CONTENTS

SPECIAL ISSUE ON PROGRAMMING ENVIRONMENTS

46-54  Language-Specific Editors for Block-Structured Programming Languages  
       C.D. MARLIN

55-60  User Interface Issues in Programming Environments  
       D.A. CARRINGTON

61-66  An Overview of GED, A Language-Independent Syntax-Directed Editor  
       G.S. MORETTI and P.J. LYONS

67-74  An Adaptive Program Editor  
       J. WELSH, G.A. ROSE and M. LLOYD

75-80  Kay: A Beginner's Environment  
       B. DWYER

81-86  Recording the Computational History of a Research Project  
       B.G. COOK

87-89  A Prolog Environment System  
       J.D. NEWMARCH

90-95  Programming Environments – Deskilling or Job-Enriching?  
       C. SAUER

SPECIAL FEATURES

45  Guest Editorial

96  Letter to Editor

97-104  Book Reviews

Published for Australian Computer Society Incorporated  
Registered by Australia Post, Publication No. NBG 1124
TANDEM NONSTOP SYSTEMS ARE LEADING AUSTRALIANS AND NEW ZEALANDERS INTO ANOTHER WORLD.

TANDEM NONSTOP COMPUTER SYSTEMS PRIMARILY DELIVER ALL THE ELECTRONIC TRANSFER SERVICES IN AUSTRALIA AND NEW ZEALAND.

Tandem Computers control AUTOMATIC TELLER MACHINES, POINT OF SALE DEVICES and ELECTRONIC SWITCHING between our Financial Institutions.

TANDEM Computers have unique capabilities:

- **NonStop** - No single fault will cause the system to go down and the fault can be repaired while the computer system is still operational. Engineers can replace boards and modules plus the system can be upgraded whilst the system remains operational.
- **Growth** - A basic two processor Tandem System rated at 4.4 MIPS can be expanded in a selectable linear growth path to a 35.2 MIPS sixteen processor system without software changes or hardware replacements.
- **Networking** - Up to 255 systems can be inter connected to form a common, NonStop, data communications network, which is viewed as a single resource with the processing capacity of 4,080 central processors.
- **Relational Database** - A unique high performance relational database which permits views or relationships to be established at run time and which supports a truly distributed database across a network of Tandem systems.
- **Transaction Processing System (TPS)** - Which supports the efficient handling of a large volume of terminals in a multiple processor environment integrated with a unique dynamic load balancing to ensure maximum terminal response.

Companies base their business on computers today and cannot afford "not" to have the features available from "State of the Art" Technology.

![TANDEMCOMPUTERS](image)

MELBOURNE
(03) 267 1577
ADELAIDE
(06) 211 8655
SYDNEY
(02) 957 5566
PERTH
(09) 325 1588
BRISBANE
(07) 369 4511
WELLINGTON (NZ)
(04) 72 3286
Software development and adaptation is now a major industry and demands a high degree of professional knowledge, skill and experience. Software development spans specification, implementation, verification and commissioning and these phases should be respected even when existing software is modified or adapted. Every phase has been refined over many years and there is now an extensive technical literature. Accordingly, there is an obligation for professionals engaged in software development to be aware of developments and to incorporate systematically the evolving principles into their practice. Moreover it is essential to provide software tools for the various phases and to integrate the tools to form a development environment.

This issue of the Australian Computer Journal on Programming Environments focuses on the preparation and editing of computer programs and therefore is mainly concerned with the programming or implementation phase. Aids developed for program preparation frequently have wider application; for example, they might be adapted to accommodate languages or notations used for specification. Techniques for automatic generation of aids e.g. for exploiting modular structure, for suppressing detail, for version control, etc., also have widespread application. Now that software development is a significant activity world-wide, it is important to increase productivity through well-designed environments.

This issue includes eight papers which cover a broad range of issues from specific technical detail to thought-provoking social implications. Chris Marlin’s paper provides an excellent overview of editors for a specific class of programming languages, namely block-structured languages. The paper reveals the different points-of-view in the evolution of such editors, their advantages and some of their difficulties. The paper can be read with a minimum of background and affords an excellent introduction to the field of programming environments. David Carrington’s paper complements Marlin’s in that it concentrates on hardware aspects of modern user interfaces such as higher-resolution, bitmap-graphic displays, multiple windows and various input devices. Personal workstations vs. shared central facilities are also discussed.

Giovanni Moretti and Paul Lyons present techniques for the generation of a syntax-directed editor for any LL(1) language defined in extended BNF notation. Generators are particularly attractive because they accommodate a class of languages rather than a specific language. Jim Welsh, Melfyn Lloyd and myself describe an adaptive program editor which addresses the problems of accommodating extensive programs. Techniques for defining and manipulating contexts within a program and for suppressing detail are discussed. The paper also discusses structured vs. unstructured views of program text.

Barry Dwyer describes a programming environment, called Kay, of particular interest to educators responsible for teaching the principles of structured programming to raw beginners. Kay’s constructive responses constitute a pleasant interactive environment, aspects of which could be included with advantage in larger systems. Bruce Cook addresses the issue of retaining accurate records of the numerous files and versions which enter into the histories of a substantial computational project. A prototype project-control system is described which includes some automatic documentation facilities.

Jan Newmarch describes in a short communication an environment for Prolog, a language now of interest to a large and growing number of ACJ readers. The provision of a file management system assists the Prolog programmer in composing programs from existing code sections and to check various compatibilities, such as number of arguments, which are not part of Prolog per se. Chris Sauer asks whether programming environments are deskilling or job-enriching, and follows the evolution of ideas and consequences. Obligatory broad, social-impact studies might well join the phases of technical development.

As guest editor, I thank the authors of all submitted papers for their interest and effort. Unfortunately, not all of these could be included in the issue. I also thank the referees for their assessment and advice and the ACJ Editor, John Lions, for his guidance. I commend the papers in this issue to readers and encourage them to communicate with the authors: Programming Environments are of vital concern to a key professional activity.

Gordon A. Rose,
University of Queensland.
Language-Specific Editors for Block-Structured Programming Languages

C. D. Marlin†

The most common means of creating and manipulating a program is the text editor, which treats a program as a sequence of characters organized into lines. A recent development has been the emergence of language-specific editors, which enable the program to be created and manipulated using commands which take account of its structure. Knowledge about the structure of the program being edited can then be used to provide a more convenient program development tool. This paper examines a particular class of language-specific editors, those for block-structured programming languages. Examples of such editors are described and some issues in their design identified.

Keywords and Phrases: language-specific editors, syntax-directed editors, language-sensitive editors, language-oriented editors, structure editors, block-structured programming languages

CR Categories: D.2.3, D.2.6

1. Introduction

A language-specific editor is one which has been designed specifically for the creation and modification of programs written in a particular language; such editors are also known by a variety of other names, such as 'syntax-directed editors', 'language-sensitive editors', 'language-oriented editors', 'language-based editors' and 'structure editors'. Being special-purpose tools, such editors can provide commands which deal in the concepts of the language concerned and so can make the task of editing a program in the language more convenient to the user. Since they typically require fewer keystrokes from the user (a single, brief command may, for example, insert several keywords) and since they frequently involve extensive syntactic and semantic analysis of the program being edited (thus requiring fewer runs of a separate compiler, if one is used at all), such editors can result in more efficient utilization of a computer system when developing programs.

Language-specific editors manipulate some structured representation of the program. By far the most common representation used is a tree, which is typically known as an abstract syntax tree. In this case, editor commands are actually tree manipulation commands, although the extent to which the user is aware of this fact varies somewhat. A textual representation of a program, such as would be displayed on the screen of a terminal, can be obtained from the abstract syntax tree representation by a process called unparsing. There is considerable flexibility in the way that this unparsing is done, particularly with regard to formatting; because of its close relationship to the way some compilers produce formatted listings from their internal representations of programs, unparsing is sometimes also called prettyprinting.

Although the tree is the most common program representation, other representations have been used. For example, the Intelligent Program Editor described by Shapiro, McCune and Wilson (1982) employs a knowledge base which represents the structure of programs from a variety of views, such as a character-by-character representation and a 'semantic' description documenting the user's intent for various parts of the program. A number of attempts have been made to represent programs within the framework of the relational database; examples include the system described by Kanasaki, Yamaguchi and Kunii (1982) and the OMEGA programming environment (Powell and Linton, 1983; Linton, 1984). As admitted by Linton (1984), this form of program representation suffers from performance disadvantages at present; although this situation may change as database hardware and software evolves, the abstract syntax tree is pre-eminent at present and is the principal program representation in the systems surveyed in this paper.

In the next section, three language-specific editors are described; these editors represent three different...
and important approaches to language-specific editing of programs for block-structured programming languages. Each of the editors is described in sufficient detail to give the reader some impressions of what it would be like to use the editor. The interested reader is referred to Marlin (1985) for a more extended description of these editors. Each of the three approaches represented in Section 2 has its adherents and the three editors concerned have all influenced later designs. The subsequent section then discusses various issues in the design of a language-specific editor for a block-structured language and describes some emerging trends.

2. Approaches to Language-Specific Editing

2.1 The Cornell Program Synthesizer

Perhaps the most significant of the language-specific editors described in this paper is the Cornell Program Synthesizer (Teitelbaum and Reps, 1981; Teitelbaum, Reps and Horowitz, 1981). The Cornell Program Synthesizer is a programming environment specific to the programming language PL/CS (a subset of PL/I) and was developed at Cornell University to assist with the teaching of this language. It has influenced the design of most later language-specific editors.

The editor component of the Cornell Program Synthesizer is language-specific and so the entry and modification of program text is guided by a grammar for the language concerned. All but the simplest constructs must be entered using predefined templates. These templates contain all of the keywords and punctuation characters required by the construct, along with 'placeholders' for the other components (such as expressions, statements and so on). If a construct has an optional part, this is not normally included in the template; to have it displayed, the command '.o' must be employed (in which the ' preceding a command is a 'boldface dot', rather than a period, and signifies that this is a command).

The constructs of the language which do not have to be entered using templates are called 'phrases'. The only constructs of PL/CS regarded as phrases are assignment statements, expressions and lists of variables. Phrases are parsed immediately after they have been entered; if errors are detected, they are reported and can be corrected by text editing operations. A program is created by inserting templates and then replacing placeholders in the program by phrases or further templates.

To give some idea what it is like to create a program using the Cornell Program Synthesizer, part of an editing session will firstly be described. In this description, underlined commands (such as 'return') represent a single keystroke. Initially, the screen will show only the placeholder 'object'

with the cursor (represented by '□') indicating the '□'. If the user enters the command '.main', the effect is to replace the placeholder by the following template for a PL/CS main program block

```*/ comment */
name: PROCEDURE OPTIONS (MAIN);
{declaration}
{statement}
END name;
```in which there are now three further placeholders: 'comment', '{declaration}' and '{statement}'. A placeholder of the form '{...}' stands for zero or more occurrences of the item between the braces and so indicates that a list of such items is allowed here. The name of the program being developed was previously entered at the beginning of this session and is now inserted in the appropriate places (it is 'name' in this case). The cursor now indicates the 'c' of 'comment' and if the user now types 'a program', the effect is as follows

```/* a program */
name: PROCEDURE OPTIONS (MAIN);
{declaration}
{statement}
END name;
```When the user enters 'return' to signify that the comment is complete, the cursor moves down to the '{' of '{declaration}' and the screen would now show

```/* a program */
name: PROCEDURE OPTIONS (MAIN);
{declaration}
{statement}
END name;
```Entering the command '.fx' now inserts the template for a FIXED declaration at the cursor

```/* a program */
name: PROCEDURE OPTIONS (MAIN);
DECLARE (list-of-variables) FIXED;
{statement}
END name;
```and entering 'k, m' followed by 'return' replaces the current placeholder by this phrase, moves the cursor on to the second item in the list of declarations (for which a new placeholder appears) and the display now shows

```/* a program */
name: PROCEDURE OPTIONS (MAIN);
DECLARE (k, m) FIXED;
{declaration}
{statement}
END name;
```Entering 'return' again signifies the end of the declarations; the placeholder '{declaration}'
disappears and the cursor moves on to the placeholder '{statement}'. The statements for the block can now be inserted, and so on.

The process described above is typical of how programs are created using the Cornell Program Synthesizer. Creating a program is mostly a matter of replacing placeholders such as '{declaration}' by templates such as

DECLARE (list-of-variables) FIXED;

until a point is reached where a phrase (such as 'list-of-variables') is expected. At such a point, free text is entered as the replacement for the phrase. Because it involves a mixture between the top-down replacement of placeholders with templates (which may contain further placeholders) and the bottom-up construction of phrases from free text, the editor in the Cornell Program Synthesizer is sometimes referred to as a hybrid editor.

The nature of this hybridization (that is, which of the constructs may be entered as free text and which of them must be entered as templates) is fixed in the case of the Cornell Program Synthesizer. Later language-specific editors, such as Syned (Horgan and Moore, 1984) and the editors within the SUPPORT project (Zelkowitz, 1984) and ISDE (Chesi, Dameri, Franceschi, Gatti and Simonelli, 1984) programming environments, offer considerably more flexibility regarding the entry of free text. In addition to allowing the use of templates, they also permit free text to be entered at any time; this text is then parsed and, if it corresponds to the construct represented by the current placeholder, the resulting structure is added at the current point in the abstract syntax tree. These editors can thus be said to exhibit variable hybridization and this permits a simple solution to a difficulty that occurs with editors such as that in the Cornell Program Synthesizer.

This difficulty is concerned with the transformation of one structure into another (such as enclosing an existing statement in a loop). Editors with the variable hybridization property described above frequently allow structure transformations to be effected as follows:

- the structure to be transformed is unparsed to a textual form,
- this text is then transformed using traditional text-editing operations, and
- the resulting transformed text is then parsed and re-inserted into the program structure.

Another aspect of the Cornell Program Synthesizer is its provision of "comment templates". Comments are only allowed in three restricted contexts in PL/CS programs created with the Cornell Program Synthesizer: in the comment field of a procedure template, in the comment field of a comment template and at the end of a list of variables in a variable declaration or parameter specification. The comment template can be used to associate a comment with a single statement, a list of statements, a list of declarations or a list of parameter specifications. One use of this kind of template is to inhibit the display of parts of the program not currently of interest using the 'ellipsis' command, which displays only the comment and a line consisting of '...'. With the limited size of display devices, being able to leave out detail to reveal more of the surrounding context is clearly important. In general, this is called elision and the primitive form provided in the Cornell Program Synthesizer has been improved in later editors.

The work on the Cornell Program Synthesizer, a programming environment specific to the PL/CS language, has led to the development of a system for the generation of similar environments for other languages. This system is called the Synthesizer Generator (Reps, 1984; Reps and Teitelbaum, 1984) and produces a programming environment from a description of the syntax and semantics of the language, and a specification of some aspects of the user interface. Examples of programming environments generated by the Synthesizer Generator include a Pascal programming environment, a desk calculator and a proof checking system. More recent work by this group at Cornell University has been directed towards employing a relational database to describe some aspects of the program under development, while continuing to use an abstract syntax tree for other aspects (Horowitz and Teitelbaum, 1985).

2.2 Mentor

Mentor (Donzeau-Gouge, Huet, Kahn and Lang, 1980; Donzeau-Gouge, Kahn, Lang and Melese, 1984; Donzeau-Gouge, Kahn, Melese and Morcos, 1983) is described by its authors as a "structured document manipulation system" and is the result of a long-term project at Institut National de Recherche en Informaticque et en Automatique (I.N.R.I.A.) in France. The original goal of the project appears to have been the design of a Pascal programming environment, but later work has been directed at the manipulation various kinds of documents (including, but not restricted to, programs). The description here mainly focusses on the version known as MENTOR-PASCAL (Donzeau-Gouge et al., 1980), which is a language-specific editor for the Pascal language.

The Cornell Program Synthesizer, as described in the previous section, attempts to hide the underlying abstract syntax tree and present an entirely textual view to the user. For example, cursor movement commands in the Cornell Program Synthesizer are really commands which move the focus around the abstract syntax tree, but are presented to the user as commands which move the cursor through the program text in certain ways. Mentor, on the other hand, represents an approach to editing which presents the user with a tree manipulation language as the only means of editing the program.
Language-Specific Editors for Block-Structured Programming Languages

Figure 1. An abstract syntax tree for Mentor.

The internal nodes of a Mentor tree either have a fixed number of descendants (e.g. to represent the conditional statement "if ... then ... else ...") or a variable number of descendants (e.g. to represent a list, such as a list of statements). The leaf nodes are so-called 'atoms'; examples include constants, variables and procedure names. For example, the Pascal statement

```pascal
if X > 0 then P(X, A[Y, Z])
else begin
  Y := Y * 2;
  X := 0
end
```

yields the abstract syntax tree shown in Figure 1. Note that this kind of tree is different from a parse tree, where internal nodes always represent nonterminal categories in the grammar for the language. The internal nodes of the abstract syntax tree represent 'operators' in the language; examples of such operators for Pascal (the ones used in Figure 1), and their meanings, are shown in Table 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ass</td>
<td>assignment statement</td>
</tr>
<tr>
<td>call</td>
<td>procedure call</td>
</tr>
<tr>
<td>gtr</td>
<td>'greater than' comparison</td>
</tr>
<tr>
<td>if</td>
<td>selection statement</td>
</tr>
<tr>
<td>index</td>
<td>array element selection</td>
</tr>
<tr>
<td>lexp</td>
<td>list of expressions</td>
</tr>
<tr>
<td>lstat</td>
<td>list of statements</td>
</tr>
<tr>
<td>mult</td>
<td>multiplication</td>
</tr>
</tbody>
</table>

The operands in a Mentor abstract syntax tree are classified into 'sorts' (called 'phyla' in later papers); examples include 'exp' and 'stat', standing for 'expression' and 'statement', respectively. The notion of a sort allows the description of restrictions on what kind of language construct is permitted for a particular operand of a particular operator. In Figure 1, for example, the second and third operands of the 'if' operator must be of sort 'stat'.

Comments are regarded as attributes attached to nodes in the abstract syntax tree; these attributes are themselves trees in their own language. Two attributes are used in the case of Pascal: prefix and postfix comments. These attributes are not normally displayed with the program text, but are available for display at the user's request.

A user communicates with Mentor via an interpreter for a tree manipulation language called Mentol. This language includes variables called 'markers', which may be used to refer to 'locations' (nodes and the subtrees they define) within an abstract syntax tree. The names of markers are preceded by the character '3'; the marker '3K' is called the current marker. 'Location expressions' evaluate to locations; they consist of some 'base marker' (which defaults to the current marker) and some displacement relative to the location defined by the base marker. Displacements are expressed in terms of operations such as those shown in Table 2. These operations can be concatenated into arbitrary sequences. Thus, for example, if the node for the operator 'mult' in the abstract syntax tree of Figure 1 were denoted by the marker '3A', the location expression '3A U R S2' refers to the atom '0' within the tree. Mentol also has an assignment operator ('3:'), which allows a location to be assigned to a marker. For example, '3K:3A U R S2' causes the current marker to now refer to the atom '0'; in fact, the command '3:3A U R S2' would have the same effect, since an omitted marker defaults to the current marker.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Up: go to the parent of this node</td>
</tr>
<tr>
<td>L</td>
<td>Left: go to the node to the left of this node</td>
</tr>
<tr>
<td>R</td>
<td>Right: go to the node to the right of this node</td>
</tr>
<tr>
<td>Sn</td>
<td>Son: go to the nth descendant of this node</td>
</tr>
</tbody>
</table>

A number of control structures are provided for combining Mentol commands and for controlling their execution. For example, they can be executed in sequence (with ':'), iterated (either for a fixed number of times or until failure) and selected conditionally. It is also possible to write procedures in Mentol and these procedures may have parameters.

