be consistent. Such systems would therefore have to develop some mechanism for maintaining a program listing, probably by associating phrases to files as done with cleanup in AIMLESS. These environments either would not provide a re-test mechanism or would have to maintain dependency information (either implicitly or explicitly on request by the user).

Having function makestring implemented is essential for the two debugging mechanisms of AIMLESS: re-testing and tracing. The current implementation of makestring imposes overheads on space and execution, but could be used by other implementations. The approach discussed in Section 5.1.4.3 of makestring as an overloaded operator with a more general overloading scheme than presently available in ML is promising. If such a makestring was available, the current efficient implementation of tracing (changing the text of the traced function) could not be used, the traced function and all uses of it would have to be re-compiled for possible implicit parameters.

Experience with ML may suggest alternative implementation strategies for AIMLESS. For example, the system of undoing objects that can be side-effected makes assumptions about their usage. If the checking for changes to reference objects between phrases is found to be too expensive then checkpoints or assignment raising a flag may have to be employed instead.

7.4. Future Work.

More work remains to be done on AIMLESS. The most immediate area for work is in providing a graphical interface. This interface should display the script and the buffer and have a window for interacting in the eval loop. It should also provide the ability to select a phrase to be manipulated by pointing and clicking with the mouse. It should be possible to alter a selected phrase on the script, and to push or del a selected phrase on the buffer.
Chapter 7: Conclusions

The implementation of the re-evaluation algorithms would improve the responsiveness of AIMLESS from editing to being able to execute (i.e., the time taken to push phrases back onto the script after editing). The implementation of the algorithms must balance their cost (e.g., the space overheads of saving abstract syntax trees, the time overheads of maintaining dependency relationships) against the increased responsiveness they offer. These costs and the balances they entail, must be determined by experiment.

Experience will determine whether other tools are necessary to enhance AIMLESS. A syntax-directed editor could be used for appending and editing phrases, although the current textual editing in AIMLESS has proven convenient and will be preferred by many users. The tracing mechanism could be extended to include breakpoints. A breakpoint would interrupt a computation and allow intermediate and global values to be examined before the computation was resumed. Performance analysis tools, such as the profiling system in the New Jersey ML system, would aid in improving the performance of ML code.

Logical sessions would improve the convenience of programming in the eval loop and should be implemented. As the script is consistent with the program image, it can be used to re-generate the image.

Finally, AIMLESS has concentrated on the issues of programming in the small for ML. The wider issues of programming in the large must also be considered. Basically these deal with the question of what is done with a module once it is developed. This includes the problems of how modules are maintained and how they are co-ordinated and used in multi-module programs. The powerful module system now part of standard ML should prove a fertile testbed for ideas.
Appendix 1: ANU ML

ANU ML does not currently implement standard ML. This appendix outlines the areas where ANU ML is not standard ML and then describes the existing implementation of modules. See Cardelli et al (1991) for details, the descriptions below quote and paraphrase this source.

1. Non-standard Parts Of ANU ML.

- The withtype variant of type declarations is not implemented.
- Side-effects by a definition raising an exception are undone. This is done to keep the script consistent.
- Weak types are allowed at the top-level.
- Weak types not correctly implemented for exceptions.
- The module system is non-standard.

2. The Module System.

The ANU ML module system is very simple compared with the sophisticated module system of standard ML. The module system of ANU ML allows for separately compiled bindings.

A module is a set of bindings (values, functions and types). Modules can be compiled and later imported as a declaration. Modules are identified by module names, either simple identifiers or strings containing Unix+ file paths.

Module definitions can only appear at the top-level, and they cannot rely on bindings defined in the surrounding scope. All bindings used by a module must either be defined in the module itself, or imported from other modules.

The processing of a module definition is called a module compilation. Module definitions do not evaluate to values, and they do not introduce new bindings.

+ Unix is a registered trademark of AT&T Bell Laboratories
Appendix 1: ANU ML

Modules can be defined interactively, but usually they are contained in external source files. In the latter case, command use "Mod.ml"; can be used to compile a module definition obtained in the file Mod.ml. Once a module is compiled it can be imported using the import command. Import takes a module name and adds the module bindings to the binding environment.

Importing can result in loading a module, when that module is imported for the first time in an interactive session, or in sharing an already loaded module, all the subsequent times. The module body is evaluated when the module is loaded. If the module body contains side-effecting operations (such as input/output), they have effect at loading time, i.e., they do not have effect if the module is being shared.
Appendix 2: A User's Guide to AIMLESS

1. Introduction.

AIMLESS, An Interactive ML Editing Support System, is an editing system and programming environment for ANU ML (Cardelli et al 1991). AIMLESS provides a user interface for interactively entering and debugging ML programs. For an introduction to AIMLESS see Ophel (1988).

2. AIMLESS.

On starting AIMLESS a short message is printed followed by the interactive prompt ‘>’. Phrases may then be entered. AIMLESS works in a read-eval-print loop. The user enters a phrase, the phrase is compiled and executed, and the result then displayed. If the phrase is a definition, the result is a set of bindings; the bindings returned are prefaced, for the first line, by ‘>’, and by ‘|’ thereafter.

Valid phrases are evaluated and added to the script. Invalid phrases are added to the buffer. The script and buffer form the phrases list. The phrases list is denoted by \(<p_1,\ldots,p_i,p_{i+1},\ldots,p_n>\), with \(<p_1,\ldots,p_i>\) the script and \(<p_{i+1},\ldots,p_n>\) the buffer. Phrases are added to the right of the script but to the left of the buffer. That is, phrase \(p_2\) was added after \(p_1\) on the script while phrase \(p_{i+1}\) was added after \(p_{i+2}\) on the buffer. The top-level environment and the store consist of the bindings and the side-effects respectively of the phrases on the script. The script contains those phrases compiled and executed by the evaluator. The buffer contains phrases entered, but not evaluated and therefore not part of the top-level environment or the store.