Mentol provides commands for the manipulation of subtrees within the abstract syntax tree: the change command

```markdown
< location > C < tree >
```

replaces the subtree indicated by the location expression with the abstract syntax tree described by the second argument, and the exchange command

```markdown
< location > X < location >
```

exchanges the two subtrees whose roots are identified
by the location expressions (provided that these sub­
trees are disjoint). Both of these commands check
that the root of a subtree to be inserted at a particular
location in the abstract syntax tree is of the correct
sort. For example, assuming that the node represent­
ing the ‘if’ operator in Figure 1 were denoted by the
marker ‘STOP’, the command ‘STOP S2 X S3’ would exchange the call statement and the compound
statement in the program, whereas the command
‘STOP S1 X S2’ would result in an error.

Mentor also contains facilities for tree pattern
matching. A tree pattern (called a ‘schema’) is an
abstract syntax tree containing at least one leaf node
which is a so-called ‘metavariable’. These metavari­
ables become defined when the schema matches some
tree structure; a successful match occurs when the
schema is the same as some subtree, with the meta­
variables taking on appropriate values. Mentor pro­
vides predefined schemas for each of the operators of
Pascal and these can be used to perform transforma­
tions on a Pascal program.

The remaining facilities of Mentor include explicit
unparser and parser commands. The unparser can be
used to display tree structures as text and embodies an
elosion facility. The parser permits the creation of
tree structures from text; if the entered text contains
errors, the parser attempts to make as much sense of
it as possible, deducing the likely intended structure of
the program, a task which is clearly not trivial.

Another difficult aspect of writing a parser of this
kind is the question of deciding to which node a com­
ment should be attached when the comment is
encountered in reading the program text.

From a single language-specific editor for Pascal,
the Mentor project has evolved into a family of edit­
ors, all sharing the Mentol tree manipulation
language. Two themes have emerged in the later' work.
Firstly, efforts have been directed at accom­
modating different coexisting views of a document;
this amounts to permitting the simultaneous editing of
abstract syntax trees belonging to different languages
and can be used, for example, to allow comments to
be edited in a manner quite different from the way in
which the remainder of the program is edited. The
second theme concerns the ability to generate a Men­
tor editor from a description of the language to be
manipulated; the language Metal (Donzeau-Gouge et
al., 1983; Kahn, Lang, Melese and Morcos, 1983) is
used to describe the syntactic structure of the
language to be manipulated by the generated editor.

The Mentor system has had considerable influence
on CEYX (Hullot, 1983), a system for VLSI design,
also produced at I.N.R.I.A. Like Mentor, CEYX
employs an abstract syntax tree based on the
operator-operand model, the notion of phyla and the
ability to simultaneously edit abstract syntax trees
belonging to different languages. Very few other edit­
tors have adopted the explicit tree editing approach of
Mentor; exceptions include EDT (Finlayson, 1983).

2.3 COPE
Like the Cornell Program Synthesizer, COPE (Archer
and Conway, 1981; Conway, DeJohn and Worona,
1984) is a programming environment for the PL/CS
language. Despite this, and the fact that they were
developed contemporaneously at Cornell University,
the two environments adopt markedly different
approaches to language-specific editing. Whereas the
editor in the Cornell Program Synthesizer (which was
described in Section 2.1) is essentially a top-down
template-based editor, the COPE editor is bottom-up:
editing is performed via a text editor tied to an error-
correcting parser. Input to the system is first parsed;
if it is correct, it is executed or inserted into the pro­
gram, as appropriate. If the input contains errors,
it is corrected before it is used; these corrections may
involve the creation of template-like structures.

All commands to the COPE system are followed by
either ‘FILE’ or ‘EXECUTE’; like the underlined
command names in the description of the Cornell
Program Synthesizer, these represent single keystrokes
on a special keyboard. The former suffix means that
this command is to be saved (i.e. it is a call on the
editor), whereas the latter means that the command is
to be executed immediately (i.e. it is a call on the
compiler and execution system). The text before the
‘EXECUTE’ keyword is interpreted according to the
rules of PL/CS and executed as if it were inserted at
the current point in whatever program is being edited
or executed, unless it is preceded by a period (.), in
which case it is regarded as a file name.

The COPE editor operates in several different ways
depending on how much information is supplied by
the user. If the command entered by the user consists
of a single keyword, the editor inserts the template
denoted by that keyword. However, if a complete
construct is entered, the text entered by the user is
checked, perhaps corrected and then inserted into the
program. In fact, automatic correction of errors is
an important part of the design of COPE; for exam­
ple, it automatically inserts declarations that may be
needed to ensure the correctness of a piece of text
entered by the user.

To give the flavour of the COPE philosophy, con­
der the following example of the creation of a new
program. The first command might be ‘.SAMPLE
FILE’, specifying that a new procedure ‘sample’ is
to be created and stored on the file ‘.sample’. After
this command, the following text is displayed to the
user

```c
=> SAMPLE: PROC;
  END SAMPLE;
```

in which ‘=>’ signifies the current position of the
‘edit-pointer’. Next, the user enters the command
‘WHILE FILE’, which does not mean the insertion
of a ‘WHILE’ keyword, but rather indicates that a
complete template for a while loop is required:

```plaintext
SAMPLE: PROC;
=> W1: DO WHILE cond;
END W1;
END SAMPLE;
```

where 'cond' is a placeholder for a condition in the created template. Note that because PL/CS requires all while loops to be labelled, the editor has inserted a label ('W1'). If the user now enters the command

```plaintext
get x sum = sum + x FILE
```

display would become

```plaintext
SAMPLE: PROC;
DCL (X) FLOAT;
DCL (SUM) FLOAT;
W1: DO WHILE cond;
GET LIST(X);
=> SUM = SUM + X;
END W1;
END SAMPLE;
```

If the entry had been a legal condition, it would have replaced 'cond', but since it was not, it was interpreted as statements to be inserted into the loop body. The first two components of the entry ('get x') are corrected to the statement 'GET LIST(X)'; and also cause the introduction of a declaration for the variable 'X'. The remainder of the entry is regarded as the assignment statement 'SUM = SUM + X'; and this is also inserted into the loop; in doing so, a declaration for 'SUM' is inserted at the head of this procedure.

This short example illustrates how much deduction and error correction is carried out implicitly within this kind of editor. The difficulties of accurately guessing what a user meant by an incorrect entry are well-known (for example, see Section 8.4.1 of Pyster, 1980) and clearly apply in this case. One factor which reduces the severity of this problem in the case of a system such as COPE is that, being interactive, the user can review the guesses made by the system and make corrections if necessary.

The broad differences between COPE and the Cornell Program Synthesizer are illustrated by the fact that Teitelbaum, one of the designers of the Cornell Program Synthesizer, has characterized COPE as 'anarchistic', whereas the designers of COPE describe the Cornell Program Synthesizer as 'totalitarian' (Archer and Conway, 1981). Like the Cornell Program Synthesizer, COPE has inspired the design of a number of other language-specific editors using a similar bottom-up error-correcting approach, including the Pascal-oriented editor Poe (Fischer, Johnson, Mauney, Pal and Stock, 1984).

3. Design Issues

3.1 Introduction

The three editors described in the previous section illustrate three quite different approaches to language-specific editing. The Cornell Program Synthesizer adopts a highly structured approach, in which programs are developed in a top-down fashion using templates; the only exception to this structured approach is the ability to enter phrases as free text. Mentor is also highly structured, but differs from the Cornell Program Synthesizer in that its editing operations are manifestly tree manipulations, as opposed to the attempt to hide similar tree manipulations behind a textual facade in the Cornell Program Synthesizer. COPE represents a bottom-up approach to the construction of programs, where all input is free text and is parsed by an error-correcting parser; depending on the results of the parser's analysis (and possible corrections), the system then makes some change to the program under development or directly executes some piece of program.

In this section, various issues involved in the design of a language-specific editor for a block-structured language are discussed briefly. Most of these issues have already been illustrated at least partially in the previous section.

3.2 Top-down versus bottom-up

It is clear from the previous section that a major issue in the design of language-specific editors is the matter of top-down development of programs using templates versus the bottom-up construction of programs or program fragments from free text.

Language-specific editors which are purely template-based, only permitting free text for tokens such as identifiers and constants, are rare. One example is Emily (Hansen, 1971), which is perhaps the earliest language-specific editor for a block-structured language; this editor is table-driven (with tables being written for PL/I, among other languages) and operates in top-down fashion with alternatives for nonterminal symbols being selected from a menu with a light-pen. Other purely top-down language-specific editors are the editor described by MacLennan (1981) and ALOE editors generated by ALOEGEN (Medina-Mora, Ellison, Garlan, Kaiser and Notkin, 1983), such as GNOME (Garlan and Miller, 1984).

Purely bottom-up editors include COPE (Archer and Conway, 1981), Poe (Fischer et al., 1984) and Mentor (Donzeau-Gouge et al., 1980), all mentioned previously; in each case, the parser which analyses the input performs error correction. In the case of COPE and Poe, this error correction may involve creating template-like sections of program text. In other language-specific editors adopting the bottom-up approach, such as the editors in the SAGA (Campbell and Kirslis, 1984) and Magpie (Delisle, Meniconosy and Schwartz, 1984) programming environments, there is no attempt at error correction; the erroneous sections are simply highlighted in some way. Morris
and Schwartz (1981) describe a Pascal-specific editor which is bottom-up down to the level of single characters. Since this approach has the potential to introduce a large number of intermediate syntax errors when editing a program (consider, for example, editing keywords), the restriction is imposed that the text up to the token before the cursor must be a legal beginning for a Pascal program; an error need not be corrected immediately, but the cursor is confined to that part of the program at or before the error.

Apart from the above extremes, two kinds of hybrid schemes are possible. Firstly, as has already been discussed in Section 2.1, the Cornell Program Synthesizer is what can be termed a fixed hybrid: the language constructs have been partitioned into those which are entered via templates and those which are entered as free text. A similar approach is used by the editor for Ada† within the Arcturus programming environment (Standish and Taylor, 1984). Other hybrid editors permit the use of templates or free text entry at virtually any point during program editing; these may be termed variable hybrid editors. Examples of the latter include:

- Syned (Horgan and Moore, 1984),
- the EDT editor (Finlayson, 1983),
- the editor in the SUPPORT environment (Zelkowitz, 1984),
- the editor in the PASES environment (Shapiro, Collins, Johnson and Ruttenburg, 1981),
- the editor in the ISDE environment (Chesi et al., 1984),
- editors generated by the DICE system (Fritzson, 1984),
- editors generated by the HEG system (Bottos and Kintala, 1983), and
- editors in environments generated by the PECAN system (Reiss, 1984).

### 3.3 Language independence

A clear trend in the work on language-specific editors has been the increasing emphasis on the development of tools which allow language-specific editors (and complete programming environments) to be generated from specifications of the language to be handled. This trend is illustrated in the development of the Cornell Program Synthesizer and Mentor projects, described in earlier sections: both began as attempts to produce a programming environment for a particular language (PL/CS in the former case and Pascal in the latter case), but have led to tools for the generation of systems.

The Gandalf project (Habermann, 1979) at Carnegie Mellon University is another project which appears to have evolved from the production of a single programming environment for a particular language to the generation of environments from descriptions. The initial goal of the project, as described by Habermann (1979), was an Ada programming environment called IPC; later, this goal had become the production of an environment (called IPE and, later, LOIPE) for a language called GC, a variant of the language C, and an associated editor generator (Medina-Mora and Feiler, 1981). The final product, at this stage, is ALOEGEN (Medina-Mora, 1982; Medina-Mora et al., 1983), a syntax-directed editor generator; editors generated by it, such as GNOME (Garlan and Miller, 1984), are regarded as instances of an editor called ALOE. Notkin (1985) gives a description of the history of the Gandalf Project and the nature of its components.

Other editor/environment generators include HEG (Bottos and Kintala, 1983), Poegen (Fischer et al., 1984), CEYX (Hullot, 1983), PECAN (Reiss, 1984) and the systems which generate SAGA (Campbell and Kirsils, 1984) and ISDE (Chesi et al., 1984) environments. An interesting comparison of such generators is provided by Klint (1983), who presents the complete specifications required to generate environments for a language PICO using Mentor, the Synthesizer Generator and CEYX.

### 3.4 Comments

Comments present a number of difficulties for language-specific editors. As pointed out by Fritzson (1984), there are essentially two possible approaches to handling comments in language-specific editors:

- the textual approach: allow comments anywhere and reconstruct the text exactly as the user created it, and
- the structured approach: impose restrictions on where the comments may appear.

The first approach is fraught with implementation difficulties, including the difficulty, mentioned earlier in the context of Mentor, of attaching comments to appropriate abstract syntax tree nodes when parsing program text. Fritzson’s own DICE system follows the latter approach and allows only ‘left’ comments (before a node) and ‘right’ comments (after a node).

Another difficulty with comments is that, being text, they probably have no meaningful internal tree structure and hence editing operations being used to manipulate the program are not suited to them. This has led, for example, to the use of separate structures for a program and its comments in the Mentor system.

### 3.5 Unparsing

There are a number of issues concerned with the display, or unparsing, of the abstract syntax tree for the program being edited. One which has received a good deal of attention is the problem of displaying as much meaningful context as possible around the current focus, given the size limitations of the display. The process of building a textual display of

---

The Gandalf project (Habermann, 1979) at Carnegie Mellon University is another project which appears to have evolved from the production of a single programming environment for a particular language to the generation of environments from descriptions. The initial goal of the project, as described by Habermann (1979), was an Ada programming environment called IPC; later, this goal had become the production of an environment (called IPE and, later, LOIPE) for a language called GC, a variant of the language C, and an associated editor generator (Medina-Mora and Feiler, 1981). The final product, at this stage, is ALOEGEN (Medina-Mora, 1982; Medina-Mora et al., 1983), a syntax-directed editor generator; editors generated by it, such as GNOME (Garlan and Miller, 1984), are regarded as instances of an editor called ALOE. Notkin (1985) gives a description of the history of the Gandalf Project and the nature of its components.

Other editor/environment generators include HEG (Bottos and Kintala, 1983), Poegen (Fischer et al., 1984), CEYX (Hullot, 1983), PECAN (Reiss, 1984) and the systems which generate SAGA (Campbell and Kirsils, 1984) and ISDE (Chesi et al., 1984) environments. An interesting comparison of such generators is provided by Klint (1983), who presents the complete specifications required to generate environments for a language PICO using Mentor, the Synthesizer Generator and CEYX.

### 3.4 Comments

Comments present a number of difficulties for language-specific editors. As pointed out by Fritzson (1984), there are essentially two possible approaches to handling comments in language-specific editors:

- the textual approach: allow comments anywhere and reconstruct the text exactly as the user created it, and
- the structured approach: impose restrictions on where the comments may appear.

The first approach is fraught with implementation difficulties, including the difficulty, mentioned earlier in the context of Mentor, of attaching comments to appropriate abstract syntax tree nodes when parsing program text. Fritzson’s own DICE system follows the latter approach and allows only ‘left’ comments (before a node) and ‘right’ comments (after a node).

Another difficulty with comments is that, being text, they probably have no meaningful internal tree structure and hence editing operations being used to manipulate the program are not suited to them. This has led, for example, to the use of separate structures for a program and its comments in the Mentor system.

### 3.5 Unparsing

There are a number of issues concerned with the display, or unparsing, of the abstract syntax tree for the program being edited. One which has received a good deal of attention is the problem of displaying as much meaningful context as possible around the current focus, given the size limitations of the display. The process of building a textual display of
part of a program, leaving out detail to reveal more context, is known as elision (or holophrasting). This process of elision is usually under the control of the editor; sometimes, as in the case of the editors described by Nienhuis (1983) and Broom and Welsh (1986), the user is given the ability to widen and narrow the 'focus', controlling the amount of context displayed. Mikelsons (1981) describes a sophisticated approach to elision which examines the user's recent behaviour and deduces whether the user is currently working in 'edit', 'reading' or 'multiple focus' mode; based on this analysis, an appropriate form of elision is used.

The fact that elision is feasible with language-specific editors, whereas it is not with text editors, does not in itself justify abandoning windowing (displaying as many contiguous lines as will fit on the display) as a means of displaying program text. As pointed out by Waters (1982), good elision is difficult to do and the large number of approaches in current use suggests that no method is clearly superior.

3.6 Semantics
Most language-specific editors perform some static semantic checking, but this is an area where further developments can be expected. In particular, few editors have commands which make use of the static semantics of the program in the same way that the syntactic structure is used. One way that this might be done is in a collection of commands which make use of the meanings of names, such as searching for the declaration corresponding to a particular use of a name or iterating over all uses of a name declared in a particular declaration. Another way that this knowledge can be used is to highlight references to a variable whose declaration has been removed or was never inserted.

Unfortunately, the descriptions of existing language-specific editors contain very few details on their semantic aspects. In fact, there appear to be no editors with the range of semantics-based commands described above.

4. Summary and Conclusions
This paper has described some approaches language-specific editing of block-structured programming languages. The three editors covered in Section 2 are representative of three such approaches and have also turned out to be significant in terms of influence on the design of later editors. None of the three approaches represented by the editors discussed in Section 2 has so far emerged as the pre-eminent one.

The way in which the predominantly top-down approach of the Cornell Program Synthesizer can be blended with the bottom-up approach of the COPE system has already been discussed in Section 3.2. The increasing availability of workstations with high-resolution displays is leading to the development of editing environments which offer the user a variety of language-oriented editors. This may, for example, allow the user to switch between text-based editing (corresponding to systems like the Cornell Program Synthesizer and COPE) and graphical editing of the abstract syntax tree (amounting, perhaps, to a graphical version of the Mentor editor). Indicative of the kind of programming environment which is possible on high-resolution displays are environments generated by the Pecan system (Reiss, 1984) and the MultiView programming environment (Marlin, 1986).

5. Acknowledgements
The author would like to thank Michael McCarthy, Michael Oudshoorn, Mary Pfreundschuh and Mark Prior for their comments on earlier versions of this paper.

References


Biographical Note

Chris Marlin has been a Lecturer in the Department of Computer Science at the University of Adelaide since 1984. He completed his Honours degree in Computing Science in 1973 and his Ph.D. in Computing Science in 1979, both at the University of Adelaide.

From January 1980 to December 1983, he was an Assistant Professor of Computer Science at the University of Iowa, Iowa City, Iowa (U.S.A.). His research has primarily been concerned with programming language design and implementation, especially in relation to coroutines and parallel processes, and various aspects of integrated incremental programming environments.
User Interface Issues in Programming Environments

D. A. Carrington†

A programming environment is a context in which software is developed. The user interface refers to the interaction between the software developer and the computer and is an important aspect of the programming environment. Technological advances, especially in hardware, are changing the nature of this interface; the current trend is personal workstations. This article reviews some features of modern user interfaces such as bitmap graphics and multiple windows and discusses their consequences.

Keywords and Phrases: user interface, programming environment, personal workstation

CR Categories: D. 2. 2, D. 2. 6, I. 7

1. Programming Environments

What is a Programming Environment?

A programming environment is a context in which software is specified, designed, implemented and tested. The broad view is that the context includes not only the technical tools and techniques used, but also the organisational and managerial structures that influence the development of software. Such a definition makes the study of programming environments synonymous with (at least) some definitions of software engineering. In this paper the discussion is restricted to the purely technical aspects of programming environments, in particular the computer-based tools.

A major concern that has led to interest in programming environments is a widespread desire to improve both the quality and the productivity of software development. Because human productivity has become increasingly important, the interface between the computer system and the human software developers is crucial to efforts to improve the programming environment.

Changes in System Architecture

Changes in the architecture of interactive systems are influencing the development of programming environments. There is a continuing trend away from centralised timesharing systems toward personal computers or workstations. The term, workstation, is commonly used to describe higher performance personal computer systems, generally with high resolution graphic displays. Comparisons between personal computing and timesharing systems (Gutz, Wasserman and Spier, 1981; Yalamanchili, Malek and Aggarwal, 1984) generally emphasise the improved response that results from localised processing power. Other positive aspects are the improved availability and potential for incremental growth in a computing facility.

The major disadvantage of personal computers is the potential loss of communication and sharing among users. The importance of communication has forced the development of networks to connect personal machines. Another disadvantage is the increased complexity associated with system administration including file backup and recovery, hardware and software maintenance, and security.

Most personal computer networks are composed of independent computer systems that normally operate in isolation. The conventional network is intended to overcome this isolation, but tends to cause considerable duplication of resources (both hardware and software, e.g. copies of programs) because resources are not shared conveniently. For example, idle processors are typically unavailable for other users of the network. The advent of diskless workstations that use disk storage elsewhere in the network is an attempt to address one aspect of this problem.

The trend toward decentralisation can be extrapolated to envisage each user with only a display server that uses a network to access other resources. A network of this form provides servers to control access to shared resources. An important difference is that the overall system is partitioned into components that are unlikely to be identical or independently useful. The primary function of a display server is to provide sufficiently responsive interaction. Based on experience, most of the display processing will need to be performed locally.

†Department of Computer Science, University of New South Wales, Kensington 2033, Australia. Manuscript received January, 1986; revised April, 1986.
Examples

To illustrate various aspects of this topic, I shall use two examples: the Smalltalk-80 system (Goldberg and Robson, 1983; Goldberg, 1984) and the AT&T 5620 terminal (Thompson and Kelly, 1983). Both are intended as a programming environment and offer a graphical interface via a bitmap display. The Smalltalk-80 system is the outcome of many years of research at Xerox PARC; it includes a programming language and a set of software tools. Smalltalk-80 is an example of an environment developed to operate on a variety of personal computers. By comparison the AT&T 5620 DMD (Dot Mapped Display) terminal is a programmable, bitmap terminal that, together with some software, enhances the well-known UNIX† programming environment (Ritchie and Thompson, 1974; Kernighan and Mashey, 1981).

The AT&T 5620 terminal has its origins in the Blit terminal developed at Bell Laboratories by Pike, Locanthi and Reiser (Pike, 1984a; Pike et al., 1985). As well as a device for improving the user interface of existing timesharing systems, the 5620 can be viewed as a prototype display server for a functional network. The Rainbow workstation (Wilkes et al., 1984) is another example of a display server. It is an advanced terminal designed as a front-end to remote computing power accessed via a local area network.

A comparison between Smalltalk-80 and the 5620 is valuable since the major components of Smalltalk-80 were designed as a single entity, while the 5620 terminal is an addition that occurred long after the original design of UNIX. The term integrated expresses a requirement that tools interact with one another in a consistent way and that they are used according to a set of well-understood conventions (Gutz et al., 1981). Smalltalk-80 and UNIX are examples of integrated programming environments.

The Programming Process

Producing software is a complex task involving many distinct activities. A classification by Schneideman (1980) listed six activities: specification and design, composition of new programs, comprehension of existing programs, testing to verify that programs satisfy their design requirements, program debugging and modification, and documentation of all the other activities. A programming environment should make these activities easier to perform and reduce the opportunity for errors or omissions in the development process. To simplify the user interface, Delisle, Menicosy and Schwartz (1984) advocate that the user should see a programming environment not as a collection of tools, but as

"a single tool that encapsulates the mechanisms used to implement the environment's functionality."

(p. 49)

This approach encourages uniformity and consistency in the user interface making it easier to learn and easier to use.

2. The User Interface

The term user interface describes the interaction between the user and the computer. A definition of user interface by Moran (1981) presents two viewpoints. For the designer, the user interface is

"the terminal hardware plus the software that receives, interprets and sends messages (and displays) to the user." (p. 5)

while for the user,

"the user interface is any part of the computer system that the user comes in contact with—either physically, perceptually, or conceptually." (p. 5)

Using an interactive system requires a sequence of requests from the human user interspersed with the presentation of computer generated responses. The user interface dictates the form of the input requests and the output responses. While the user interface can be divided into the components concerned with input and those concerned with presentation of output, there is normally considerable interaction between these two classes of components.

Conventional Displays

For many years, the user interface of interactive computing systems has been dominated by the cathode ray tube (CRT) display terminal. This device has traditionally been used as an output device capable of displaying 24 lines each of 80 characters. It is normally complemented by a keyboard as an input device. The conventional user interface for this type of display device is based on line-oriented commands although screen-based editors typically use character-level commands.

Bitmap Displays

Alternative user interfaces have generally required more graphics capability than is possible with the standard video terminal. With the falling cost of hardware, bitmap displays have become popular. Bitmap graphics are based on the discrete nature of the CRT tube with one bit of memory determining the state of one pixel of the display. More than one bit is required to generate displays with grey scales or colour. The graphics model for bitmap displays uses the bitblt (bit-boundary block transfer) operator as the building block for all graphics operations. It operates on rectangular bitmaps and is used for character drawing, highlighting regions of the screen, scrolling text and many other useful bitmap operations. Bitmap displays use large quantities of memory and processing power. For example, scrolling the screen requires moving every bit representing a pixel. For a 1024×1024 display, this is more than one million bits. The efficiency of the bitblt operator, in hardware or software, has considerable impact on the overall performance of the display system. A detailed discussion

†UNIX is a trademark of AT&T Bell Laboratories.
of bitmap graphics, including a specification of bitblt, possible implementations and performance issues, can be found in Pike, Guibas and Ingalls (1984).

Input Devices
The introduction of graphics into the user interface exposes a weakness in the keyboard as the only input device. The most common need is the ability to point at some object or position on the display. Numerous devices have been suggested for this task. The light pen and the touch sensitive screen use direct pointing at the display, but with a vertical screen, it can be tiring to use these devices and the user's hand tends to obscure the screen. Devices such as a mouse, joystick, tracker ball or tablet all use indirect pointing. Moving one of these devices causes a tracking symbol (or cursor) to move on the screen.

Many pointing devices also have buttons to acknowledge the completion of a positioning operation. The mouse, for example, typically has from one to three buttons. The 5620 and the Xerox implementation of Smalltalk both use a mouse with three buttons, although Smalltalk has been implemented on the Apple Macintosh, which has a one button mouse. The buttons on a mouse differ from the keys of a keyboard because pressing and releasing a mouse button are treated as independent operations. It is common to press a mouse button causing an action associated with the current position, move the mouse, and then release the button causing a second action related to the new position. An arbitrary sequence of text on the display can be selected by indicating the end-points in this way.

One concern in designing a user interface is to minimise the need for the user to switch between input devices. The most successful interfaces generally use the keyboard for entering only text.

Menus and Icons
Instead of line-oriented commands, graphic interfaces generally use menus or icons. A menu is a list of items representing the set of possible choices. Menu systems come in several different styles. On conventional terminals, each menu typically replaces the previous contents of the screen to avoid confusion. On bitmap terminals, more sophisticated techniques can be used in conjunction with a pointing device. A pop-up menu appears in response to a user action temporarily obscuring whatever is behind the menu. Once a selection has been made the menu disappears, restoring the obscured region. A pull-down menu is normally represented by a title. Selecting the title causes the full menu to appear. The advantage of these methods is that the screen area is used temporarily and is restored after the menu selection is complete. Menu systems work best when the set of possible choices is small.

An icon is a graphical image representing an object or an action. Icons can be classified (Lodding, 1983) as

- representational where the icon is a simplified picture of the object,
- abstract where the icon attempts to convey some abstract concept, or
- arbitrary where the meaning of the icon must be learnt.

Icons can provide status information or they can be manipulated to control the user interface. Scroll bars are icons that perform both roles. They indicate which section of a document is visible in a window. By selecting different parts of the scroll bar, it is normally possible to reposition the window relative to the document.

The cursor is a visual cue whose position governs the context of input. There may be multiple cursors—one for keyboard input and another identifying the current pointing position. In a graphical system, icons can be used for the cursor. Changing the shape of a cursor can provide visual feedback about where the user is pointing and the current status of the system. Icons can also be used as items in a menu.

Examples of cursor icons in the 5620 are

- a gun target to request selection of an object,
- a skull & cross bones to confirm a potentially dangerous operation and
- a coffee cup when a time-consuming operation is active.

Windows
Many interactive systems constrain the user so that the response from each request must complete before the next request is processed. This approach restricts the user to a single active task at any one time. The UNIX shell (command interpreter) introduced a convenient notation for creating asynchronous processes so that each user could have multiple independent activities. There are some limitations. Once a process has been created to run in the background, it can not interact with the user. Also the output from background processes is (non-deterministically) interleaved with output from other processes unless it is redirected.

Some of these limitations are overcome by the job control facilities of the C-shell from the Berkeley version of UNIX. These facilities allow processes to be switched from the background to become interactive and vice versa. Some control of output is also available but it is not complete. What is not provided is the visual context associated with each process. The screen displays only the most recent activities and does not restore the displayed information after a context switch between activities.
A better model for handling multiple activities with a single display is the window concept where each activity has a virtual terminal or window. The input and output for each independent activity is differentiated by being displayed in a separate window. A window manager is the software that maps the set of windows to a physical display and handles the dynamic creation, manipulation and deletion of windows. Almost all windowing systems require that windows be rectangular. There are several possible relationships between multiple windows. The simplest is to divide the physical display into non-overlapping regions. A more general alternative permits one window to obscure part or all of other windows but requires more effort to implement.

Overlapping windows have the advantage that the size of windows is not constrained by the number of windows and more useful information can be displayed on the screen. With overlapping windows, there is a total ordering imposed on the collection of windows that determines the composite image. The top window is not covered by any other and the visibility of each window is determined by the size and position of all the windows above it. The user requires more functionality to manipulate overlapping windows. Smalltalk-80 and the 5620 both use the overlapping window paradigm.

Windowing systems are not restricted to bitmap displays. There are several implementations for conventional character terminals (Weiser, 1985; Bresnahan, Barnard and MacLeod, 1984), but they are a poor substitute for a bitmap windowing system. The primary problems with windows on conventional terminals are the limited screen size and the lack of support in the terminal for manipulation of sub-regions of the screen. For example, scrolling one window without affecting the rest of the screen is normally impossible. The result is increased load on the host system which must model the screen contents to support windowing. Other limitations compared to a bitmap display are the lack of multiple character fonts and the inability to easily mix text and graphics.

The basic operations for manipulating windows are:
- creating a new window.
- moving a window around the screen.
- changing the size of a window.
- deleting a window.
- altering the top to bottom ordering of overlapping windows. Operations are normally provided to make a selected window either the top or the bottom.

One design issue is determining which window receives input from the keyboard and input graphics device. In Smalltalk-80, the current or active window for input must be on top but this is not true for the 5620. A difficult aspect to implement is output to a partially covered window. Some window systems preclude this by only allowing output to the current window that must be the top one. This restriction severely limits the power of the windowing system since activities in windows other than the current one can not proceed in parallel.

For the Blit, Pike (1983) extended the concept of a bitmap to include rectangular regions that are subdivided into visible and invisible components. These are known as layers. The traditional bitmap operations were extended to work with layers. The management of layers when windows are created, moved or destroyed is done by support software in the terminal and it is not a concern of programs using the windows. This design allows existing programs to be used with the 5620 without modification.

A partial set of criteria for judging the quality of a user interface is suggested by Furuta, Scofield and Shaw (1982):
- the amount of detail that the user must memorise to use the system;
- the amount of mental and physical effort that is required to perform common functions;
- the average number of errors made by the user, especially including errors from which recovery is difficult;
- the amount of time that the user is required to wait for the system to perform its function.

A significant advantage of a graphical interface is that it exploits human visual processing capabilities. The task of remembering and entering commands is replaced by the simpler task of recognising and selecting menu items or icons. This style of user interface is particularly suitable for novice or casual users but its acceptance is not restricted to that class of user. Both the Smalltalk environment and the AT&T 5620 are designed for experienced programmers.

Much of the pioneering work with graphical interfaces for computing systems was done at Xerox PARC. Englebart is credited (Warfield, 1983) as being one of the earliest to choose the mouse as a graphical input device in preference to alternatives such tracker balls, lightpens and touch-sensitive screens. The Xerox Star (Smith et al., 1982) and the Smalltalk environments are two better known examples of this work. The desktop metaphor introduced by the Star is an example of a conceptual model for users. The intention was for the screen to act as a model of a real desktop. The actions of manipulating objects on the screen was explained by analogy with the familiar physical desktop. This metaphor has subsequently been widely adopted and popularized by the Apple Lisa and Macintosh.

3. Requirements
The hardware requirements to support a graphical user interface for software development are:

1. A high resolution display: It should be capable of displaying a complete page of text. The requirements for text and graphics typically imply a bitmap display. Smalltalk-80 has been implemented on a variety of hardware configurations including workstations manufactured by Xerox, Tektronix, Hewlett-Packard, Digital Equipment Corporation and Apple. The 5620 has a monochrome, 800* 1024 pixel display. The display is interlaced which reduces the memory bandwidth required to refresh the display, but most workstations use non-interlaced displays to improve the quality of dynamic graphics.

2. Local processor and memory: These must be sufficient to support dynamic graphics and an acceptable level of response. Feedback in an interactive system is an important feature of the user interface and reliable response is essential. Current requirements would be for a 32 bit processor with at least one megabyte of memory. The 5620 has a 32 bit proprietary processor and 256K bytes of memory while an option supplies one megabyte. The display itself uses 100K bytes of this memory. There is also 64K bytes of EPROM which contains the basic software for controlling the terminal. Bitmap operations are often implemented using special purpose hardware but the 5620 implementation of the \textit{bitblt} operator demonstrates that this is not necessary (Pike et al., 1985).

3. Input devices: A keyboard and a graphics input device such as a mouse are required to allow the user to identify locations on the screen by pointing at them. The conventions for the mouse buttons with the 5620 are that the left button selects objects, the middle button produces a pop-up menu of actions relevant to the currently selected object, while the third button produces a pop-up menu of global actions.

Other hardware components are found in many user interface devices although items such as disks are generally not required to support the user interface. Colour displays and audio input and output are additional possibilities to enhance the user interface; so far little use has been made of them in programming environments.

The software requirements for a graphical user interface are more open-ended but include:

1. Bitmap graphics support: The software should include the ability to create, display and manipulate menus, icons, fonts and other graphic elements. Obviously this software will be designed to match any hardware support for \textit{bitblt}.

2. Window management: Software is required to control the display of information from multiple concurrent contexts and to respond to the input devices. As an illustration of the advantages of windows, consider the task of debugging a program. This normally requires access to at least some of the following forms of information: the program specification, the source code and possibly the object code, the input data and the corresponding output, and tracing information. With a window for each source of information, it is possible to display the relevant parts simultaneously.

It is sometimes desirable to copy or move information from one window to another – this is often referred to as \textit{cut and paste} operations. The released 5620 software does not do this, but under Edition 8 UNIX (the AT&T Bell Laboratories research version) the 5620 software combines editing, window management and terminal services in one program (Pike, 1984b). In this way, other user interface features such as shell history and escapes to invoke commands become unnecessary.

3. Communications: Information must be transferred between the user interface and the rest of the computer system. Common examples at the user level are terminal emulators for interaction with a remote host and file transfer programs. The 5620 uses a multiplexed protocol to permit the different layers share the single hardware communications resource. A serious limitation in the 5620 is the low bandwidth of the serial link. Each layer in the 5620 can be customised by downloading a program to operate in that layer. By default, each layer is a simple terminal emulator but much more powerful interfaces can be constructed.

The user level tools such as editors, compilers and debuggers will be built on top of this lower level support software.

4. Conclusions
This paper has surveyed some aspects of the interface between the software developer and computer-based support tools. The trend is to make the interface more flexible and accommodating for the human user by taking advantage of rapid developments in technology. The increased emphasis on graphics is intended to exploit human visual skills. Bitmap displays increase the range of output styles, while windowing systems allow the user to manage the visual context associated with several parallel activities. The aim is to provide more powerful user interfaces that improve the productivity of software developers and reduce the complexity associated with learning and using these interfaces. The process of developing new user interfaces is highly experimental. There is no possibility that an ideal interface can be abstractly defined and then implemented without experimentation. The ideas that have been discussed in this paper are not limited to programming environments; most of them are relevant to many other applications for interactive computing.
5. Acknowledgements
Much of the work underlying this paper was done while I was on a Special Study Program at ELXSI, San Jose, California. ELXSI provided an excellent environment for my study program and I would like to thank Len Shar, Robert Olsen, Don Wise, Russell Williams, and Mike Sherman for their assistance. One of my projects involved an evaluation of the AT&T 5620 DMD terminal and I am very grateful to Ken Fraser and Bob Aitchison of AT&T who made this possible. This work has also been partially supported by an ARGS grant.

References

Biographical Note
David Carrington is a lecturer in the Department of Computer Science at the University of New South Wales where he obtained his Ph.D. in Computer science in 1984. His research interests include software development environments and the interaction between tools and formal methodology. His Special Study Program at ELXSI during 1985 with their user interface group provided an opportunity to assess contemporary commercial practice.
An Overview of GED, A Language-Independent Syntax-Directed Editor

G. S. Moretti† and P. J. Lyons*

This paper describes GED, a full-screen syntax-directed editor for any LL(1) language that can be defined in Extended BNF (Backus Naur Form). The editor reads the syntax of the target language (in EBNF) augmented with print formatting commands. It displays a pretty-printed outline of the program, automatically inserting all required keywords and symbols. User input is checked on entry, making it impossible to generate syntactically incorrect programs. A ‘help’ area continuously displays the current production(s) and lists allowable start symbols. This display, a multi-level ‘undo’ command and the program outline generated by the editor provide a way of exploring the constructs of a language.