For example, with the initially empty phrases list \(<|>\), if the user enters the phrase \(\text{val x} = 10;\) it is compiled and executed and the phrases list is \(<\text{val x} = 10;|>\). If the phrase \(\text{val a} == x;\) is then entered, it is invalid and added to the phrases list. The phrases list would then be \(<\text{val x} = 10;\text{val a} == x;>\).

Definitions that raise a top-level exception are considered to be invalid and are therefore placed on the buffer. An expression raising a top-level exception is however considered valid and added to the script.
Continuing the previous example with phraselist of `<val x = 10; |val a == x; >`, if the phrase `val x = 10 div 0;` is entered it raises a top-level div exception. The phrase is added to the buffer. The phraselist would be `<val x = 10; |val x = 10 div 0; |val a == x; >`. Executing phrase `10 div 0;` also raises a top-level div exception but as the phrase is an expression it is placed on the script. The phraselist would then be `<val x = 10; , 10 div 0; |val x = 10 div 0; , |val a == x; >`.

When an invalid phrase is added to the buffer, the phrase and the rest of input is put as one phrase on the buffer. For example, if the phrase `val x = y; val z = x;` is entered with `y` undefined in the top-level environment, the ML phrase `val x = y;` will be invalid. The whole phrase `val x = y; val z = x;` is placed on the buffer.

3. The AIMLESS Editing Commands.

The AIMLESS editing commands manipulate the phraselist. Commands shown to take integer argument `n`, default to 1 if no argument is given.

`reset`: Command `reset` restores AIMLESS to its initial state. The phraselist is empty.

`defer <ide> {:<type>};`: Command `defer` adds a temporary function definition for its specified identifier. The deferred function can then by mentioned by later phrases, but the application of a deferred function at run time is an error. If the optional type constraint is not given, the function has type `'a -> 'b`.

`pop n`: Command `pop` moves phrases from the script onto the buffer. All bindings added to the top-level environment and store side-effects caused by popped phrases are undone. No error message is reported if `n` is greater than the number of phrases on the script. `pop all;` moves the whole script onto the buffer.

`push n`: Command `push` attempts to move `n` phrases from the buffer onto the script. If a phrase is invalid, the phrase remains on the buffer and the command stops. If more pushes remain on an error, an introduced push is added to the invalid
phrase. No error message is reported if \( n \) is greater than the number of phrases on the buffer. **push all**; attempts to push the whole buffer onto the script.

**del** \( n \); : Command **del** deletes the first \( n \) phrases on the buffer. No error message is reported if \( n \) is greater than the number of phrases on the buffer. **del all**; deletes the whole buffer.

**edit** \( n \); : Command **edit** textually appends the first \( n \) phrases on the buffer, and places the resulting text in a text editor. In AIMLESS on Unix, the text editor is specified by the shell variable **EDITOR**; the default editor is **vi**. After editing the phrase is pushed. No error message is reported if \( n \) is greater than the number of phrases on the buffer. **edit all**; textually appends the whole buffer.

**move** \( k \); : Command **move** \( k \); takes an integer argument \( k \) and puts the \( k^{th} \) phrase in the buffer as the first phrase on the buffer. If \( k \) is not in range an error is reported.

**alter** \(<ide>\); : Command **alter** \(<ide>\) takes an identifier argument \( ide \), and finds the most recent phrase that defines \(<ide>\) in a val or fun definition. If no phrase is found an error is reported. If a phrase is found, it is moved onto the buffer by successive **pops**, edited (in the same manner as command **edit**) and then all phrases popped by this command are pushed. If a pushed phrase is invalid, the pushing stops.

**altertype** \(<ide>\); : Command **altertype** \(<ide>\) takes an identifier argument \( ide \), and finds the most recent phrase that defines \(<ide>\) in a type, datatype or abstype definition. If no phrase is found an error is reported. If a phrase is found, it is moved onto the buffer by successive **pops**, edited (in the same manner as command **edit**) and then all phrases popped by this command are pushed. If a pushed phrase is invalid, the pushing stops.

**make** \([let | local | abstype]\); : Command **make** prompts the user for phrases to include in an aggregate form specified by its argument.

**unmake** : Command **unmake** articulates the most recent phrase on the
script, which must be an aggregate form.

\texttt{show [script | buffer | all | program];} : Command \texttt{show} lists phrases specified by its argument.

\texttt{whatdefn <ide>;} : Command \texttt{whatdefn} takes an identifier \texttt{<ide>} and displays the most recent phrase on the script defining \texttt{<ide>} in a \texttt{val} or \texttt{fun} definition.

\texttt{whattype <ide>;} : Command \texttt{whattype} takes an identifier \texttt{<ide>} and displays the most recent phrase on the script defining \texttt{<ide>} in a \texttt{type}, \texttt{datatype} or \texttt{abstype} definition.

4. The AIMLESS File Operations.

The AIMLESS file operations load or save parts of the phraselist from or to file store.

\texttt{use "<filename>";} : Command \texttt{use} redirects input from the specified file until end of file or an invalid phrase.

\texttt{save [script | buffer | all | program | "<filename>";} : Command \texttt{save} writes out phrases specified by its argument.