Keywords and Phrases: syntax-directed editor, programming environments, BNF, pretty-printing, language independence, LL(1)

CR Categories: D.2 (Software Engineering), D.3 (Programming Languages)

Introduction
GED is a program, written in Pascal to run on a Prime 750 computer, that has been designed to provide a language-independent syntax-directed editor that does not require a great amount of relearning to implement new languages. Previously, there have been two major approaches to implementing syntax-directed editing environments. In the first approach, the syntax of the target language is implicit in the structure of the editing program, and the editor is thus specific to that language. Examples are the original Cornell Synthesiser (Teitelbaum, Reps, 1981) and the COPAS system (Atkinson, North, 1981).

In the second approach, the syntax of the target language is rewritten in a form intelligible to an editor-generator and this, together with the editor procedures proper, is compiled to generate a new editor. The Synthesiser Generator (Reps, Teitelbaum, 1984) and the ALOE editor generator (Medina-Mora, Notkin 1981) are examples of this approach.

Only the second of these approaches is language-independent. However, the effort in implementing an editor for a new language is still considerable. The implementor must learn the generator’s input language and then translate the target language’s syntax into this language. GED avoids both of these time-consuming steps by reading the EBNF (Extended Backus Naur Form) definition of the target language directly. This metalanguage was chosen because it

Copyright © 1986, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ’s copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

†Computer Centre, Massey University, Palmerston North, New Zealand. *Computer Science Department, Massey University, Palmerston North, New Zealand. Manuscript received November, 1985; revised March, 1986.

commands need be learnt.

The system resulting from combining GED with an EBNF syntax is a full-screen template-oriented structure editor like the Cornell Synthesizer and Editor Alan Poe (Fischer, Johnson, Mauney, Pal, and Stock, 1981). It provides similar editing and help facilities (by constantly listing the correct input symbols), and has a multi-level 'undo' command. However, it will provide these facilities for any language defined in EBNF.

The Implementor's View
In the variant of Extended BNF used (Pagan, 1981), optional items or groups of items are enclosed in square brackets and repeated (groups of) items are enclosed in braces. To aid the editor's error recovery when parsing the EBNF grammar, a 'S' symbol is appended to each definition and 'SS' is used to signal the end of the grammar. Any line starting with a '%' is treated as a comment and ignored.

The editor creates a data structure that represents the EBNF definitions. No textual copy of the grammar is saved, as a printable representation may be regenerated from the data structure.

The first production of the grammar is taken as the root production. This conforms to the usual method of writing EBNF grammars. The examples that follow are based on the grammar for the Pascal-like language shown in Figure 1, which contains some lexical definitions (defined below). Grammars have also been written for Snobol, Lisp, and two local languages: one for a machine architecture description language and one for a data dictionary system.

The lexical syntax of the language's tokens is given in the optional first section of the grammar. This type of lexical information is sufficient for a wide variety of programming languages. If this section is omitted, the lexical defaults are those for Pascal (i.e. those given above, excluding the comma). It is assumed that the syntax of an identifier is:

\[
<\text{identifier}> ::= <\text{identifier}_\text{start}_\text{char}> \{(<\text{identifier}_\text{body}_\text{char}>\text{)}\}
\]

Having a separate delimiter set gives GED the flexibility to handle languages such as PLZ-SYS (Snook, Bass, Roberts, Nahapetian and Fay, 1978) which ignores newlines, commas, semicolons, colons, tabs and form-feed characters. Three types of comment convention may be defined:

1. Everything following a special character is ignored (use \texttt{START\_COMMENT} only).
2. Specially delimited lines in the program are comments (use \texttt{COMMENT\_COLUMN} and \texttt{START\_COMMENT}).
3. Comments are enclosed between special characters (use \texttt{START\_COMMENT} and \texttt{END\_COMMENT}).

The Skeletal Program
A skeletal program is derived by GED from its internal representation of the EBNF grammar. For the language defined in Figure 1, the skeletal program generated is shown in Figure 2. Note that required productions are expanded as far as possible. As \texttt{<block>} uniquely produces the sequence 'BEGIN <statement> END', this sequence is also included. The alternative expansions of \texttt{<statement>} prevent further automatic expansion.

All required terminal symbols, such as the THEN in an IF statement will be included automatically. This ensures that all bracketting symbol pairs (such as parentheses or BEGIN-END) are balanced. The display in Figure 2 has been formatted according to a common pretty-printing convention. The actual print-formatting commands that are added to the grammar will be given later.

The User's View
The user's VDU screen is divided into three areas; a program display area, a status line and a help area. The status line (blank in Figure 2) is used for error messages, file name prompts and general messages. The 'help' area shows the syntax of the current production and a list of its allowable start symbols.

As some GED commands operate on arbitrary-sized subsections of the program tree, a highlighted region on the screen is used as a cursor. In the examples given in this paper, we have underlined such regions (see \texttt{<program\_name> } in Figure 2).

User Input
Initially the cursor highlights the first placeholder in the program. The user may choose either to enter one of the possible start symbols, or to leave the placeholder unexpanded and to move to another insertion point.
To enter a symbol (symbols), the user simply types it (them) and presses the return key when finished. If correct, the symbol(s) will be incorporated into the program and the cursor will move to the next point of user input. Optional and iterated placeholders are only displayed when they are the current production (e.g. \texttt{<output file>} in Figure 3). If the current production is optional, the word ‘Nothing’ is appended to the list of correct symbols. The editor notifies the user of the first occurrence of an identifier with a beep and a message (Figure 3).

\begin{verbatim}
PROGRAM demo [ <output file> ] ;
BEGIN
  <statement> ;
END

First Occurrence of Identifier : demo
Correct Symbols : identifier Nothing
FD=HELP | Syntax ::= [ <output_file> ]
\end{verbatim}

Figure 3. Optional Placeholders are Visible under the Cursor

Errorneous Input
When a symbol is entered that cannot start the current production and the current placeholder is not optional, the editor beeps and an error message is displayed in the status area.

If the placeholder is optional, the editor searches forward from the current insertion point attempting to find a placeholder that starts with the input symbol. If a production is found, it is expanded; if not, an error message is given. The lookahead stops when a required placeholder is encountered.

The lookahead is illustrated by entering ‘IF’ while the cursor is on the ‘<output_file>’ placeholder (Figure 3). Only an identifier can start the current production but ‘IF’ is in the start set of \texttt{<statement>} and so the display changes to that shown in figure 4. As placeholders may be expanded in any order, commands are provided to move the cursor away from a placeholder without expanding it.

User Commands
Commands (except for ‘undo’) fall into two distinct classes. The first class (\textit{tree oriented}) treats the program as a tree structure. It includes commands for moving the cursor to the next or previous item at the same syntactic nesting level, deleting and inserting (previously deleted) subtrees; ascending the program tree (usually as a prelude to deleting a subtree) and elision. (Elided subtrees are displayed as ‘...’).

The second class (\textit{textually oriented}) treats the program as a text sequence. It includes commands for: moving the cursor from one line to another or one screen to another; moving the cursor to the next or previous insertion point (whether or not text has been inserted at this point); searching (for a specific token); setting and moving to markers at a particular place in the program.

Commands are available for undoing the previous command(s) (up to thirty) and repeating the last command. A command summary is available by pressing one of the function keys (F0). Although they have not been implemented, named macros (stored sequences of commands) would enable structural rearrangement to be made available by mimicking the commands a user would perform to alter one construct to another. Macros would provide the generality necessary in a language-independent environment.

Pretty-Printing Commands
The EBNF definition of any object in the syntax may have print formatting (pretty-printing) commands associated with it. This enables GED to display the developing program in a pleasing way. Note that the programs in the preceding figures have been pretty-printed, although the grammar (Figure 1) contains no print formatting commands. This would not normally occur. The print formatting commands are:

\begin{verbatim}
PROGRAM demo ;
BEGIN
  IF <expression> THEN
    <statement> ;
  END

First Occurrence of Identifier : demo
Correct Symbols : identifier number
FD=HELP | Syntax ::= <expression>
\end{verbatim}

Figure 4. After entering ‘IF’ while on ‘<output_file>’
A Language-Independent Syntax-Directed Editor

### Command Meaning

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Print associated terminal symbol, production name or trace the derivation of this production.</td>
</tr>
<tr>
<td>n</td>
<td>Skip to a new line.</td>
</tr>
<tr>
<td>m</td>
<td>Go to left margin.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Increase the indentation level.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Decrease the indentation level.</td>
</tr>
<tr>
<td>tnnn</td>
<td>Tab to column nnn.</td>
</tr>
<tr>
<td>s</td>
<td>Print one space.</td>
</tr>
<tr>
<td>c</td>
<td>Comments attached to this node are printed in their alternative position</td>
</tr>
</tbody>
</table>

The layout of the print-formatting commands is ignored, but usually follows the same form as the syntactic definition. Each formatting command is preceded by an ‘&’ followed by a number to indicate which syntactic item in the definition it is attached to. Figure 5, which defines a compound statement, includes print-formatting commands. (The asterisk indicates the start of a comment.)

Figure 5. Defining the Layout of a Compound Statement

### Placeholder Symbols

Symbols are displayed using EBNF syntax. An unexpanded non-terminal (e.g. `<statement>`) has its name printed as a placeholder whereas an expanded non-terminal has its derivation displayed instead. For example ‘IF <expression> THEN <statement>’ would replace ‘<statement>’. When a production has alternative derivations, its name is used as a placeholder. For example `<expression> ::= identifier | number’ would have ‘<expression>’ as a placeholder.

For optional and iterated items, the first item in the list, surrounded by ‘[... ]’ or ‘{ ... }’, is used as the placeholder (figure 6). The full production is displayed in the ‘help’ area.

Figure 6. Optional Items Form Their Own Placeholders

### Stratagems for the Syntax-Writer

As placeholders are generated automatically from the grammar, it is desirable to write productions to make the grammar self-promting. For example, many languages have a production of the following type:

```
<labelled statement> ::=[ <label> ] <statement>
```

The editor would use `[<label>]` as the prompt and the user would have to skip over it explicitly every time a statement could be entered.

Rewriting the production as:

```
<statement> ::= <unlabelled statement> | [ <label> <unlabelled statement> ]
```

<statement> becomes the prompt, but a label may be entered if one is desired.

In most cases, intermediate productions are not displayed, and all productions are expanded until they either derive terminal symbols or have alternatives. This has unfortunate consequences in Lisp, for example, in which virtually everything derives to the standard Lisp data structure `<s-expression>`. The problem is illustrated by the COND construct (figure 7).

```
<lisp function> ::= COND <predicate_result>
<predicate_result> ::= "( <predicate> <result> )"
```

If instead, the name of any production that directly derives one of the predefined items IDENTIFIER, NUMBER, STRING or a single non-terminal production (such as `<predicate>` and `<result>` which both derive `<s-expression>`) is printed, the much more readable version

```
COND ( <predicate> <result> )
```

is obtained. GED prematurely terminates the search of the program tree under the above conditions, and thereby allows productions to be introduced to act as prompts. This does not alter the target language but enables a small amount of help information to be incorporated directly into the EBNF grammar.

### Problems with EBNF Grammars

The major difficulties in using existing EBNF grammars as input to GED lie in the area of implicit syntactic restrictions. For example, the published EBNF syntax for the Pascal IF statement (Jensen, Wirth

---

1974) permits the following constructs to be generated:

\[
\begin{align*}
\text{IF [red, green, blue] THEN } \ldots \\
\text{IF 3 + TRUE THEN } \ldots 
\end{align*}
\]

These constructs have no meaning in Pascal (although some languages would attempt to perform 'appropriate' coercions) and are forbidden in the English language explanation accompanying the report. The problem could be alleviated slightly at the expense of a more complex EBNF definition. However it cannot be completely avoided in any syntax-only editor (without a symbol table and associated type-checking code).

Comments, while not executable, are permitted in most languages but are usually omitted from the syntax. The scheme initially implemented in GED was to have a special lexical item COMMENT, which built comments according to their lexical definition. The optional production [comment] was then inserted at key places in the EBNF syntax (e.g. after each declaration). This worked but was found to be unreasonably restrictive.

GED now permits comments to be added to any node in the program tree. This overcomes the need to specify the comment locations, but raises another question - should the comment be displayed before or after the subtree to which it is attached? The answer was found in the nature of the comments themselves. They can generally be divided into two types, block and trailing comments. Block comments usually occur before major syntactic constructs (e.g. procedures and statements) and explain what is to follow whereas trailing comments occur after some action. GED applies these criteria in the following way: a comment attached to a node corresponding to the first item in a EBNF definition is treated as a block comment and displayed before the subtree. All others are treated as trailing comments. This convention functions well as a default, but may be overridden by a printformatting command if necessary.

Currently only comment insertion and deletion are supported. However work on a EMACS-like editor (Stallman, 1981) for comments is well advanced.

Most published EBNF grammars are written with comprehensibility in mind, and are thus rarely LL(1). Some rewriting is usually necessary, most often to ensure that productions have unique start symbols.

Of the three types of shortcoming listed above, the informal syntactic restrictions are the most difficult to deal with. Rewriting the grammar to introduce meaningful prompts (e.g. <boolean variable> rather than <variable>) can make the intention of the grammar more evident, but the editor cannot ensure that these intentions are being adhered to.

**Syntax-Directed versus Text-Based Editing**

Once the initial work on implementing the core of the editor had been completed, our attention turned to the command structure. While the designers and those familiar with the syntactic view of languages found the editor to be usable, it was no competitor for the more familiar system editor (a full-screen EMACS lookalike). Initially GED was implemented with the (syntactically-oriented) cursor movement and tree editing commands usual in syntax-oriented editors, i.e. those for moving to the next/previous insertion points, next/previous placeholder, next/previous subtree, token searching, ascending/descending the program tree and subtree insertion/deletion. These provided all necessary operations but were confusing from the user's viewpoint, especially when moving the cursor around the program.

In an attempt to alleviate the problem, further syntax-based commands were added, but resulted in only minor improvement. It was finally recognised that users prefer to move around the program in a text-based fashion, and so commands for moving to next/previous line, to next/previous page and to a specific line number were added. These removed the need to move about the program in syntactic units and greatly improved the feeling of being in control.

A separate issue relating to the perceived user-friendliness of the editor was the initial decision to use function keys to invoke the various editor functions. These proved to be more of a distraction than a help and so EMACS-compatible key sequences were added for all insertion/deletion, movement, searching and utility commands. The more GED was made to emulate the main system (text) editor, the easier it became to use. This is consistent with the findings of Waters (1982).

Adding EMACS key sequences to GED greatly reduced the time taken to learn to use the editor, and meant that EMACS command sequences entered in error while using GED were interpreted reasonably. If both text and syntax-directed editors are in frequent use, it seems essential that commands given in one environment should be interpreted identically in the other environment, or at least not do anything that cannot be undone.

**Concealing Tree Structure**

It appears that the more an editor's syntax orientation is hidden, the easier it will be to use. Beginners, who make many syntactic errors, find a tree-oriented command structure confusing, whereas more experienced programmers make fewer syntactic errors and thus use the facility less. For the former group, the editor's value lies in its syntactic checking; for the latter, it lies in the pretty-printing and the automatic provision of redundant symbols. An exception is Lisp, where the editor's ability to provide prompting (e.g. <predicate> <result>) and automatic balancing of parentheses was found to be a definite asset.
Conclusion
The initial aim in this work was to produce a generalised syntax-directed editor that would be easy to set up for new languages. The editor which resulted is capable of providing a syntax-directed editing environment for any LL(1) language described in EBNF. This environment includes help and ‘undo’ facilities derived automatically from the grammar, which are therefore available for all languages.

The benefits of using EBNF were as originally conceived; that is, its familiarity and readability would reduce the time taken to produce an editor for new languages. These advantages are, however, not free. The EBNF grammar must be augmented with display formatting commands and a small number of lexical definitions. The task of massaging grammars for existing languages to work with GED is non-trivial (an average of one man-week for the languages mentioned), but it is enormously less than the time required to implement a custom designed editor. Once the language has been defined, small changes to the syntax and display formatting can be incorporated in minutes.

However, it is impossible to incorporate much semantic information into an EBNF grammar without major additions. Consequently the editor cannot provide any semantic checks or execution capabilities.

The resulting system has proven to be a worthwhile testbed for ideas relating to the facilities needed in a syntax-directed editor, in both the areas of editing commands structures and language-independent definitions.

References


Biographical Note
Giovanni Moretti trained in electronics before moving into computing and gaining a BSc from Auckland University and later an MSc from Massey University. He is currently employed as a Computer Scientist in the Massey University Computer Centre. His research interests lie in the areas of programming languages and expert systems, especially as applied to electronics.

Paul Lyons has an MSc in Chemistry, but has been lecturing in Computer Science, first at Auckland and then at Massey University, since 1978. His research interests are centred on the design and implementation of programming and syntax languages, computer-aided design of electronic circuits and computer networking.

An Adaptive Program Editor

J. Welsh†, G. A. Rose† & M. Lloyd*

A syntax-directed editor for the preparation of syntactically and static semantically correct programs is described. It differs from other editors described in the literature in that a pluralistic view of program structure is supported, which accommodates a number of development styles. Language and machine-independent aspects, such as efficient screen utilisation through adaptive formatting and detail suppression, incremental parsing and display, and minimal sequences to effect program changes are discussed.

Keywords and Phrases: syntax-directed editors, programming environments
CR Categories: D. 2. 2 (tools and techniques), D. 2. 6 (programming environments)

Background

Syntax-directed editors are seen as the key component of integrated programming environments, and considerable research effort has gone into their construction. In practice, however, many of the basic design issues noted by Marlin (1986) have yet to be resolved by widespread practical use.

This paper describes the design of an editor intended for use by professional programmers, in which user expectations and hardware limitations have been given careful consideration. Although many of the design decisions are independent of the language and machine involved, the specific development is of a Pascal editor for use on ICL Perq workstations.

The project is a collaborative one between the University of Queensland (UQ) and the University of Manchester Institute of Science and Technology, with financial support from the UK Science and Engineering Research Council and the Australian Research Grants Scheme.

Design Guidelines

The basic objective of any language-oriented program editor is to facilitate construction of programs that are correct from the syntactic and static semantic viewpoint of the language concerned. Existing editors have met this requirement in a variety of ways. In this section we examine the factors which seem likely to determine success or failure in this respect, and define the guidelines that determine the design decisions discussed in subsequent sections.

The implied discipline of program construction

A text-editor imposes no discipline on how it may be used for the construction of program texts, other than the basic constraint of left-to-right character input. In progressing to program-specific editors, the opportunity clearly exists to impose some discipline on how programs are constructed. For a specific user with an established program construction discipline, an editor which reinforces that discipline is clearly a boon. Establishing a consensus between users of what that discipline should be is another matter. At this stage, when the widespread replacement of text-editors by program-specific editors has still to take place, we believe that the editors offered should accommodate as wide a variation in construction discipline as is consistent with their basic objective defined above. To this end, the editor described here is designed to minimise the disciplinary constraints involved.

Variations in the user model of programs

When a text-editor is used for program construction, the conceptual structure of the program is purely the user's concern, and all interaction with the editor is expressed in terms of character manipulation. The final program, of course, must have a well-defined structure, as a tree of well-formed constructs, which the user must appreciate and work towards during the construction process.

When designing a program-specific editor, it is tempting to express all program manipulation in abstract structural terms—in effect to give the user a tree editor, within which character sequences are significant only as the leaves of the tree concerned. The editors developed within the Mentor project (Douzeau-Gouge, Huet, Kahn, Lang and Levy, 1975) typify this approach. In our view, the limitations of this approach arise from several sources:

1. The assumption that users are willing and able to think exclusively in tree terms is clearly questionable. In practice, many users have a pluralistic view of the programs they manipulate, seeing
them sometimes as tree structures, sometimes as symbol sequences, and sometimes as character texts, according to their purpose at that time. For such users, an editor that recognises only tree structures is no better than one that recognises only character texts.

2. Even if the user thinks exclusively in tree terms, the limitations of current display technology force the program display that the user sees to be in textual form, albeit with structural overtones achieved by indentation, highlighting, etc. This textual frame of reference makes the description of some relatively simple tree manipulations difficult (changing a while-statement to an if-statement is a classic case), and users find it easier to conceive the change required in textual terms (replacing while by if and do by then).

3. Problems also arise at the boundary between lexical and sub-lexical structure, particularly where trivial typing errors arise. A single character accidentally typed by the user should not require its correction to be expressed in higher-level structural terms, either at the moment of its typing or at any subsequent time.

As Marlin (1986) demonstrates, the facilities offered by the Cornell Program Synthesiser (Teitelbaum and Reps, 1981) are a compromise solution, in which higher-level language constructs are created and manipulated as predefined templates, while their lower-level components are handled as phrases, which are input and edited as text. While this approach avoids the adverse impact of tree-structure at lower levels, it tends to increase boundary problems, in that the user must now be aware of the distinction between the template and phrase levels, and use different techniques at each.

The argument that program editors should retain interfaces closer to that of a text-editor has been made by Waters (1982) and others. In our view the need to support a pluralistic view of program structure is a critical factor in the design of program editors, and will remain so until radical changes occur both in user training and in input/output technology.

Limitations of display devices
The limitations of current display technology do not just preclude a tree-like display. They also have a significant effect on the way in which a textual display is organised. While the sixty-line screens on workstations such as the Perq are a significant improvement on twenty-line screens, they still are very limited for the display and inspection of all but the most modest of program texts. If program editors are to play their full role in the programming process, as tools for reading programs as well as writing them, they must optimise the use of limited display capacity by economical formatting, by irrelevant detail suppression, and by judicious text comparison facilities. In our view, program editors to date have paid insufficient attention to optimising display utilisation.

Incremental operation and response time
A syntax-directed editor clearly does more processing of the text it receives than a text-editor. By doing much of this processing incrementally, as the user inputs or alters a program symbol by symbol, the editor can hide much of this work from the user, and provide a response which is only marginally slower than that of a text editor. Incremental processing has been identified as the key to program-specific editors from the outset, but it is important to be clear what we mean by incremental.

On the one hand, the user must realise that the response is at best proportional to the degree of structural or semantic change involved, not the number of keystrokes required to cause it. A minor textual change may radically alter the syntax or semantics of the program, with a consequent slow response by the editor.

On the other hand, the implementor must recognise that the need for incremental processing does not just apply to syntactic and semantic analysis, but to all processing involved in generating any response, be it parsing, unparsing, or whatever. If a user requests a completely new view of the program it is reasonable that the editor may take several seconds to generate it, but a minor adjustment of the current view must be instantaneous if user frustration is to be avoided. In general, we believe that achieving a response time consistent with the scale of the operation requested across all aspects of the editor is more important that the absolute response time achieved for any one aspect.

The factors discussed above are those which have determined the design choices made in developing the UQ program editor. Their impact on particular aspects of that system is discussed in the following sections.

Program Input
In determining how program input should be supported, the specific design objectives were:

1. to provide detection of syntactic and static-semantic errors in the program input, at the earliest point convenient to the user;
2. to reduce the keyboard effort required;
3. to avoid undue intrusion by the system for those users skilled in either language or keyboard use.

Input of program text proceeds by interaction with a predictive parser. The parser carries out an immediate expansion of the program string implied at each step as a sequence of inevitable lexical symbols interlaced with holes which represent symbol sequences still to be determined by the user. For a language like Pascal, whose major constructs begin with distinctive keywords, the dynamic generation of a hole/symbol sequence on receipt of the initial symbol has an effect equivalent to template selection in
some other systems. Its advantage is that it avoids any distinction between template and sub-template constructs as far as the system specification, and hence the user, is concerned. The difference observed by the user between symbols with a template-generating effect, such as program, procedure, if, while, etc., and those without is a direct consequence of the syntax rules concerned, rather than a partitioning of the language into template and sub-template classes by the system designer. In effect, the system adapts to the predictive capabilities of the language syntax concerned. For most modern programming languages, this adaptive approach seems as effective as the provision of designer-determined templates.

As noted above, the expansion of holes and symbols produced by the parser in response to the keyword program, say, is a complete template for a program. In keeping with the minimum intrusion strategy, however, the holes are not displayed to the user as explicit placeholders. Instead holes become blank space, and adjacent holes are coalesced so that the user is not aware of the sub-structure of hole sequences. Thus, after typing the symbol program the display that the user sees would be:

```
program ;
begin
end.
```

with the cursor (underscore) indicating the current input point.

This omission of explicit placeholders in the displayed representation of holes is consistent with the minimum intrusion objective, but may seem unhelpful to users unfamiliar with some aspects of the language. Instead, the UQ editor incorporates a help facility which during program input indicates, on request, the allowable inputs at the current point, as a syntactic description derived from the rules concerned, together with an associated informal semantic summary when appropriate.

Left to right filling of holes is the normal input pattern. Each symbol input to a hole may cause further expansion by the parser, with consequent subdivision of the hole and anticipation of further symbols.

Filling a given hole is completed by

1. input of all necessary symbols, if the sequence represented by the hole is self-delimiting, or
2. input of sufficient symbols from the user's viewpoint, if the sequence represented by the hole is not self-delimiting (e.g., if it finishes with an indefinite iteration).

In case (2), sufficiency is signalled by confirmation of the follower symbols already anticipated. An anticipated symbol may be confirmed either by overtyping it, or by use of a single keystroke <accept>. This flexibility allows good typists to ignore the display completely and touch-type the symbol sequence involved if they so choose, while allowing poor typists to minimise the keystrokes involved by exploiting the anticipatory powers of the editor.

The minimum input required of a poor typist to create the procedure:

```
procedure Check;
begin
  if Flag then count := count + 1
end;
```

is thus the 40 keystrokes: `procedure Check <accept> <accept> if Flag <accept> count := count + 1 <accept>

Typing the semicolon instead of the first <accept>s is also valid and perhaps slightly more efficient for most typists. For a touch typist, however, the 54 keystrokes: `procedure Check; begin if Flag then count := count + 1 end;` may be faster, and are again equally valid. (To generate the same well-formatted procedure with a text editor takes 63 keystrokes.)

The response to errors during program input is determined by the objective of imposing the minimum input discipline consistent with system effectiveness. To this end, each symbol representing a syntactic error is rejected immediately, as continued syntactic analysis with automatic error correction is of doubtful benefit to the user. Since correction of the majority of syntactic errors involves replacement of the symbol that causes the error's detection, this is effective system behaviour from the user's viewpoint.

For semantic errors, however, the correction required (such as a change to, or insertion of, a declaration) often involves a program change at a point quite different to the point of detection, and users prefer to defer such corrections to a later, more convenient time. Indeed the time at which users prefer semantic errors to be signalled is itself open to debate. Rather than have each semantic error signalled at the first point at which its existence is implied, arguments can be made for delaying semantic error signals to the end of significant semantic units, such as statements or even blocks, or indeed until the user requests their output. To this end, the design includes a configuration parameter, which can be set to individual user's tastes, to select semantic error reporting:

a. on a symbol by symbol basis,

b. once per statement or declaration,

c. once per block, or

d. on demand.

Whenever it occurs, semantic error reporting is non-intrusive, and involves output of an audible beep, display of the offending construct(s) in a distinctivive font, and display of a specific or a general error message depending on whether one or more distinct errors
An Adaptive Program Editor

are involved. When multiple errors exist, the user obtains the error message relevant to any one by highlighting the construct concerned.

In this way the editor will adapt to a variety of user disciplines in dealing with semantic errors. It should be noted, however, that the configuration parameter only determines the point at which errors are signalled to the user. Incremental semantic evaluation on a symbol by symbol basis may still be the implementation strategy used to ensure optimum system response when the signal point is reached.

Program Display

The manner in which the program is displayed by a program editor is as important as the means of program input or modification that it provides. In this area the user requirements may be summarised as automatic formatting, abstraction, and comparison facilities. All of these requirements must be met within the stringent limits of display capacity offered by current devices.

Automatic formatting

Automatic formatting is an essential feature of any program editor, but if simple fixed formats are adopted for all constructs, rapid consumption of the available display capacity results. To avoid this prodigal consumption of display capacity, the UQ editor employs the adaptive formatting system described by Rose and Welsh (1981). This allows trivial occurrences of potentially large constructs to assume one-line formats, while larger occurrences adopt the minimum multi-line form consistent with their extent.

For example, a Pascal for-statement may adopt any of the following formats, depending on the extent of its components:

\[
\begin{align*}
&\text{for variable } := \text{ initial to final do statement} \\
&\text{for variable } := \text{ initial to final do long or multiline statement} \\
&\text{for variable } := \text{ long initial value to final do statement} \\
&\text{for variable } := \text{ long initial value to equally long final value do statement}
\end{align*}
\]

Abstraction

To view any text that is larger than the available display capacity clearly requires some form of abstraction or windowing, but the simple scrollable windows employed by text editors are unsatisfactory for viewing computer programs. Two elements determine the optimum way in which a program view is presented: the highlight, which represents the user's current focus of attention, and the context, which relates the highlighted area to the overall program involved.

In the UQ editor, the guaranteed display of some easily identifiable context in which every highlight is rooted is seen as an essential requirement. For a language like Pascal an obvious mapping of contexts onto program, procedure or function blocks is available, with other substantial constructs such as record type definitions an additional possibility. In principle, therefore, every view presented to the user by the UQ editor has the general form:

\[
\begin{align*}
&\text{program/procedure/function } X; \\
&\quad \text{highlight} \\
&\quad \text{end } \{X\}.
\end{align*}
\]

To fit this combination of the highlighted text and its immediately enclosing context onto the available display area, some degree of detail suppression may be required. Such suppression is achieved in two ways: by structural level, and (as a last resort) by scrolling the text within the available viewing window.

While generalised level suppression algorithms, in which any construct is liable to suppression according to the level of nesting at which it occurs, have been used in systems such as Mentor, we believe it is preferable to limit automatic level suppression to distinguished suppressible constructs. Again, in Pascal the bodies of nested procedure and function declarations are obvious candidates for such suppression. In the UQ editor therefore, the following is a typical view of the outermost block of a Pascal program:

\[
\begin{align*}
&\text{program } \text{Example}(\text{input}, \text{output}); \\
&\quad \text{type data } = \text{ record key, info: integer end;} \\
&\quad \text{var NextValue: data;} \\
&\quad \text{procedure ReadValue(}\text{var v: data); }\ldots \\
&\quad \text{procedure ProcessValue(}\text{var v: data); }\ldots \\
&\quad \text{procedure WriteValue(v: data); }\ldots \\
&\quad \text{begin } \{\text{Example}\} \\
&\quad \quad \text{while not eof(}\text{input}) \text{ do} \\
&\quad \quad \quad \text{begin} \\
&\quad \quad \quad \quad \text{ReadValue(NextValue);} \\
&\quad \quad \quad \quad \text{ProcessValue(NextValue);} \\
&\quad \quad \quad \quad \text{WriteValue(NextValue)} \\
&\quad \quad \quad \text{end} \\
&\quad \text{end } \{\text{Example}\}.
\end{align*}
\]

The ellipsis points (...) after each procedure heading indicate that the corresponding procedure bodies have been suppressed by the display system.

To view any of these bodies, the user moves the highlight to (any part of) the construct immediately enclosing the ellipsis, i.e. the procedure declaration as illustrated above, and requests the editor to <zoom
in>. This replaces the current view by one in which the procedure concerned forms the context. In this new view, the bodies of any nested procedures or functions are again suppressed. To return to the context which encloses the current one, the user may request the editor to <zoom out> at any time. Additional commands <pan forward> and <pan back> enable direct movement between sibling contexts when required. Alternatively, absolute selection of a new context can be made at any time from a pull-down menu of all blocks in the program.

For some contexts, such as the outermost block of large Pascal programs, level suppression by itself may not generate a view which fits the available display area. In the current version of the editor, this problem is solved by resorting to the text-editor solution of scrolling the viewable text within the display window. During program input automatic scrolling is used to keep the insertion point in view. At other times the user controls the scrolling by use of a scroll bar displayed at the left hand side of the screen.

With level suppression, most viewable contexts in a well-structured program are displayable on a screen of reasonable size such as that provided by the Perq. For those that are not, text scrolling is not seen as an ideal means of viewing the structured text involved. For this reason, further suppression by structural distance is being investigated (Broom and Welsh, 1986). In suppression by distance, lines furthest from the highlight (in some structural sense) are suppressed first. With suitable parameters for the calculation of structural distance, and simple user control of the degree of 'focus' involved, we believe that an effective alternative to traditional scrolling can be provided.

Comparison facilities

The remaining user requirement to be met by the display system is the comparison of textually distant program fragments, typically located in distinct contexts. On workstations such as the Perq, the generation, maintenance and shuffling of multiple overlapping windows within the physical screen area is easily implemented. However the simultaneous display of two or more views is still subject to the physical limitations of display capacity. For this reason simple manual controls for opening and closing non-overlapping comparison windows have been adopted.

The creation of a comparative view is achieved by <cloning> the current view to a new window created in free space at the bottom of the display area, and then adjusting the view in either window using the normal context-controlling commands until the required comparison is obtained. The initial size of the new window is determined by the view to be displayed and the free display space available. Users may adjust the size thereafter by dragging the lower edge of the window up or down.

At any moment one of the views displayed is the active view, to which all context and highlight commands apply; users <select> a new active view simply by pointing to it. The active window may be <closed> at any time. All free space created by window closure or adjustment accumulates at the bottom of the display area.

Highlight Definition

As already implied by the discussion of program display, the user's focus of attention at any moment is represented as a highlighted area within the current display. The range of program fragments that can be highlighted are determined by the highlight-related operations provided by the editor. Given the liberal editing facilities to be described in the next section, the highlight itself must be definable in a variety of forms. To the user it may represent a single character, a sequence of characters, a single symbol, a sequence of symbols, or a well-formed construct. From the editor's point of view, however, the highlight is simply a sequence of one or more characters. The significance of the user's interpretation lies only in how the highlight is defined, rather than in the highlight itself.

The only limitation on allowable highlights is that the highlight must lie within the active context. This restriction seems consistent with the general concept of viewable contexts, and is of practical significance in few realistic situations.

With such a flexible definition of what the highlight may be, the significant design decision is how the user defines or redefines the highlight. In general this may be done absolutely, by pointing at the text to be highlighted, or relatively, in terms of the existing highlight position.

On a workstation with an effective pointing device, such as the Perq's tablet and puck, pointing is likely to be the dominant means of highlight redefinition. For any textual display, the smallest logical unit of pointer resolution is an individual character. However, from the users viewpoint, pointing at individual characters is not a convenient means of highlight definition in most cases. By using the three control buttons on the puck, the user is able to point at one of three levels:

1. character level, in which case the new highlight is the character pointed to;
2. symbol level, in which case the new highlight is the lexical symbol containing the character pointed to; and
3. construct level, in which case the highlight is the smallest sequence of symbols that forms a well-formed construct and includes the character pointed to.

Arbitrary sequences are indicated by 'dragging' the puck, with the appropriate button depressed, from one end of the sequence to the other.

Even with a good pointing device, however, relative movement or adjustment of the highlight remains preferable in some situations. The UQ editor therefore supports a set of commands for relative highlight movement, which reflect the alternative conceptual levels at which the user may wish to operate. These are summarised as follows:

- `<character left>`
- `<character right>`
- `<symbol left>`
- `<symbol right>`
- `<tree left>`
- `<tree right>`
- `<up>`
- `<down>`

The commands `<tree left>`, `<tree right>`, `<up>` and `<down>` provide conventional tree-walking of the visually distinct or concrete structure within the program.

In addition to their role in construct manipulation, the commands `<up>` and `<down>` provide a smooth transition between the character, symbol, and construct level. From any construct, repeated `<down>`s will reduce the highlight to the leftmost character of that construct, and from any character repeated `<up>`s will locate the enclosing construct at any level.

In overall terms, the relatively large number of highlight movement commands listed above is disturbing, and runs contrary to basic design principles for interactive systems. The commands chosen, however, do reflect the multiple conceptual levels at which the user may view a program text. Once the distinction between these levels is recognised by the user, the commands provide a flexible means of highlight manipulation. Given that these multiple levels exist, we believe the multiplicity of commands is preferable to a design which precludes operation at the intuitively appropriate level.

### Program Editing

The objectives adopted in determining the program editing facilities to be provided were:

1. that users should be able to conceive and express edit operations in an intuitively natural manner, on well-formed constructs, symbols, or characters as appropriate;
2. that the sequence of changes involved in any program transformation should be subject to minimum disciplinary constraints consistent with effective editor operation.

Objective (1) supports the pluralistic user model of program discussed under design guidelines, and given the highlight definition already discussed, is realised in practice by allowing users to `<delete>`, `<insert before>`, `<append after>`, or `<change>` any highlightable program fragment. Depending on the nature of the highlight or of the text inserted, the user may perceive the change involved as a tree manipulation, a symbol sequence alteration, or a character-based textual alteration, but is not required to distinguish which it is.

In the case of `<insert>`, `<append>`, or `<change>`, the text inserted is parsed in exactly the manner described for program input. On receipt of an `<end insert>`, however, the original program tail is 'joined' to the extended program head at the input cursor position, so overriding any new anticipated tail that has been created.

The effect of `<delete>` is simply to 'join' the program preceding the highlight to the fragment that follows it, so eliminating the highlighted text itself.

Objective (2) determines the conditions under which an edit operation is valid, and the precise semantics of the 'join' operation:

- In the case of `<delete>`, no pre- or post-conditions are imposed on the operation. The user may delete any highlightable fragment of the program, and no requirement is imposed that the program is correct in any sense after deletion.
- For `<insert>`, `<append>`, and `<change>`, however, a precondition must be applied, to ensure that the parser can be run effectively on the inserted program text. In effect this precondition is that the program head, i.e., up to the point of insertion, must be 'sufficiently correct' to determine an unambiguous initial parser state for the insertion which follows it. In its simplest implementation, 'sufficient correctness' usually means that syntactic errors in the program head cannot be tolerated, but semantic errors may be.
- As with `<delete>`, no post-condition is applied to insertions, so that the inserted text does not have to align syntactically or semantically with the text that follows it.

This definition of editing facilities is consistent with the minimum disciplinary constraints objective, but does mean that overall program correctness is not guaranteed after any edit operation. The 'join' operation which follows insertion or deletion simply ensures that the overall program remains displayable. To recover the guarantee of overall correctness the user may request a program `<recheck>` when appropriate.