\texttt{cleanup [script | buffer | all | program | {prompt}];} : Command \texttt{cleanup} writes phrases specified by its argument to their associated file. If the optional argument \texttt{prompt} is given, every phrase is prompted for (so the user can change associations if desired).

\texttt{make module;} : Command \texttt{make module} prompts for module name and then packages up specified phrases into a module. The module is compiled and then imported.

5. The AIMLESS System Operation.

Two functions are provided for interfacing with the Unix environment.
**system** "<Unix command>"; : Command **system** takes a string and executes the string as a Unix command. For example,

```ML
-fun vi str = system "vi "^str;
>val vi = fn: string -> ()
```
defines a function **vi** that takes a string name of a file and then calls the Unix command to edit that file.

**systemquery** "<shell variable>"; : Command **systemquery** takes a shell variable as a string and returns the string bound to that variable in the Unix shell environment. If the variable is not bound exception **systemquery** is raised. For example,

```ML
-systemquery "HOME";
"/users/jlo" : string
```

where the variable **HOME** is bound to /users/jlo in the current shell environment.

6. **The AIMLESS Re-test Mechanism.**

The AIMLESS re-test mechanism provides feedback on changed test (top-level expression) evaluation. Any expression popped onto the buffer and later pushed back prints a warning message if its previous value is not equal to its current value.

For example,

```ML
-fun f x = x;
>val f = fn: 'a -> 'a
-f 10;
10 : int
-alter f;

(* edit f to be fun f x = x + 1; *)

>val f = fn: int -> int
**********************
test value changed
Test Phrase: (f 10)
Old: 10: int
```
New: 11: int

**********************

Assertions can be placed on the script. An assertion has the form

```
assert <expression>;
```

The expression must be of type boolean. If an assertion is appended or pushed
and is not true, it is in error and placed on the buffer.

```
make assert; : Command make assert converts the most recent phrase on
the script from a test into an assert. make assert takes a test t and its value v
and generates the assertion assert t = v; The conversion fails if the test and its
values cannot be compared by equals, for example if the test returns a closure or
an abstract data type.
```

7. The AIMLESS Trace Operation.

The AIMLESS trace commands allow function calls to be traced. The function
and its evaluated arguments are displayed on entry. On exit, the result of the
application is displayed.

```
trace <ide-list>; : Command trace sets the trace flag on the definition
of each identifier in <ide-list>. The definition must be a fun definition or a
structured val definition of the form val <ide> = fn PAT => EXP; It can
appear in an abstype or local declaration but not a module. Note that editing
a function definition by edit or alter does not remove the trace flag. The trace
command will fail if identifiers std_out, output, ^ or makestring are re-defined.
Note that identifier ^ is now an ML primitive and should not be defined in the
standard library (as was done previously) or else functions cannot be traced.
```

```
untrace <ide-list>; : Command untrace resets the trace flag on the defi-
nition of each identifier in <ide-list>.
```

```
importtrace <module>; : Command importtrace is equivalent to an imp-
port command except that any functional form imported can now be traced. Note
that popping the importtrace phrase resets any trace flags set.
```

140
8. Using ML.

The ML system takes two arguments: the library and a boot file. Both are compulsory and must be full path names. The text editor is specified by the shell variable EDITOR.

9. Files Used.

AIMLESS creates files in the /usr/tmp area. In the case of an extraordinary exit, these can be removed by `rm /usr/tmp/ml*`.

10. Errors.

An `edit` or `push` that pushes a phrase with an `edit` or a `push` followed by more text will lose this text.
Appendix 3: An Example AIMLESS Session

Appendix 3 is an example AIMLESS session, with an insertion sort being developed. Lines prefaced by ## are comments on the session, and are not part of the session.

AIMLESS: An Interactive ML Editing Support System

- fun isort [] = []
  | isort (x::y) = insert (x,isort y);
  Unbound Identifier: insert

## The user enters a definition of insertion sort but has not defined function insert. The phrase is invalid, and placed on the buffer.

- defer insert: 'a * 'a list -> 'a list;
> val insert = fn : 'a * 'a list -> 'a list

## The definition of insert is deferred.

- push;
push phrase: fun isort [] = [] | isort (x::y) = insert (...)
> val isort = fn : ('a list) -> ('b list)

## The definition of isort is pushed onto the script and compiles successfully.

- isort [];
  [] : 'a list

## Function isort is tested on the nil case.

- isort [1,3,2];
** Function insert is not defined **

## The expression isort [1,3,2] cannot be evaluated as insert is not defined. The expression is placed on the buffer.
Appendix 3: Example Session

- show all;

** insert is not defined **

fun isort [] = []
  | isort (x::y) = insert (x,isort y);

isort [];

******************************
isort [1,3,2];

******************************

## Command show all displays the phraselist. The script (above
## the line of asterisks) contains three phrases, the buffer
## (below the line) contains one phrase.

- pop 3;
pop phrase: isort [];
pop phrase: fun isort [] = [] | isort (x::y) = insert (... pop phrase: ** insert is not defined **

## The user could have used the alter command to change the
## deferred definition of insert. Instead, to illustrate
## AIMLESS, the phrase is popped, deleted and a new definition
## appended.