These commands allow users to maintain intermediate program correctness to whatever degree they choose, but do not in general impose specific edit disciplines upon them. The only effective discipline imposed arises from the required precondition for insertion operations. This means, for example, that in inserting a `<repeat>` and an `<until>` clause around an existing sequence of statements, the `<repeat>` must be inserted first, otherwise the `<until>` will be rejected by the parser. This left-to-right discipline for the build-up of bracketing constructs is unlikely to trouble most users, but the general constraint implied by the insertion precondition may in some circumstances.

---

The editing facilities described were arrived at by completely pragmatic pursuit of the pluralistic-user-model and minimum-dicipline objectives within the constraints of a straightforward parsing strategy. It is interesting to note that, from the user viewpoint, the facilities provided are very close to those described for the Tektronix editor (Morris and Schwartz, 1981). They differ in that movement of the highlight into the tail of the program after an edit operation does not in itself imply any extended correctness requirement. Deletions may thus be made in an incorrect tail, and correctness up to the highlight is only required when an insertion is requested.

From the user viewpoint, the approach may still seem unsatisfactory both in the constraint implicit in the insertion precondition, and in the lack of automatic detection of inconsistencies beyond the point of edit. To this end a more sophisticated parsing strategy which uses a more liberal 'sufficient correctness' precondition, and provides detection of post-edit inconsistencies within a fully incremental mode of operation, is now being evaluated (Kiong and Welsh, 1986) and may be incorporated in future versions of the editor.

Implementation Summary
All features of the editor described are operational at the time of writing except the static semantic checking. Integration of semantics, and a number of performance improvements are now being carried out.

The technology used in implementation is conservative, exploiting as far as possible techniques, and code, proven in previous prototype systems. Input parsing is implemented by a table-driven LL(1) parser which stores its complete parse state at any moment within the representation of the anticipated program tail. 'Incremental' parsing is achieved by leaving copies of this tail at appropriate restart points in the internal program representation. The behaviour obtained by this technique is not strictly incremental in the sense defined earlier, but is effective in most situations. Replacement of this simple strategy by the incremental parser described by Kiong and Welsh (1986) will be considered for later versions of the editor.

Semantic analysis is being integrated using a package of semantic routines derived from the model implementation of Pascal described by Welsh and Hay (1986), which have been extensively tested against the Pascal Validation Suite. These routines have been adapted to form pre- and post-actions for the non-terminals of the grammar; communicating with one another through a single semantic stack. These actions are activated by traversal of the program representation generated by the parser. Incremental semantic analysis following edits with pervasive semantic effects, such as alterations to declarations, is on a per-block basis, being directed by identifier dependence sets maintained for each block. In this way acceptably incremental semantic evaluation is achieved at a storage cost which is little more than the symbol tables maintained by an offline compiler.

Unparsing of program blocks for display is implemented using the formatting algorithm described by Rose and Welsh (1981) with minor extensions to support context delimiters and level suppression. Mapping the formatted output to the multi-windowed display structure supported by the editor poses few additional problems.

The implementation strategy is partially generic, in the following respects:

a. all user commands, and all response generation is language-independent, and relies only on a few general assumptions about block-structured languages;

b. all syntactic, format and context defining details for the language concerned are input as a set of EBNF rules, augmented with format information as suggested by Rose and Welsh (1981);

c. the parser relies on a language-specific lexical scanner routine which is hand-coded mainly for efficiency of input from text files when required;

d. the semantic action routines are clearly language-specific, with the semantic driver relying only on a one-to-one correspondence between actions and non-terminal syntactic structure for their activation.

Thus, to adapt the editor to another Pascal-like language, an augmented EBNF syntax definition is required, together with a plug-in lexical scanner and semantic package for that language. While the latter is clearly a major implementation effort, the language-independent editor framework is still an attractive starting point for any such project.

Conclusions
The features of the UQ editor described in this paper differ significantly from those of other editors described in the literature, and these differences stem directly from the design objectives adopted. The objectives themselves are determined by largely pragmatic considerations, such as apparent user behaviour, technological limitations, and transitional factors in the introduction of program editors. As such, the editor may be seen as an evolutionary step towards the widespread use of program editors, rather than a revolutionary tool that breaks the mould of current program construction methods. Given the influence that user reaction has on the success of all such optional software tools, we await its release with interest.

References


---

**Biographical Note**

Jim Welsh was Reader in Computer Science at the Queen's University of Belfast, and Professor of Software Engineering at the University of Manchester Institute of Science and Technology, before moving to the University of Queensland in 1984. His interests are programming languages, language implementation, and programming tools.

Gordon Rose was appointed foundation Professor of Computer Science at the University of Queensland in 1969. Previously, he held academic positions at the University of Adelaide and the University of New South Wales. His interests are programming environments and the formal specification of software and systems.

Melfyn Lloyd worked in local government before graduating with first class honours in Computation from the University of Manchester Institute of Science and Technology in 1983. He now is a senior consultant with the UK's National Computing Centre.
Kay: A Beginner's Environment

B. Dwyer†

Kay is a complete programming environment, designed to teach raw beginners the principles of structured programming. It is an interesting experiment in applying 'learning theory'—specifically 'operant conditioning'. An interactive computer environment is a potent modifier of behaviour. Kay's responses have been designed to emphasise reward rather than punishment, so that is provides a pleasant environment. The same approach could be applied to larger systems, with advantage.


CR Categories: D.2.5 Diagnostics, D.2.6 Programming Environments, K.3.1 Computer-assisted instruction, K.3.2 Computer Science Education.

1. Introduction

The Kay programming environment is deliberately a simple one. Even so, it is interesting, because it adopts a scientific approach to the idea of 'user-friendliness'. The central issue in its design is the process by which a student learns what Kay can do. Ideally, someone who knows nothing about Kay will learn all that there is to know simply by using it. This is a goal that authors of larger systems should aim for—and by this, I don't simply mean giving their users access to 'help' files. An interactive computer system is a potent tool for behaviour modification; it had better teach what is wanted. This is a practical illustration of how it can be exploited in a medium-sized environment.

What's in a name? In this case, Kay's father is Karel the Robot (Pattis, 1981), who is himself descended from a Logo turtle. Kay is a programmable robot that has 'adventures' on a 24 by 80 visual-display screen. Kay's behaviour is almost identical to Karel's; but whereas Karel was designed for pencil and paper exercises, Kay is designed for typists whose technique can usually be described as 'seek and destroy'. To help them, Kay allows identifiers to be abbreviated by ignoring any small letters that they contain. Thus the original 'Karel' became shortened to 'K', but later grew again to 'Kay'. Unlike Karel, Kay is a non-sexist robot, for although 'Kay' is often a girl's name, Sir Kay was one of King Arthur's knights, who went on quests and usually got lost—exactly what Kay's programmers achieve.

Kay was developed to help teach first-year students at the University of Adelaide the principles of structured programming. We are not able to assume that these students have any programming background, so Kay is used to introduce control-flow concepts in preparation for teaching Pascal. The assumption that this approach is useful was originally made by Pattis in designing Karel. In adapting it, I have made the same assumption. Kay is also used later in the course to help teach step-wise refinement and problem-solving. Its advantages are that it covers a lot of ideas with little typing, and the semantics of incorrect programs are immediately obvious to the would-be programmer.

Kay is the outcome of experience that we gained with a computer implementation of Karel the Robot (Brumley, 1983). (Pattis himself has also written an implementation (Pattis, 1981).) It proved to be an effective means of teaching problem-solving, but was unsuitable for beginners. One reason is that even experienced typists find Karel's language verbose. Students first had to learn how to use an editor. A special program was needed to set up Karel problems, which were then stored on files. In total, four languages had to be learned in order to use our Karel system; Karel's language, the editor commands, the problem specification language, and a few operating system commands—a confusing mixture for a beginner. Kay overcame these difficulties by using special characters for keywords, allowing abbreviations for names, dispensing with the editor, and by giving Kay the ability to set up problems for itself. Kay can be invoked automatically when the student logs-in, and handles files transparently. Only one language and a minimal set of concepts are involved.

Kay needs a 24 by 80 visual-display terminal with cursor addressing and the ASCII character set. A session begins with Kay reading a file containing problems and attempted solutions; it ends by saving an updated version. During the session, the student can create new procedures, test them, replace them, or delete them. Most of the time, the screen displays

Copyright © 1986, Australian Computer Society Inc.

General permission to republish, but not for profit, all or part of this material is granted, provided that the ACJ's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

†Department of Computer Science, University of Adelaide, P.O. Box 498, South Australia 5001. Manuscript received November, 1985; revised March, 1986.
Kay is facing East at 1 North, 1 East. Kay's bag has 1000 markers in it.

Type a command, then hit RETURN.

Press space bar to go fast. Hold CTRL and hit: S, freeze; Q, melt; C, cancel.

Figure 1

three windows; the 'grid', a message area, and a command area. Its general appearance is shown in Figure 1. The command window occupies two full lines across the bottom of the screen, the other two windows are side by side above it.

One line of the command area is used to enter commands, which are thus limited to one line in length. For good or ill, this forces students to write short simple procedures. The other line is often used to display previously typed commands.

The grid window displays a square of 21 by 21 points. Kay (the robot) can move about this grid under program control. If it steps out of the window, the window is moved to a new part of the grid. The grid can be as large as the implementation of integers allows, or it can be given boundaries. Apart from Kay, which is represented by an arrow-head (> , ^, <, or v), the grid can contain 'markers' (o), which Kay can move, and 'blocks' (#), which it cannot. Blocks are used to make walls, obstructions and mazes. Markers are collected, moved about, arranged in patterns, used to mark trails, and to count. (The + signs that appear in Figure 1 mark grid co-ordinates that are multiples of 5.)

The message area is used to display information about the status of Kay, error diagnostics, or 'help'. It is never empty.

2. Goals
We must distinguish the goals of Kay the Robot, as part of a course in computer programming, from the more limited goals of the Kay environment, and also from the goals of the student.

The teaching objectives that motivate the use of Kay are to introduce the program structures of sequence, selection, iteration, and procedure, and to teach problem-solving using step-wise refinement. These objectives are met through lectures and practical exercises using Kay. There are other implicit objectives; giving the student confidence, practice at using a computer terminal, and firing the student's interest. Because lecture time is valuable, the instructor concentrates on concepts, leaving the details to be taught by the Kay environment.

The specific objectives of the Kay environment are therefore to teach the student the correct syntax of commands, and their semantics. In this respect, Kay has the same objectives as any programming environment should. Kay's syntax can be summarised by five rules for forming statements, and five rules for forming commands. (These are summarised in Appendix A.) The rules are expressed in extended BNF notation, which the Kay environment must also teach.

The student's (short-term) goal is to solve a series of problems set by the instructor. The intellectual effort involved is graded from simple copying, through adapting existing procedures, to creative problem-solving.

The psychological basis of the design is the (experimentally verified) learning theory principle of 'operant conditioning' (Skinner, 1938). Briefly, it is that when an action is immediately followed by a reward, its...
future probability is increased; but when it is followed by a punishment, it is decreased. It does not predict that when incorrect actions are punished, correct actions will automatically take their place. As the author has pointed out elsewhere (Dwyer, 1981), when an environment tries to modify behaviour by punishing mistakes, users may learn simply to avoid using it.

Learning theory is incorporated into Kay's design by rewarding success rather than punishing failure. What is a reward depends on the individual, but in a problem-solving activity, it is usually accepted that knowing that progress has been made, or exercising control, is a reward (Shneiderman, 1979). Kay rewards progress by giving an immediate visible or audible response to every action by the student. If an action is incorrect, Kay guides the student towards constructing a correct one. Attention is drawn not to what is wrong, but to what would be right. (An example will be discussed in Section 4.) There is also a 'help' facility, whose messages can depend on context. For example, if the student asks for information about any of Kay's built-in condition tests, the text that is displayed will say whether the condition is currently true or false.

Shneiderman has shown that the phrasing of error messages has a strong effect on the time taken to correct errors (Shneiderman, 1982). He showed that the 'tone' of the message was important, and so was its specificity. He compared the time taken by programmers to correct errors, given diagnostics issued by a compiler, with the time taken using 'doctored' compiler output containing improved messages. The improved messages avoided implicit criticism of the programmer, or gave more detailed information about the source of the error. Unfortunately, it is not easy to see how a compiler could be smart enough to issue some of his doctored messages without making an occasional 'howler.'

Error messages are usually phrased as if the user knew what to do, but made a mistake. It is just as likely that the user didn't know what to do at all. In the design of Kay's error messages, it is always assumed that mistakes occur because the user lacks information. This may sometimes be too charitable, but it is a harmless assumption. Experiment has shown that error messages that inform by stating rules are at least as good as the best that can be designed by following the criteria recommended by Shneiderman (Clow, 1982). They are not significantly better, but they don't require the computer to be smart.

3. Using Kay
Kay has built-in primitive actions, such as 'TurnLeft' and 'Move'. Any action is also a statement, and also a command. The simplest Kay command is simply 'Move', or more simply, 'M'. A fixed iteration is specified by preceding a statement by an integer constant, so '3TL' will make Kay turn left three times, effectively turning right. The student can define this as a new procedure by typing '! TurnRight = (3TL)'. A procedure definition must be typed on a single line. Because it is assumed that most users will be poor typists, compromises must be made between clarity and brevity. Kay allows a student to define a procedure with the name 'TurnRight' and then refer to it as 'TR'. The directory will then show the full name, and so will the '#TR' (show 'TR') command.

Any rule that allows abbreviations must be 'stable'. That is, once an abbreviation has been recognised, it should retain its original meaning, despite the future creation of similar names. Kay's rule is to ignore all small letters. Consequently, if the student later tried to define a procedure called 'TurnRound', Kay would invite the student to choose a different name, or to lose the 'TurnRight' procedure.

A Kay program is simply an unstructured collection of procedure definitions, activated by typing a statement that invokes them. All procedure names are global. There are no named data objects, no variables, no expressions and no parameters. Kay's built-in commands comprise those inherited from Karel, some new ones that are used to set up problems, and a few, such as 'Directory', which display or print information. (These are summarised in Appendix B.)

In addition to fixed iteration, just illustrated, Kay allows while ... do, if ... then, if ... then ... else, and begin ... end structures. For ease of typing, these are indicated by special characters rather than keywords, for example '(' and ')' replace begin and end.

The only editing that Kay provides is a 'rub out' or 'delete' key, and to create, replace, or erase procedure definitions. There is no facility for amending them. This is a conscious choice. First, what isn't provided doesn't need to be learned. Second, it encourages short procedures. Third, and important from the learning theory viewpoint, it forces the student to rehearse constructing correct statements. This is a case of being cruel to be kind. To speak from my own experience, I have always written a new Cobol program by amending an existing one. Consequently, after fifteen years practice, I still cannot remember the correct sequence of sections and paragraphs in a Cobol environment division. Being able to correct programs may be convenient, but it does not help students learn syntax.

The student sets up a problem by invoking a procedure already written by the instructor. The student may then experiment by typing single commands or structured statements to study Kay's behaviour, eventually defining one or more new procedures. Kay will remember these procedures until they are replaced or erased. Statements are checked for correct syntax and compiled (into internal tree structures) as they are
entered from the keyboard. Kay also keeps the text of each procedure definition, precisely as it was typed. References to procedures are resolved as pointers to symbol table entries. If no entry is already present, a new one is created that will cause a run-time diagnostic if an attempt is made to execute the procedure. Kay therefore supports either a top-down or a bottom-up approach to problem-solving and program development.

From this description, it will be seen that potential errors can be divided into a number of categories. Syntax errors are those that can be detected without executing a statement. These can result from breaking the rules of Kay's grammar, or from using the name of an action or condition inappropriately; for example, from trying to replace a built-in condition by a new procedure. Semantic errors are those detected during statement execution. They can result from executing an undefined procedure, or from asking Kay to do the impossible; for example, to pick up a marker that isn't there. An error that cannot be detected by Kay is an 'error of intent', that is, a failure to solve the problem. The Kay environment must address the issue of helping the student recover from each of these kinds of error.

4. Shaping the Student's Behaviour
It is impossible to know what is in the student's mind (if anything) when a syntax error occurs. Attention must therefore be focussed on how the error was detected. Kay uses an LL(1) grammar, and a predictive parser that places the expected right-hand side of each production on a stack. For example, when Kay finds a left-parenthesis (meaning begin) in its correct context, it places '(' stmt-list ') on the stack. At all times, the stack predicts what valid continuations of the text are possible. When a symbol is found that does not match the prediction, the following are known: any characters before the error are a valid start to some statement, the stack contains a valid way of continuing it, and the symbol, in isolation, may itself have a valid use. All this information is potentially useful to the student.

Figure 2 shows Kay's response to a syntax error. Suppose that the student should have typed '!TurnRight=[3TL]' but used square brackets instead of parentheses. Since square brackets are used to describe syntax, Kay's response to this is as confusing as it ever gets.

My thesis is that diagnostics shape learning; so I need to defend every phrase of this message. First, the student is not reprimanded. Kay, not the student, takes the blame, for not being able to read what was typed. (The heading, 'SYNTAX ERROR:' does not help the student; it is there so that the student will eventually learn what the words mean.)

Attention is then focussed on the symbol where the error was found. This has a meaning in the Kay environment, so it is unlikely that the student did not mean to use it. Therefore, the message gives examples of its use. It is a pity that Kay always gives the same examples; a more appropriate one could have been chosen here.

Then Kay fixes on the part of the command that appeared to be correct. This is shown in the command window, because of its potential length. The prediction in Kay's stack appears in the last line. (The stack contents are first translated into terms that the student should recognise.) This tells the student the rules that Kay was using, and, with luck, how to correct the problem.

(Although the braces and brackets used to describe syntax are themselves a concept to be learned, they are an investment for the future; they can be used later to describe Pascal, or other languages. But because they are part of what the student must learn, Kay always explains them when they appear in a message.)

Finally, if the message is not enough in itself, it advises the student how to obtain more help.

When the student presses the Return key, the message in Figure 2 disappears, and is replaced by a standard display that describes Kay's status. The prediction derived from Kay's parser remains visible to guide the student in constructing a correct command. This is a clear example of 'shaping'. The correct part of the student's action is re-enforced, and the student is shown how to improve. A 'chain of association' is formed, from the student's intention, to the appropriate rule, to its correct application.

The other kind of syntax error Kay detects is that of using a name in the wrong context. For example, if the student uses (an abbreviation of) the name of an action where a condition test is needed, Kay lists all the valid condition tests. In other words, whenever Kay knows something that might help the student, it displays it, or tells the student how to find it. Unaided, the student might not know where to seek help.
One kind of semantic error occurs when Kay is directed to do something impossible. For example, if Kay is directed to walk into a block, the message in Figure 3 will appear, beginning with a display of the procedure stack. Again, Kay tries to shape the student's behaviour by localising the problem, and suggesting a corrective action. A second kind of semantic error is to invoke an undefined procedure. Here, Kay reminds the student how to define a new procedure.

Procedure Stack ...
  CollectMarkers
  GoIntoHouse

BLOCKED
You tried to make Kay move into a blocked area.
This is an error.
Your procedure has been halted, as above.
You could test 'FrontClear' before 'Move',
if you want to be sure it is safe to do it.
Waiting for you to press RETURN.

Figure 3

Finally, the student can make errors of intent. Most of these are clearly visible from Kay's behaviour on the screen. Kay normally makes one program step per second, so its action is easy to follow. The student can speed up or freeze the action during execution, as desired. The message area is continuously updated to display the procedure stack. If the student can see where Kay goes wrong, freezing the action will also freeze the stack display, helping to isolate the error. If Kay gets stuck in a loop, the student can interrupt it, and cancel the command.

Despite the obvious effects of most commands, there are a number of occasions when Kay might appear to do nothing. It is important that the system should still show a response. One of these occasions is when the student defines a new procedure. Kay simply displays the new definition again, below the command line—anything is better than nothing. A second occasion is when the Kay program starts up. Because only the texts of procedures are saved on file, they must be recompiled. If the computer system is busy, and there are many procedures, this can take an embarrassingly long time. The message area tells the student to be patient, but it is easy to believe that nothing is happening. So during this operation a picture appears on the screen, showing Kay reading. Kay's eyes move. This is admittedly nonsense, but it is a clear sign that the system is working. For symmetry, when a Kay session finishes, Kay is seen to write—with a pencil. I have no excuse for this; saving procedures on a file actually takes very little time, Kay does not really have a pencil, nor is there any learning theory basis for it. I just felt that students would need assuring that Kay was taking good care of their work.

5. Results
Kay was originally implemented by a Pascal program of just over 2100 lines. At the University of Adelaide it runs on a Digital VAX, under VMS. The executable image occupies 67K bytes of virtual memory. (The same image is shared by all students currently using it.) Robyn Deed at Flinders University has adapted Kay for the Prime and Pyramid computers. The conversion was not trivial; the original program takes advantage of some special features of VAX Pascal.

My impression is that Kay has been a successful teacher. In previous years, the first few weeks of practical work have been bedlam. Now, supervisors complain of being bored. Students have worked quietly, encountering few problems that they could not solve for themselves. Many have gained confidence by finding that their first encounter with a computer was rewarding. This was not without cost. Three-quarters of the Kay program are its diagnostic and help routines. With over 300 students per year, the investment was worthwhile. The benefit was not only to the students; the teaching staff had less questions to answer. Of course, Kay teaches only one aspect of programming. Teaching staff later had to answer many questions about Pascal variables, types, and parameter passing. Few questions concerned control structures, which indicates that Kay did what it was supposed to do.

Making the transition to Pascal was an important part of the overall teaching program. The correspondence between the Kay's and Pascal's different ways of representing the same control structures was explained in lectures. This seemed to be effective, except that some students who had no trouble in using semi-colons correctly in Kay's syntax failed to apply the equivalent rules consistently in Pascal. Perhaps this was because Pascal also uses semi-colons in other contexts. However, students' confusion about semi-colons did seem less. I had feared that students would wrongly try to apply Kay's rule for abbreviating names to Pascal identifiers. But if it happened, it was not evident.

Although Pascal's and Kay's control structures are similar, Pascal's primitives—reading, writing and arithmetic—are quite unlike Kay's. A screen-based program demonstrated them. A window showed a list of Pascal input, output, and assignment statements, which students had to select for execution according to the demands of an algorithm. Other windows showed the values of variables, the input and output buffers, and the terminal screen. Like Kay, this program made semantics visible, even the rather subtle behaviour of Pascal's buffer pointers. Students then had to prepare the same program for execution using an editor and the Pascal compiler in the regular way. After this they were considered ready to develop—in stages—a large Pascal program to play
the game of 'Othello'. No design skills were required; a working Pascal program had been translated line by line into English, and students had simply to reverse the translation.

Until this point of the course, Kay could be claimed to have saved students almost exactly as much time as it had cost them to learn it. But its usefulness to a first course on computer science was not over. It was used in a segment in problem-solving; the most difficult exercise being to make Kay find goals in mazes like that of Figure 1. Finding a solution would have taken much longer in Pascal, especially to debug. Kay has two advantages, which it inherits from Logo and Karel. Some problems have a visual pattern that suggests to the eye a suitable decomposition—the program structure is self-evident. On other occasions, the student can see the problem from the robot's frame of reference—to adapt a phrase from Papert, by 'playing robot' (Papert 1980).

That students had learned two languages helped in later lectures; for example, those about syntax analysis. It was also possible to discuss program verification, and the formal properties of control structures; for example, the rule that the condition guarding a while loop is always false when the loop terminates. Such observations are easily understood in Kay's simple environment, especially as it provides no operators with which to form compound conditions.

Software authors should stop thinking of diagnostics as numbered one-line messages, added as an afterthought; this was appropriate when memory space was limited and messages were stored on external files. Virtual memory systems allow each message to be tailored to the specific situation. Error diagnostics are an important means by which a user learns to use a system. Software writers should devise system responses that re-enforce correct action, give the user rules to follow, or display status information. They should begin by designing all system responses, and make environments better teachers.

References

Appendix A. Kay's Syntax

Execute command: stmt
Define command: name= (stmt (; stmt))
Erase command: name=
Show command: @name
Exit command: .
Simple Statement: name
Iteration Statement: number stmt
Compound Statement: (stmt (; stmt))
If Statement: ?["test", stmt ; /stmt]
While Statement: @["test", stmt]

Appendix B. Kay's Built-in Actions & Conditions

Actions
Appear ............... makes Kay become (more) visible
Block ................ creates a block where Kay is standing
Create ................ creates a marker inside Kay's bag
ClearGrid .............. initialises the grid, the markers, & Kay's position
Directory .............. displays all the names that Kay knows about
Edge ...................... reduces the size of the grid
Format ................ summarises the rules for typing statements
Grid ...................... prints a snapshot of the grid
Help ...................... summarises the rules for typing commands
Join ...................... joins two edges of the grid
List ...................... prints all the procedure definitions that Kay knows
Move ...................... makes Kay move one step ahead
Noise ...................... makes Kay make a noise
PutDown .............. makes Kay put a marker from its bag onto the grid
PickUp .................. makes Kay take a marker from the grid into its bag
Stop ...................... makes Kay cancel your procedure
TurnLeft .............. makes Kay turn left by one right-angle
Vanish .................. makes Kay become (more) invisible

Conditions
AtMarker ................ true iff Kay is standing at a marker
BagEmpty ................ true iff Kay's bag has no markers in it
FrontClear ............. true iff there is no edge or block ahead of Kay
FacingEast .............. true iff Kay is facing East
FacingNorth ............ true iff Kay is facing North
FacingSouth ............. true iff Kay is facing South
FacingWest ............. true iff Kay is facing West
LeftClear .............. true iff there is no edge or block to Kay's left
RightClear ............. true iff there is no edge or block to Kay's right

Biographical Note
Barry Dwyer is a Senior Lecturer in Computer Science at the University of Adelaide. He received his Masters degree from Oxford University in 1959. He also has a Diploma in Advanced Engineering from Cranfield, and a Graduate Diploma in Business Administration from the South Australian Institute of Technology. Following graduation, he worked as an electronics engineer, but soon became more interested in software. He worked closely with Michael Jackson (of the design technique) from 1965 to 1971. After immigrating in 1972, he worked for a time as a Systems Analyst, then began teaching in 1978. In 1980 he spent a year as Visiting Researcher at Bell Laboratories in Murray Hill, New Jersey. He is interested in program and system design techniques, and harpsichords.
Recording the Computational History of a Research Project

B.G. Cook †

In research projects with substantial computational requirements, it is important that a detailed and accurate record be maintained to enable a thorough audit of any part of the conclusion-reaching process. The paper discusses the documentation needs of such projects and describes a prototype project control system which has been used to automatically document the computations of a long-term natural resources research project.

Keywords and phrases: Programming environments, project control, project documentation, computational history.


INTRODUCTION

The project control system described arose from the need to document and control the computations for a long-term research project involving a number of participants. An earlier similar research project had shown how easily informal record keeping could result in doubts or disagreement on what procedural path had been taken to a particular end point. This is particularly a problem when methods used evolve during the course of a project and a number of computational paths may be tried before a final course is decided upon. Another feature of research projects, which distinguishes them from production systems, is that it is often impracticable to thoroughly test programs which are to be used only once, or a few times.

In this situation, it is very important that a complete record be maintained to allow an audit to be carried out of the computing procedures used in reaching a particular conclusion.

An early response to this need was to propose the storing of all programs, datasets and job control listings in a large data base, which would thus contain a complete record of the data processing of the project. Although this approach was rejected as impractical, from it developed the concept of a data base which recorded the location of programs, datasets and job control procedures, together with the necessary additional data and relationships to define the sequence of computational procedures carried out. Instead of using normal operating system commands to attach input datasets, initiate execution of programs or save output datasets, these functions could be performed by special commands which would also cause appropriate information to be recorded in the data base. It was decided to test the feasibility of this idea by developing a prototype project control system, capable of providing the documentation and control desired, but within a restricted environment.

It can be argued that a sequential file of job control listings should provide a complete and therefore adequate record of the project procedures, and that this together with the preservation of all programs and datasets used should provide complete documentation. This view needs qualifying in a number of respects. Firstly, job control listings are rarely formatted for easy comprehension and usually contain a mass of detail which tends to conceal the information required. Secondly, where a number of versions of a procedure, program or dataset exist, the particular version being used is not always identified. (Where by default the latest version is used, determining which version was in fact the latest at a particular time may require considerable effort.) Thirdly and most importantly, there is no assurance that a manually maintained sequential file is in fact a complete record of all that has occurred.

Another problem with an informal documentation system is that it promotes informality of operation: deletion of files used in earlier procedures, or their replacement by later versions with the same identity.

Much has been written about computer systems for project management and control, but these are usually concerned more with planning, estimating and scheduling rather than documenting. Closer to the project control system discussed here is the file system of Toolpack/IST (Osterwell and Clemm, 1984) and the proposal of Cowell (1984) for collaborative mathematical software development projects. Cowell's software development and testing environment incorporates a project data base which contains programs, test data and a record of the status of the work.

SYSTEM REQUIREMENTS

In designing the project control system, the initial aims were to:

- uniquely identify every version of each dataset, program or procedure used;
- maintain a project library of datasets, programs

† CSIRO Division of Water and Land Resources, GPO Box 1666 Canberra ACT 2601. Manuscript received November, 1985.

and procedures;
— prevent deletion of any dataset, program or procedure used in the generation of any subsequent dataset;
— record the sequence of computational steps;
— identify the datasets input, the procedures and programs used, and the datasets generated at each step;
— generate reports, automatically or on demand as appropriate.

Also, for operational convenience, the system should:
— use as a default the latest version of any program, procedure or input dataset specified by name;
— identify as the latest version of that name any dataset, program or procedure created or entered into the system.

Unique identification of datasets, programs and procedures known to the project control system and explicitly used within it is reasonably straightforward. More complicated, if not necessarily more difficult, is identification of the programs and procedures indirectly referenced. Further, if a complete record is to be maintained to enable the computational history to be determined unambiguously, the versions of utility and application packages used and the versions of compilers referenced. Further, if a complete record is to be maintained to enable the computational history to be determined unambiguously, the versions of utility and application packages used and the versions of compilers called to generate object programs must be recorded.

THE PROTOTYPE SYSTEM

The prototype project control system which was achieved necessarily involved a trade-off between the full requirements stated above and the resources available for its implementation. The prototype system was implemented on the CSIRONET Control Data Cyber 76 computer in the batch processing environment of the Scope operating system.

The environment which the prototype system provides is defined and discussed below.
— The names of datasets, programs and job control procedures are restricted to five characters, with two digit version numbers. (This allows each to be given a unique alias - formed by concatenating name with version number - consistent with the seven characters allowed for Scope library partition names.)
— The 'core' control commands of a job are contained within a single job control procedure, the only commands outside the core being the job environment commands, declaration of project library, and initiating and terminating commands for the project control system. (This means that all job commands meaningful to the project, and not just to its control system, are contained in the one command procedure, allowing the computational steps in a job to be defined by identifying a single procedure.)
— A full trace through the tree of user procedures and programs is not attempted. Only the core procedure and direct program references from it are fully identified by version as well as name. (This restriction, which significantly compromises the design aims, was simply the result of expediency.)
— A core procedure cannot call other user procedures, but may call any number of user programs. (This restriction followed from the decision not to trace through the tree of references, and ensured that identification of the core procedure completely defined the commands of the job.)
— All control procedures, source programs and object modules are ordinarily resident within a single project library. (This simplifies the management of project resources and access to them.)
— Datasets are ordinarily resident either in the project library or on any specified set as catalogued files. (Some datasets are too large to be accommodated within the library.)
— Datasets, source programs or procedures, once created, are not modified other than by creation of a new edition, with retention of the old. (This avoids the complexity of recording update transactions, but is a severe restriction in principle which would make it infeasible to monitor a project requiring data base operations.)
— Only those datasets which are retained in the project library or as catalogued files are recognised by the system. (This was a bad decision. Although there is no necessity to keep printed or plotted datasets as files in the system – they can always be recreated if required – the lack of a record of their creation was inconvenient, and detracted from the usefulness of the control system.)
— There is no provision for deletion of datasets, programs or procedures. (This was not a problem in practice, but in a large project the inability to clean-up files of no possible further interest – as nominated by the user but verified by the system – would be wasteful of resources.)
— While there is provision for using earlier versions of datasets and procedures, there is no provision for executing earlier versions of programs. (Object modules are held in the project library under their names, rather than their aliases, and only the latest version is retained. The ability to specify the version of directly referenced programs would not have been difficult to achieve, but extending this to indirectly referenced programs would have been very complex.)

SYSTEM COMMANDS

The project control system has three groups of commands by which the user invokes the control system environment. The first group is used in jobs which perform project computations: to invoke the core-procedure and, within it, to communicate with the project control system. The second group is used to enter datasets, programs and procedures into the project control system and to manage these resources. A third group of commands generates reports from the control system.
The core procedure of a job is invoked by the command:

\[
\text{RUN (procedure name, version number)}
\]

where \textit{version number} defaults to the latest version. Three commands are available for use within the core procedure. A short description of the function of the core procedure may be passed to the project control system by:

\[
\text{EXPLAN (description)}
\]

Input datasets are defined by:

\[
\text{INDATA (logical name, dataset name, version number)}
\]

and the disposition of output datasets is defined by

\[
\text{OUTDATA (logical name, dataset name, destination, setname, description)}
\]

where \textit{version number} defaults to the latest version, \textit{destination} indicates whether the dataset is resident on the project library or as a catalogued file on set \textit{setname} and \textit{description} allows for a short, plain-language description of the dataset. (In the case of output datasets destined as catalogued files, an earlier \textsc{REQUEST} command is needed to declare the set residence.)

\[
\text{.PROC MATRP.}
\]

\[
\text{EXPLAN (REFORM MATRIX GPMTR TO ORDER GPNAMS)}
\]

\[
\text{INDATA(TAPE1,GPNAM)}
\]

\[
\text{INDATA(TAPE2,GPNMM)}
\]

\[
\text{INDATA(TAPE3,GPMTR)}
\]

\[
\text{REQUEST(TAPE4,**PF,SN=CLR2471)}
\]

\[
\text{MATRO.}
\]

\[
\text{OUTDATA(TAPE4,GPMTA,**PF,CLR2471,$GROUP DISSIM MATRIX - GPNAM ORDERS)}
\]

Figure 1. A simple core procedure

Figure 1 illustrates a simple core procedure MATRP 01, which, invoked by the command \textsc{RUN (MATRP,01)}, would attach the latest versions of GPNAM, GPNMM and GPMTR as input datasets, execute program MATRO, and save output dataset GPMTA as a catalogued file.

Entry of externally generated datasets, procedures and source programs to the project control system is accomplished by a further group of commands:

\[
\text{ENTRD (dataset name, description)}
\]

transfers the named dataset from the edit library to the project library, and identifies it as the latest version of that name.

\[
\text{ENTRP (name, type, description)}
\]

transfers the named procedure or source program from the edit library to the project library, and identifies it as the latest version of that name. \textit{Type} indicates whether a command procedure or source program is being referred to and, if a source program, identifies a particular combination of pre-processor, compiler and compiler options. After transfer to the project library, a source program is appropriately compiled and the object module replaces any previous version of that name.

\[
\text{NOTED (dataset name, owner id, setname, cycle number, description)}
\]

identifies an existing catalogued file for recording in the system as the latest version of a dataset of that name. \textsc{ARCHIV (name, version, type)} and \textsc{RESTV (name, version, type)} transfer a named dataset, program or procedure – which of them being determined by \textit{type} – respectively to or from a project archive maintained on a removable disk pack.

The third group of commands generates reports from the project control system. \textsc{DTLST} causes a list of datasets to be generated, arranged alphabetically by name and sequentially by version number. \textsc{PRLST} generates a similar list of procedures and programs. \textsc{RNHST} produces a time-sequential list of jobs run. (Clearly there is scope for numerous other forms of report, for example one which identifies the sequence of jobs and the procedures and initial datasets involved in the generation of a particular dataset. A number of such reports were planned but, in the event, never implemented.)

\textbf{SYSTEM DATA BASE}

The project control system uses a data base to record the computational history of the project and the location of datasets, procedures and programs. A CODASYL DBTG data base management system \textsc{FORDATA} (Mackenzie and Smith, 1977) was used. Figure 2 is a diagram of the data base schema.

Figure 2. Schema of project control data base

Each dataset, identified by name and version number, is represented by a \textsc{DAT} record and its location defined by membership in a set owned by a \textsc{LOC} record. A \textsc{DAT} record is created by an \textsc{ENTRD, NOTED} or \textsc{OUTDATA} command.

Each procedure or program, identified by name and version number, is represented by a \textsc{PROC} record and its location defined by membership of a set owned by a \textsc{LOC} record. A \textsc{PROC} record is created by an \textsc{ENTRP} command. Program entry-point names which differ from the program name are defined by \textsc{EPNT} records in a set
own by the PROC record. External references from a program, to program entry points known to the system and recorded in the data base, are defined by EPROC records in a set owned by the PROC record. All EPROC and EPNT records are created and linked to a set owned by the PROC record by the ENTRP command, which causes the reference table generated by the compiler during compilation of the program to be searched for entry-point names, and for external references known to the project control system.

Each batch job, identified by date and time (jobs cannot run concurrently as write access to the data base is needed) is represented by a RUN record, created by a RUN command. A single RPROC record links a RUN record to the PROC record representing the core procedure of the job. RPROCS records identify the programs explicitly called by the core procedure. RPROC and RPROCS records are created and linked by the RUN command which, prior to initiating execution of the core procedure, causes the procedure to be read to identify programs called by it.

As the core procedure is being read, it is also checked for internal consistency, for existence of all input datasets and for prior declaration of the residence of any output datasets destined to be catalogued files. Any errors discovered prevent any data base entries being made for the job, which is terminated without execution of the core procedure.

Each input dataset of a job is represented by an INDAT record linking the RUN record representing the job to the DAT record representing the dataset. An INDAT record is created and linked by an INDAT command within the core procedure of the job.