- del;

## Delete the deferred definition of insert.

- fun insert (n,[]) = [n]
  | insert (n,1 as x :: y =
  Syntax Error: ... fun insert (n,[]) = [n] | insert (n,1 ... I was expecting a "")

## A definition for insert is entered but contains a syntax
## error. The phrase is invalid and placed on the buffer.

- show all;

******************************

fun insert (n,[]) = [n]
  | insert (n,1 as x :: y =
fun isort [] = []
  | isort (x::y) = insert (x,isort y);
isort [];
isort [1,3,2];

******************************
## The definition of insert is edited. It is valid and it added to the script.

> val insert = fn : (int * (int list)) -> (int list)

- show all;

```
fun insert (n,[]) = [n]
| insert (n,l as x::y) =
    if (n>=x) then n::l
    else x::(insert(n,y));
```

## The definition of insert is on the script.

- insert (3,[]);
  
  [3] : int list

## The definition of insert is tested and returns the expected result.

- push;
  push phrase: fun isort [] = [] | isort (x::y) = insert ...
  > val isort = fn : (int list) -> (int list)

## The definition of isort is pushed. It is valid and added to the script.

- push 2;
  push phrase: isort [];
  [] : int list

  push phrase: isort [1,3,2];
  [3,2,1] : int list

## Function isort is tested. Expression isort [] returns the correct answer but expression isort [1,3,2] returns the list in the wrong order!
fun insert (n,[]) = [n]
  | insert (n,l as x::y) = 
     if (n>=x) then n::(insert(n,y))
     else x::(insert(n,y));
insert (3,[]);
fun isort [] = []
  | isort (x::y) = insert (x,isort y);
isort [];
isort [1,3,2];
*****************************
- alter insert;
## The definition of insert is edited. It is valid, so it is
## added to the script. As part of the alter command, the
## phrases popped so that function insert could be edited are
## now pushed.
> val insert = fn : (int * (int list)) -> (int list)
*****************************
test value changed
Test Phrase: (isort (1 :: (3 :: (2 :: nil))))
Old: [3,2,1] : int list
New: [1,2,3] : int list
*****************************

## The re-test mechanism warns that the test isort [1,3,2] now
## behaves differently (but now correctly).

- show all;
---------------------------------------
fun insert (n,[]) = [n]
  | insert (n,l as x::y) = 
     if (n>=x) then n::(insert(n,y))
     else x::(insert(n,y));
insert (3,[]);
fun isort [] = []
  | isort (x::y) = insert (x,isort y);
isort [];
isort [1,3,2];
*****************************
- show program;
fun insert (n,[]) = [n]
  | insert (n,l as x::y) =
      if (n<=x) then n::l
      else x::(insert(n,y));

fun isort [] = []
  | isort (x::y) = insert (x,isort y);
## Note that the tests are not included in the program.

- make local;

isort [1,3,2];
include as exported decl (y/n/q)?:n

isort [];
include as exported decl (y/n/q)?:n

fun isort [] = []
  | isort (x::y) = insert (x,isort y);
include as exported decl (y/n/q)?:y

insert (3,[]);
include as exported decl (y/n/q)?:q
include as hidden decl (y/n/q)?:n

fun insert (n,[]) = [n]
  | insert (n,l as x::y) =
      if (n<=x) then n::l
      else x::(insert(n,y));
include as hidden decl (y/n/q)?:y
> val isort = fn : (int list) -> (int list)

- show all;
---------------------------------------
local
  fun insert (n,[]) = [n]
  | insert (n,l as x::y) =
      if (n<=x) then n::l
      else x::(insert(n,y))
in
  fun isort [] = []
  | isort (x::y) = insert (x,isort y)
end;

***************************************
\begin{verbatim}
insert (3,[]);
isort [];
isort [1,3,2];
------------------------------------------------------------------------
- save program "isort.ml";

## The program is saved in file isort.ml. Alternatively, 
## CleanUp could have been used.
- "D

## The session is terminated.
\end{verbatim}
Appendix 4 gives formal definitions of the AIMLESS commands, using the notation developed in Section 3.2. Function `aim_command` takes a command, an environment sequence, a state sequence and a phraselist and evaluates the command, returning a status and a new environment sequence, state sequence and phraselist.

1. Reset.

   ```
   aim_command(reset; ES, S, [p₁, ..., pᵢ|pᵢ₊₁, ..., pₙ]) = (ok, < E₀, < S₀ >, < | >);
   ```

   Description: Command `reset` restores the binding environment to its initial bindings, the store to its initial state and the phraselist to the empty phraselist.

2. Defer.

Function `make_deferred_definition` takes an identifier and returns a deferred definition phrase and a binding environment.

   ```
   aim_command(defer ide; < E₀, ..., Eᵢ, < S₀, ..., Sᵢ >, < p₁, ..., pᵢ|pᵢ₊₁, ..., pₙ >) =
   let
     val (d, E) = make_deferred_definition(ide)
   in
     (ok, < E₀, ..., Eᵢ, Eᵢ + E >, < S₀, ..., Sᵢ, Sᵢ >, < p₁, ..., pᵢ, d|pᵢ₊₁, ..., pₙ >)
   end;
   ```

   ```
   aim_command(defer ide, ide-list; < E₀, ..., Eᵢ, < S₀, ..., Sᵢ >, < p₁, ..., pᵢ|pᵢ₊₁, ..., pₙ >) =
   let
     val (d, E) = make_deferred_definition(ide)
   in
     aim_command(defer ide-list, < E₀, ..., Eᵢ, Eᵢ + E >, < S₀, ..., Sᵢ, Sᵢ >, < p₁, ..., pᵢ, d|pᵢ₊₁, ..., pₙ >)
   end;
   ```

   Description: The first `defer` clause handles the base case of one identifier in the identifier list. A deferred definition is added to the phraselist and to the binding environment. For a list of identifiers, each identifier has a deferred definition added in turn.
3. Pop.

\[
\text{aim\_command}(\text{pop } k; , < E_0, \ldots, E_i >, < S_0, \ldots, S_i >, < p_1, \ldots, p_i|p_{i+1}, \ldots, p_n >) =
\]
\[
\begin{cases}
\text{if } (k \geq i) & (\text{ok}, < E_0, \ldots, E_i >, < S_0, \ldots, S_i >, < p_1, \ldots, p_i|p_{i+1}, \ldots, p_n >) \\
\text{else } (\text{ok}, < E_0, \ldots, E_{i-k} >, < S_0, \ldots, S_{i-k} >, < p_1, \ldots, p_{i-k}|p_{i-k+1}, \ldots, p_n >)
\end{cases}
\]

Description: The \textit{pop} command either moves all the script onto the buffer or moves the top \( k \) phrases onto the buffer. The binding environment entry for each popped phrase is removed from the binding environment sequence. The store entry for each popped phrase is removed from the store sequence.


\[
\text{aim\_command}(\text{push } i; , ES, S, PL \text{ as } < p_1, \ldots, p_i >) = (\text{ok}, ES, S, PL);
\]
\[
\text{aim\_command}(\text{push } 1; , ES, S, < p_1, \ldots, p_i|p_{i+1}, \ldots, p_n >) = \\
\text{aim}(p_{i+1}, ES, S, < p_1, \ldots, p_i|p_{i+2}, \ldots, p_n >);
\]
\[
\text{aim\_command}(\text{push } k; , ES, S, < p_1, \ldots, p_i|p_{i+1}, \ldots, p_n >) = \\
\text{aim\_command}(\text{push}; \text{ push } (k-1); , ES', S', PL');
\]

Description: The first \textit{push} command clause is the case where there are no phrases on the buffer. The status code \textit{ok} is returned with the existing environment sequence, store sequence and phraselist. The next clause pushes the top phrase on the buffer. The final clause handles the multiple push. If the next \textit{push} is successful (i.e., the pushed phrase ends up on the script) then the rest of the \textit{pushes} are attempted. If the next \textit{push} is not successful the introduced \textit{push} is placed on the buffer and the multiple \textit{push} stops at this point.

5. Del.

\[
\text{aim\_command}(\text{del } k; , ES, S, < p_1, \ldots, p_i|p_{i+1}, \ldots, p_n >) =
\]
\[
\begin{cases}
\text{if } (k \geq (n-i)) & (\text{ok}, ES, S, < p_1, \ldots, p_i >) \\
\text{else } (\text{ok}, ES, S, < p_1, \ldots, p_i|p_{i+1+k}, \ldots, p_n >)
\end{cases}
\]

Description: As with the \textit{pop} command the \textit{del} command has no problems with unsuccessful operations, and the only test is to ensure that phrases exist on the buffer to delete.
Appendix 4: Formal Definitions


Function \textit{text-edit} takes a list of AIMLESS phrases, and returns the result of a textual edit of the concatenation of these.

\[
\text{aim-command}(\text{edit } k; ES, S, < p_1, \ldots, p_i, p_{i+1}, \ldots, p_n >) = \\
\begin{cases} 
\text{if } (k > (n-i)) \\
\text{then let} \\
\quad \text{val } p = \text{text-edit}(p_{i+1}, \ldots, p_n) \\
\quad \text{in} \\
\quad \text{aim}(p, ES, S, < p_1, \ldots, p_i >) \\
\text{end} \\
\text{else let} \\
\quad \text{val } p = \text{text-edit}(p_{i+1}, \ldots, p_{i+k}) \\
\quad \text{in} \\
\quad \text{aim}(p, ES, S, < p_1, \ldots, p_i | p_{i+1+k}, \ldots, p_n >) \\
\end{cases}
\]

Description: The \textit{edit} command does a textual edit of specified phrases on the buffer and then treats the result of the edit as the AIMLESS phrase input.

7. Move.

\[
\text{aim-command}(\text{move } k; ES, S, PL \text{ as } < p_1, \ldots, p_i, p_{i+1}, \ldots, p_{i+k-1}, p_{i+k}, p_{i+k+1}, \ldots, p_n >) = \\
\begin{cases} 
\text{if } (k < 1) \text{ or } (k > (n-i)) \\
\text{then (not-ok, ES, S, PL)} \\
\text{else (ok, ES, S, < p_1, \ldots, p_i | p_{i+k}, p_{i+1}, \ldots, p_{i+k-1}, p_{i+k+1}, \ldots, p_n >)} \\
\end{cases}
\]

Description: The \textit{move} command manipulates the buffer. If the argument is out of range, no manipulation occurs and an error is signalled.

8. Alter and AlterType.

Function \textit{defined-val} takes an identifier and an environment, and returns true if the identifier has a value binding in the environment, false otherwise. Function \textit{first_defined_val} takes an identifier, an environment sequence and a phraselist, and returns the most recent phrase on the script that defines a value binding for the identifier. Functions \textit{defined_type} and \textit{first_defined_type} act similarly, except over type bindings.
Appendix 4: Formal Definitions

\[
\text{aim\_command(alt\_ide; ES as } < E_0, \ldots, E_i >, S as < S_0, \ldots, S_i >, \\
PL as < p_1, \ldots, p_j, \ldots, p_i | p_{i+1}, \ldots, p_n >) = \\
\text{if defined\_val(ide, }E_i) \\
\text{then let} \\
\text{val } p_j = \text{first\_defined\_val(ide, }E_i, PL); \\
\text{val } p' = \text{text\_edit}(p_j) \\
in \\
\text{aim\_command(push (i-j+1);, }< E_0, \ldots, E_{j-1} >, < S_0, \ldots, S_{j-1} >, \\
< p_1, \ldots, p_{j-1} | p', p_{j+1}, \ldots, p_n >) \\
\text{end} \\
\text{else (not\_ok, }ES, S, PL); \\
\]

\[
\text{aim\_command(alt\_type\_ide; ES as } < E_0, \ldots, E_i >, \\
S as < S_0, \ldots, S_i >, \\
PL as < p_1, \ldots, p_j, \ldots, p_i | p_{i+1}, \ldots, p_n >) = \\
\text{if defined\_type(ide, }E_i) \\
\text{then let} \\
\text{val } p_j = \text{first\_defined\_type(ide, }E_i, PL); \\
\text{val } p' = \text{text\_edit}(p_j) \\
in \\
\text{aim\_command(push (i-j+1);, }< E_0, \ldots, E_{j-1} >, < S_0, \ldots, S_{j-1} >, \\
< p_1, \ldots, p_{j-1} | p', p_{j+1}, \ldots, p_n >) \\
\text{end} \\
\text{else (not\_ok, }ES, S, PL); \\
\]

Description: Commands \texttt{alter} and \texttt{altertype}. locate a phrase on the script, popping all phrases above it, edit the phrase and attempt to push the popped phrases back onto the script.
Appendix 5: Implementing The AIMLESS Commands

Appendix 5: Implementing the AIMLESS Editing Commands.

A basic outline of the implementation of the AIMLESS editing commands is given using the mechanisms described in Section 4.3.

The push and edit operations use a file when appending. The text of the phrase is placed in a file, TextFile, and TextFile put on the file stack. ScanChar then reads TextFile as it does for the use command.

1. Append.

The process of appending a valid phrase is straight-forward. When the phrase has successfully compiled and executed, the binding environment and the argument stack will have been incremented by the system. An entry with text, syntax tree, fixity information, weak type information, side-effect information and it information is added to the top of the script and all new entries on the binding environment have their Phrase pointer set to the new phrase.

An invalid phrase has its text placed on the buffer.

2. Reset.

procedure Reset;
begin
  set script to nil
  set buffer to nil
  set top-level binding environment to original environment
  set argument stack to original height
  set fixity environment to original environment
  set weak type list to original list
  set side-effect change stack to nil
end

Reset sets the phraselist to empty and resets the top-level binding environment, the argument stack, the fixity environment and the weak type list to their initial values. The side-effect changed stack is set to nil.

3. Defer ide-list.

procedure Defer(list: ide-list);
Appendix 5: Implementing The AIMLESS Commands

begin
while (list not empty) do
begin
ide:=first entry in list
build AST of form
fun ide x = _undefined "function <ide> not defined"
set text to be "** function <ide> not defined **"
pass AST to analysis and rest of evaluation
list:=rest of list
end
end

Defer builds an AST for each identifier. The AST is then passed to the analyser, bypassing the parser, and is evaluated. The AST and the text are put on the phrase entry on the script. The closure bound to ide uses function _undefined. Function _undefined can only be used internally (as it is not a legal identifier). _undefined raises an untrappable 'exception' which is reported at the top-level.

4. Pop n.

procedure PopPhrase;
begin
if (script is empty) then no action
else begin
move phrase from script to buffer
using OldFixityEnv reset fixities defined by phrase
using WeakObjectPtr reset weak types
using HowMany and ArrayHowMany reset store
while (binding environment points to popped phrase) do
begin
remove entry from binding environment
EP and AP are decremented by one
end
end
end

procedure Pop(n: integer);
begin
if n not given then n:=1
while (script not empty) and (n<>0) do
begin
PopPhrase;
n:=n-1
end
end
Appendix 5: Implementing The AIMLESS Commands

PopPhrase moves one phrase from the script onto the buffer. The effects of the phrase are undone, as described in Section 4.3. All entries in the binding environment defined by the popped phrase (i.e., all entries with their Phrase field pointing to the popped phrase) are removed from the binding environment. The corresponding entry on the argument stack is removed by decrementing EP and AP.

Pop implements correctly the cases where its argument is not given and its argument is greater than the number of phrases on the script.

5. Push n.

function PushPhrase: returns status;
begin
  if (buffer is empty) then status:=successful evaluation
  else begin
    reset TextFile
    put text of first phrase on buffer into TextFile
    set up input to read from TextFile
    take entry off buffer
    repeat
      status:=ExecuteOnce
      until ((TextFile read) or
        (status = pre-runtime error or hard exception))
  end
  return(status)
end

procedure Push(n: integer);
begin
  if n not given then n:=1
  (* initialize for while loop*)
  status:=successful evaluation
  while ((status = successful evaluation or soft exception)
    and (buffer not empty) and (n<>0)) do
    begin
      PushesLeft:=(n-1)
      status:=PushPhrase
      n:=n-1
    end
end

PushPhrase attempts to push the next phrase on the buffer onto the script. The phrase may consist of several ML phrases and AIMLESS commands so the phrase
Appendix 5: Implementing The AIMLESS Commands

is read until the whole phrase is read (the TextFile is empty) or a phrase is invalid. The use of ExecuteOnce can be seen; soft exceptions can be handled appropriately.

Push calls PushPhrase until a push fails or the required number of pushes has been done. If a push fails, the value in PushesLeft is used for the introduced push. As pushes may be nested (a push may execute another push) PushesLeft is actually a value associated with each TextFile used to perform the push.

6. Del n.

procedure Del(n: integer);
begin
  while (n<>0) and (buffer not empty) do
  begin
    take phrase off buffer
    n:=n-1
  end
end

Del does not fail so status codes are not involved.

7. Edit n.

procedure Edit(n: integer);
begin
  if (buffer is empty) then no action
  else begin
    if n not given then n:=1
    reset TextFile
    while (n<>0) and (buffer not empty) do
    begin
      append text of phrase on buffer to TextFile
      remove phrase from buffer
      n:=n-1
    end
  do text edit on TextFile
  set up input to read from TextFile
repeat
  status:=ExecuteOnce
  until ((TextFile read) or
         (status = pre-runtime error or hard exception))
end

The function Edit has three stages: putting text into TextFile, doing the text edit, and pushing TextFile. The text is put in TextFile using the argument in a
Appendix 5: Implementing The AIMLESS Commands

straight-forward way. The actual text edit is done by either using a system call to an existing editor (for example, in Unix a shell is executed with an edit command) or by providing an internal text editor. The result of the edit is pushed in the same manner as PushPhrase.

Note that the system has the limitation that if the edit phrase is an edit or a push followed by more text, this text will be lost as the edit or push resets the TextFile. This limitation can be fixed by using a new file for each nested edit or push.

8. Move n.

procedure Move(n: integer);
begin
  if (n<1) or (n>number of phrases on buffer) then error
  remove nth phrase on buffer
  add it to top of buffer
end

Move is straight-forward linked list manipulation.


procedure Alter(ide: identifier);
begin
  search binding environment for the most recent value entry defining ide
  if no entry found then error
  set EditPhrase to phrase defining environment entry
  set TopPhrase to current first phrase on buffer
  (this is what is pushed up to after the edit)
  while (first phrase on buffer<>EditPhrase) do PopPhrase
  status:=Edit
  while ((status=successful evaluation or soft exception)
        and (first phrase on buffer<>TopBuffer)) do
    status:=PushPhrase
end;

Alter is a macro that does the popping, editing and pushing required to edit an existing value definition on the script. Once the various positions to pop to and to push back to are defined, the operations PopPhrase, Edit and PushPhrase do the work.
10. AlterType ide.

AlterType is the same operation as Alter except the binding environment is searched for the most recent type entry defining ide.

11. Show Script.

procedure ShowScript;
begin
  from StartScript To Script display phrase’s text form
end

The script is displayed from oldest to most recent.

12. Show Buffer.

procedure ShowBuffer;
begin
  from Buffer To EndBuffer display phrase’s text form
end

The buffer is displayed from most recent to oldest.

13. Show All.

procedure ShowAll;
begin
  ShowScript
  print a separating line
  ShowBuffer
end

The script is shown, then the buffer.

14. Show Program.

function PartOfProgram(List: PhraseEntryRefT) : returns boolean;
begin
  if (List is empty) then return(false)
  else begin
    entry:=next entry on List
    if (entry^.DependsOnIt)
      then begin
        if entry is a top-level expression
           and does not cause a side-effect
        then return(PartOfProgram(next in List))
        else return(true)
Appendix 5: Implementing The AIMLESS Commands

```pascal
end
else if entry.IsItExpression then return(false)
else return(PartOfProgram(next in List))
end
end

function IsPartOfProgram(List: PhraseEntryRefT)
: returns boolean;
begin
  entry:=next entry on List
  if not ItExpn(entry) then return(true)
  if causes side-effect(entry) then return(true)
  if PartOfProgram(rest of List) then return(true)
  return(false)
end

procedure ShowProgram;
begin
  for each entry E on script (from StartScript to Script)
    if IsPartOfProgram(E) then display E
end
```

ShowProgram is very simple. If a phrase on the script is part of the program it is printed. IsPartOfProgram is true if the current phrase is not a top-level expression, or if it is a top-level expression if it causes a side-effect or is used by another phrase that is part of the program.

PartOfProgram looks at the rest of the script from the top-level expression in question. It checks to see if the binding to identifier it made by the phrase is used by another phrase that is part of the program itself.

15. WhatDefn ide.

```pascal
procedure WhatDefn(ide: identifier);
begin
  search binding environment for the most recent value
  entry defining ide
  if no entry found then error
  display text of phrase defining entry
end
```

16. WhatType ide.

WhatType is the same operation as WhatDefn except the binding environment is searched for the most recent type entry defining ide.
17. Make Let.

```pascal
procedure MakeLet;
begin
  keep most recent phrase on script as expression
  prompt phrases on script, keep track of declarations for
  let expression
  pop phrases onto buffer, if phrase is declaration, take
  it off buffer and put it in temporary declaration list
  write "let"
  write_indented declarations
    (remove last semi-colon if present)
  write "in"
  write_indented expression
    (remove trailing semi-colon if present)
  write "end"
end
```

MakeLet is fairly straight-forward manipulation of the phraselist. Each new
line in the declarations and expression in the let aggregate are indented two spaces.

18. Make Local.

Very similar to MakeLet except must keep two lists, one for local declarations
and one for exported declarations.

19. Make Abstype.

Again very similar to MakeLet except that keyword `datatype` must be con-
verted into `abstype`. This is done textually.

20. UnMake Let.

```pascal
procedure UnMakeLet;
begin
  move most recent phrase onto buffer
  extract components of this phrase by parsing (must be
  careful of lookahead if components do not have
  separating semi-colon)
end
```

The articulation of the aggregate is done by parsing and not including the text
of `let`, `in` or `end`, and by treating each component phrase as a separate phrase on
the script.
21. **UnMake Local.**

UnMakeLocal is similar to UnMakeLet.

22. **UnMake Abstype.**

UnMakeAbstype is similar to UnMakeLet, the keyword `abstype` is parsed and replaced by `datatype`.

A datatype such as

```plaintext
data type = false | true;
```

is now represented in ANU ML as two distinct types (`false` and `true`) instead of the more efficient constant representation of `false` as 0 and `true` as 1. The overhead of this representation can be sensed as the ANU ML boolean miniptypes are still implementational as constants.

The efficient implementation:

```plaintext
fun not type = false
 | not false = true;
```

The current implementation:

```plaintext
data type = false | true;
```

```plaintext
fun not type = true
 | not false = false;
```

The paper in the first case was 90 bytes long and took 1.32 seconds to execute (averaged over 20 runs). In the second case the object code was 216 bytes long and took 184.53 seconds to execute (again averaged over 20 runs). This represents a 45% increase in object code size and 173% increase in time to execute.

160
Appendix 6: Measuring Makestring Overhead

Appendix 6: Measuring The Overhead Of Makestring In ANU ML.

Appendix 6 outlines testing done to measure the overhead of implementing `makestring` in ANU ML as described in Chapter 5. The tests sought to determine the overheads for code using constants and transparents.

The tests were executed a large number of times using function `repeat`.

```plaintext
fun repeat 0 f e = e
    | repeat n f e = repeat (n-1) f (f e);
```

1. Constants.

A datatype such as

```plaintext
datatype Boolean = False | True;
```

is now represented in ANU ML as two variant cells (with string fields for "False" and "True") instead of the more efficient constant representation of False as 0 and True as 1. The overhead of this representation can be tested as the ANU ML boolean primitives are still implemented as constants.

The efficient implementation -

```plaintext
fun not true = false
    | not false = true;

repeat 100000 not true;
```

The current implementation -

```plaintext
datatype Boolean = False | True;

fun myNot True = False
    | myNot False = True;

repeat 100000 myNot True;
```

The object code in the first case was 151 bytes and took 166.30 seconds to execute (averaged over 25 runs). In the second case the object code was 219 bytes and took 194.10 seconds to execute (again averaged over 25 runs). This represents a 45% increase in object code size and 16% increase in time to execute.
2. Transparents.

A datatype such as

```haskell
datatype 'a list = nil | :: of 'a * 'a list
```

is now represented as two variant cells, one for nil, the other pointing to a record cell, the body of ::. Previously this would be represented as a constant cell and a transparent cell (the record cell). The overhead of the new representation can be tested as lists are still implemented as a constant and a transparent cell.

The efficient implementation -

```haskell
fun append [] y = y
  | append (a::b) y = a::(append b y);

fun reverse [] = []
  | reverse (a::b) = append (reverse b, a::nil);

repeat 10000 reverse [1,2,3,4];
```

The current implementation -

```haskell
datatype 'a MyList = Nil | Cons of 'a * 'a MyList;
infix Cons;

fun myAppend Nil y = y
  | myAppend (a Cons b) y = a Cons (myAppend b y);

fun myReverse Nil = Nil
  | myReverse (a Cons b) = myAppend (myReverse b, a Cons nil);

repeat 10000 myReverse (1 Cons (2 Cons (3 Cons (4 Cons Nil))));
```

The object code in the first case was 316 bytes for append, 253 bytes for reverse and took 152.03 seconds to execute (averaged over 25 runs). In the second case the object code was 354 bytes for append, 357 bytes for reverse and took 207.25 seconds to execute (again averaged over 25 runs). This represents an increase of 25% in object code size and 36% in execution time.
Appendix 7: Implementing Trace Commands

Appendix 7: Implementing Trace, Untrace and Importtrace.

Using function `makestring` and by substituting the text pointer, functions can be traced.

1. Implementing Trace.

An algorithm for tracing a function therefore has the following form.

To process the command `trace f`

Find f, check it can be traced

1. Locate phrase p that defines f in top-level environment
   * if no such p, f not a function, f defined in a module, or f already traced then error: f not traced
2. Existing_State is the environment and argument stack that existed when p was originally defined
3. Check `std_out`, `output`, `makestring` and `"` are not re-defined in Existing_Env, if re-defined, error: f not traced

Transform the definition of f

1. Extract abstract syntax tree for f from phrase p
2. Generate version of traced f, using trace schema
3. Implant traced f into abstract syntax tree of phrase p

Generate the text of traced f

1. Analyse phrase p in Existing_State
   * shouldn’t be in error, generates information for compiler
2. Compile FAM code for phrase p, again in Existing_State
3. Select the text of traced f from the FAM code generated
4. Assemble this text

Replace old text with traced text

1. Replace the text in f’s original closure with traced text

Do book-keeping

1. Add f to the traced identifiers of phrase p
2. Implementing Untrace.

An algorithm for untracing a function has the following form.

To process the command **untrace f** -

Find f, check it is traced

---

locate phrase p that defines f in top-level environment
if no such p, or f is not traced then error
Existing_State is the environment and argument stack that existed when p was originally defined

Transform the definition of f

---

implant saved version of original (untraced) abstract syntax tree for f into phrase p

Re-generate the text of original f

---

analyse phrase p in Existing_State
compile FAM code for phrase p, again in Existing_State
select the text of original f from the FAM code generated
assemble this text

Replace traced text with original text

---

replace the text in f’s closure with original text

Do book-keeping

---

remove f from the traced identifiers of phrase p

3. Implementing Importtrace.

Implementing **importtrace** is straightforward. After analysis, when the bindings that will be imported are known, each binding is examined to see if it is a function. If it is, the temporary definition described in Section 5.3.3.2 is built. A command **importtrace X**; is converted into a sequence of phrases of the form (**import X ; val f = .. ; val g = .. ;**); where f and g are the functions imported from module X. The sequence phrase is then compiled and executed normally.
References