Similarly, each output dataset of a job is represented by an OUTDAT record linking a RUN record to a DAT record. An OUTDAT record is created and linked by an OUTDATA command within the core procedure of the job. The same OUTDATA command will have created the DAT record.

At the completion of execution of the core procedure, the RUN command causes a check for normal termination to be made. If the procedure terminated abnormally, the job is recorded as having aborted by setting a flag in the RUN record. In either case, a summary of the job (a single-entry RNHST list) is printed.

REPORTS
Three standard reports can be requested from the project control system.

Figure 3 shows a single entry from the list of datasets generated by the DTLST command. In the first column appear the name, version number, short description and creation time of the dataset GPDEF 01. (The list does not show whether its creation resulted from an ENTRD command or from an OUTDATA command.) The second column indicates that the dataset currently resides as a catalogued file on the project archive set. The third column identifies, by date and time, the jobs to which the dataset has been input, together with the short description of the core procedure of each job.

Figure 4 shows two entries from the procedure list produced by the PRLST command. In the first entry, the name and version number, short description and creation time of SCSTA 01 appear in the first column. The second column identifies it as a structured Fortran program, whose source code is currently resident in the archive library. The blank third column indicates that SCSTA is the sole entry-point name for the program, and the fourth column indicates that it calls user program SBEX. The final column shows that program SCSTA 01 has been used in only one job. The second entry defines SCSOR 02 as a core command procedure, currently resident in the project library. It references two user programs, SCMTR and SCSUB, and has been used in three jobs, the first two of which aborted.

One entry from a job history list, generated by command RNHST, is shown in Figure 5. The job used core procedure MATRP 01 (Figure 1), which in turn called one user program MATRO 01. The three input datasets are identified and described, as is the single output dataset GPMTA 02.

EXPERIENCE WITH THE PROJECT CONTROL SYSTEM
The prototype system was used to control one project over a period of two and a half years, until the imminent decommissioning of the Cyber 76 computer forced the project to another processor.

The project which was controlled was a stage of one in a series of research projects collectively known as the Coastal Basins Project of the CSIRO Division of Water and Land Resources (Laut et al., 1984). That stage involved the numerical classification and description of the landscape of 30 gauged stream catchments within the Hunter Valley, N.S.W., in the context of the landscape of the valley as a whole. The landscape description so developed has since been used to develop and test models to
predict the long-term seasonal statistics of stream flow from known rainfall and landform.

Although the project made considerable use of existing numerical taxonomic application programs, more than 60 programs needed to be written, and about a quarter of these went to two or more versions, after initial testing outside the control system. (Most programs were written for the structured-Fortran pre-processor Iftran.)

More than 90 jobs were recorded in the data base, nearly a third of which aborted for one reason or another. Those jobs used about 50 core procedures, about a fifth of which had two or more versions.

Approximately 90 named datasets were input to or generated within the system, half of these being small datasets containing parameters or directives. Less than a fifth of the datasets went to two or more versions.

Although the computational procedures used throughout the project generally paralleled those of the first project in the Coastal Basins series (Laut et al., 1984), progress was closely monitored by the project team, and there was considerable exploratory analysis resulting in methodological modification as the project proceeded. In such a context, it was frequently necessary to refer back to the computational history maintained by the project control system, to establish by which of the several paths tried had the current position been reached.

About eighteen months into the project and more than halfway to its goal, it was realised that errors had been made at an early stage in reforming a matrix. (The errors had not been recognised earlier because few of the matrix elements were affected.) The computational history maintained by the system enabled those datasets affected by the error to be quickly determined, and the necessary jobs rerun within a few days of discovery of the error. Without the project control system, that early error could have been a major disaster, resulting in a serious setback to the project.

Use of the project control system involved significant overheads in computing costs. Typically, an ENTRP command for a source program incurred an overhead of five CPU seconds, while a RUN command to execute a core procedure having several input and output datasets incurred a similar overhead. The overheads for other routinely used commands were modest by comparison. The additional costs, though significant, were acceptable in the light of the total resources devoted to the project.

CONCLUSION

Although many compromises were made in the implementation of the prototype project control system described, it proved to be an extremely valuable tool in controlling and documenting progress in a research project having substantial computational requirements. Use of the prototype system ceased with the decommissioning of the processor on which it was implemented and, while some consideration has been given to designing and implementing an improved version, no decision has been made.

Although a project control system is implemented in the environment of a particular operating system, a common user environment could be provided for project control systems implemented on different processors or...
under different operating systems (c.f. Hall et al., 1980). One of the advantages of such a uniform project control environment is the possibility of passing information between project control systems to allow control of a project using a number of processors.

REFERENCES

 BIOGRAPHICAL NOTE
Bruce Cook graduated B.Sc in mathematics and physics from the University of Sydney in 1953. He began computing on Silliac in the early 1960s while a geophysicist with the Bureau of Mineral Resources. In 1967 he joined CSIRO, where much of his work has concerned geographic information systems.
A Prolog Environment System

J.D. Newmarch†

A single user system has been implemented to manage the task of writing Prolog clauses and programs. By storing definitions individually and using knowledge about Prolog, an environment is created which reduces errors and code duplications.

Keywords and Phrases: Prolog, logic programming, programming environments.

CR Categories: D.2.6, I.2.5.

INTRODUCTION

Prolog (Clocksin and Mellish, 1984) has become popular recently not only due to the Japanese Fifth Generation Project but also because of its own merits. At present Prolog is the most viable implementation of the predicate logic based languages and offers substantial advantages and disadvantages caused by this logic base. The advantages are that within certain limits programs can be written in which the definition is the program, thus guaranteeing correctness. Some disadvantages are due to the introduction of non-logical constructs such as I/O mechanisms and negation as failure. Others are specific to Prolog wherein the execution method is known and abused. A further class of disadvantages is inherent to logic programming itself.

Primarily missing from logic are the various checks and assumptions that we have become used to in strongly-typed languages such as Pascal. For example:

i) There is no type checking for there are no types (although Prolog interpreters use typing for internal reasons).

ii) There is no notion of checking possible use against definition. In procedural languages, the use of a procedure means the procedure must be defined, for otherwise the program has no meaning. In logic, programs always have a semantic meaning whether or not procedures (clauses) are defined. They also have a clear execution method in that when an 'undefined' procedure is encountered it merely fails in a non-catastrophic manner. These strictly invalidate use versus definition checks, although some Prolog interpreters make the assumption that such checks are useful.

iii) For the same reasons, no check on the number of arguments used versus those defined has validity in logic.

These create a sometimes unpleasant environment for the logic programmer. The author has spent many hours tracking down misspelt names, omitted definitions, superfluous definitions, incorrect argument counts and so on.

This paper describes a Prolog Environment System (PES) that is essentially a file system for Prolog. This file system performs library management functions as well as performing 'compile' time tests which have proven useful. Further details are given by Newmarch (1986).

BASE COMPONENTS OF PROLOG

In creating aids to programming in any language one must of course pay attention to the natural units of the language. The minimal (and most talked of) unit for Prolog is the clause

X if Y and Z and ....

e.g.

(List [A,B,C,...] is ordered) if

(A > B) and

(list [B,C,...] is ordered)

Generally there are alternative definitions of a clause head

X if Y

X if Y'

...........

corresponding to

X if Y or Y' or ....

The set of alternatives for a clause head are termed a relation or procedure, and from a user point of view form a complete definition of a clause head

e.g.

'ordered' is completed by

(list [A] is ordered) if true

(list [J] is ordered) if true

A theory is a set (possibly empty) of relations. A program is a theory together with a goal, which is a statement of the form

X and Y and ....
A Prolog Environment System

e.g.

\[(\text{list } [A,B,C] \text{ is a perm of list } [1,3,2]) \]
and

\[(\text{list } [A,B,C] \text{ is ordered})\]

Execution of a program is an attempt to prove the goal from the theory. In the above example, if 'perm' and 'ordered' are in the theory, the goal is proven with \([A,B,C] = [1,2,3]\) i.e. the program sorts the list \([1,3,2]\).

CURRENT ENVIRONMENT

The current general Prolog environment is that of creating one or more text-files to represent a program, with these text-files all loaded by the Prolog interpreter. Certain commonly used clauses (such as list processing clauses) are sometimes stored on a separate file, and it is up to the programmer to remember to load this file. More frequently, 'useful' clauses are buried in some text-file and it is easier to create another instance.

Frequently, clause dependencies are scattered in a complex form among files and no aid is offered in ensuring that all and only those required are loaded.

The file-based 'private' mechanism of many Prolog interpreters may be used to implement 'modules' and avoid name clashes across files. To use this rigorously as a module mechanism would produce a large number of files without a management mechanism.

LOGICAL VIEW OF THE SYSTEM

The system (PES) described here is fairly low-level: whilst it does not at present include tools such as a syntax-directed editor, it facilitates the writing of such tools.

The core of this system lies in the management of the Prolog units. Each relation is kept on secondary storage as one logical file with pointers to relations which use and are used by it (see Figure 1). This implicitly builds a graph structure of relation dependencies. Any new definitions are linked into this graph, allowing immediate checks for existence, argument counts etc.

A program is created by making a flat-file containing a theory for a given goal. This is done by recursively tracking down the 'uses' pointers. Checks ensure that a relation is added only once, and warnings are issued if a relation is missing.

The primary advantage of this is that relations are defined no more than once. For example 'differences between two lists' is currently used by four separate relations of PES, but is defined only once, is present only once in compiled PES, and is available to any other program created using PES.

A major motivation in building PES concerned different functional requirements of programs. A program to run on a Tektronix graphics terminal requires different output primitives from one for an ANSI terminal. For example, both can move to X but the mechanisms differ. A general 'indifferent' clause may just require the cursor to be moved. A particular program must decide which one to use, as it cannot use both. This is catered for by:

i) Two or more logical relation files may define the same head. The fact that they are stored separately implies a distinct functional meaning.

ii) A particular program with an ambiguous functional purpose chooses which definition to use by explicit reference. This reference is made by pointing to a group of functionally close definitions (e.g. 'ansi group') which in turn points to all group members.

This mechanism has been used to write the core of PES in a relatively independent way: MU PES points to the MU group and UNSW PES to the UNSW group. These in turn resolve syntactic and (some) semantic differences by pointing to suitable definitions (see Figure 2) to give programs for the MU Prolog (Naish, 1983) and UNSW Prolog (Sammut and Sammut, 1983) interpreters respectively.

---

<table>
<thead>
<tr>
<th>User description of the relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword descriptors</td>
</tr>
<tr>
<td>Argument description</td>
</tr>
<tr>
<td>Relation definition</td>
</tr>
<tr>
<td>Pointers to relations used by this one</td>
</tr>
<tr>
<td>Pointers to relations using this one</td>
</tr>
</tbody>
</table>

Figure 1: Logical structure of a relation file.

Figure 2: Resolution of multiple relation definitions

DOCUMENTATION
There are no standards for Prolog documentation, let alone accepted ones. Built into this system is a 'default' documentation consisting of:

i) A textual description of a definition.
ii) A textual description of the arguments used.
iii) Keywords which may be used to find the name of a definition.
iv) A list of atoms used by the definition. This is maintained by the system.
v) A list of clauses which use the definition. This is maintained by the system.

For (i)-(iii) the user is given an editor environment for each on creating a definition. A user can search for a relation name on the key, or print all fields.

PHYSICAL IMPLEMENTATION
The PES system has been implemented on a Sun workstation with 1Mb of main memory (2Mb of virtual memory) running under the Berkeley UNIX 4.2 system. This version of UNIX allows file names of up to 256 characters, and an effectively unlimited number of files per directory.

Each definition is stored as three files:

i) A documentation file.
ii) A source file.
iii) A file containing other information such as
    • argument description
    • keywords
    • clauses this definition uses
    • definitions which use this one.

File names for clauses are generated as 'Clause.src.n' which often exceed the fourteen character file name limit of standard UNIX.

This particular implementation may not be portable to many other Operating Systems due to restrictions imposed on characteristics such as file-name length and number of files/directory. These may be avoided by building a file archiver, if the Prolog interpreter supports random file access or if the interpreter directly supports storage of clauses on secondary media.

CONCLUSION
The current version offers facilities to edit and print relations, functional groups and programs, and to 'compile' and run programs under both the UNSW and MU Prolog interpreters. Included is a keyword mechanism to track down names and related definitions.

The system performs its library role in a transparent fashion. The user may use any relation already defined without having to track down dependencies.

Program development can occur top-down or bottom-up without problem. The keyword retrieval and print mechanisms allow the user to build on what is already there.

Some problems are caused by the current implementation on top of UNIX. These would be expected to disappear if an archiver is used.

Many enhancements are possible and are planned. In the meantime PES has proved to be a much more congenial environment than the standard one.

REFERENCES

BIOGRAPHICAL NOTE
Jan Newmarch has a B.Sc. in Mathematics from the University of Bristol and a Ph.D. in Chemistry from the University of New South Wales. He has tutored Mathematics and later Computer Science at the University of New South Wales and has been a lecturer in Mathematics at the Papua New Guinea University of Technology and a lecturer in Physics at the University of Agriculture, Malaysia. Currently, he is a lecturer in Computing Studies at the Canberra College of Advanced Education, teaching Artificial Intelligence and Expert Systems.
Programming Environments – Deskilling or Job-Enriching?

C. Sauer †

The argument that developments in programming languages, methodology and work organisation have routinized and deskilled programming is outlined. The extent to which this argument may be applied to programming environments is examined. A case is established, but is then criticised through the recognition of ways in which they may enrich the programmer’s work. It is concluded that the technology when applied in the context of routine software production does indeed have characteristics which may facilitate further the deskilling process.

Keywords and Phrases: software engineering, programming environments, computers and society, the computing profession.
CR Categories: D2.6, K.1, K.4.

INTRODUCTION

The programming technologies of the 1960s and 1970s have exhibited features which have tended to facilitate the deskilling of programmers while at the same time requiring the acquisition of new skills. Thus a tension has been apparent in the effects of programming technology upon programmers. The diffusion of integrated programming environment technology from the academic and military/industrial R. & D. laboratories to the wider industrial, commercial and business communities is likely to generate similar outcomes. While individual additions to the programmer’s tool kit may enrich the job, some particular features of programming environments and their very existence as integrated environments will offer opportunities for the further deskilling of programming.

Our purpose is to examine in some detail how programming environments may generate such a tension. There is an existing argument (Kraft, 1977 and 1979, Greenbaum, 1979), that changes in programming technology, in methodology, in computing education, and in the organization of programming work, have tended to deskill the average programmer. We develop and extend this argument through an examination of integrated programming environments. It is not part of our purpose to discuss the validity of the deskilling argument in general (although it will not be left totally unscathed).

The discussion is worthwhile for three reasons. First, the effect of programming environments on work has received little or no attention so far. Second, there has been the development of the ADA language and the specification of an ADA Programmer Support Environment (APSE). The importance of this lies in the influence of the DoD and the US military establishment upon the directions to be taken by the computing industry. (For a discussion of this influence in the early days of computing see Noble, 1984.) Whatever the eventual success of ADA it is clear that its key concepts, including the APSE, will have a significant effect on the software industry, to the extent that integrated tool kits will increasingly become a part of the software development process.

The third reason is that programming environment technology addresses a gap in the production cycle. It is a technology that concerns how programmers actually carry out the task of building software, what tools they use, how they correct mistakes and how they document what they have done. Thus it is precisely the technology that has the most direct effect on programmers’ skills. Given the aims of the paper and now its justification, we are set fair to proceed with the question - do programming environments deskil or enrich the occupation of programming?

Initially we shall clarify as best we can the characteristics of programming environments most relevant for our discussion. We shall then list some assumptions that are work making explicit, and touch upon methodological questions. Some considerable space will be devoted to the existing arguments about the deskilling of programming, so that its subsequent extension to programming environments is comprehensible. Finally we shall offer a critique of the argument so far, which acts as a pointer to some further research directions.

PROGRAMMING ENVIRONMENTS

No detailed discussion of programming environments is given here as Marlin’s review article (this issue) provides an appropriate survey. Nonetheless certain points must be made. First, no paradigm for a programming environment is assumed. No particular tool or facility is necessary to our discussion. However, a largely interactive mode of working is assumed. Editing, documenting and debugging are likely to be done on-line. Program execution might or might not be in batch mode. Obviously if code is
interactively interpreted and executed this will affect programming practice substantially (see e.g. Teitelman, 1984, pp 241-3), but a complete break with the edit-compile-execute cycle is not essential to all aspects of the argument.

Second, it is not assumed that a programming environment is necessarily fully integrated as in Teitelman's Display-Oriented Programmer's Assistant (Teitelman, 1984) where for example text can be moved straight from a mail message into a program structure and the new program executed with the minimum of programmer activity. Instead, it is enough for our argument to see a programming environment as a collection of software tools available to the programmer during a terminal session. However, the extent to which they constitute an integrated programming environment is important and is discussed later in this paper.

A third point to note is that programming environments often embed the designers' assumptions about programming methodology and the right way to build programs, for example:

The integrated behaviour of templates and the cursor enforces the proper view that a program is a hierarchy of structurally nested components.

(Teitelbaum and Reps, 1981, p. 569)

The critical aspect is that such assumptions are embedded, but not which methodology is assumed.

ASSUMPTIONS

We make the following three assumptions:

1. Technology is not neutral in its effects upon society. Supposedly many technologists believe technology to be neutral and that it is the people who apply it who choose to apply it to good or bad ends. Whatever the reader's views on this, it is assumed here that technologists in fact do not work in a social and economic vacuum, that their innovations are at least in part influenced by the requirements and values of the patrons of research and by the likely end user. This applies as much to computer scientists developing programming environments as to nuclear physicists building bombs.

2. The argument that programming has been deskilled and is still in the process of being deskilled has sufficient truth in it to make a discussion of the effects of programming environments worthwhile. In due course the reader will have an opportunity to assess the plausibility of this assumption.

3. Considerable further development of integrated programming environments will take place. They will spread into the software development workplace and will not remain the province of highly skilled programming technicians working on novel projects at the forefront of technology. Rather they will become a conventional part of every programmer's work even in mundane COBOL shops where maintenance forms a large part of the job.

Programming environments will affect the majority rather than the minority.

METHODOLOGICAL REMARKS

Our approach is limited to an examination of the characteristics of the technology rather than documenting the effects of programming environments on the workplace. It seems inappropriate to survey practitioners in view of the current shortage of integrated programming environments in use in everyday software development. While clearly most programmers these days have access to display editing, file systems and debugging tools, we are still in a transitional period between the old technology of batch programs using cards and the new technology of highly integrated sets of software tools where programs can be built and maintained interactively.

The focus on the technology is a function of Assumption 1, that technology is not neutral and Assumption 2 that existing points made about programming languages and methodology are valid. Indeed, the emphasis on the technology is a virtue in that it allows us to hypothesise about the likely success and uptake of various types of programming environment by the computing industry as a whole. We are not as yet ensnared by the complexities of the social relations of the workplace, and thus we can make a first assessment of the technology of integrated programming environments without many of the difficulties inherent in social observation.

THE DESKILLING OF PROGRAMMING

Following Braverman (1974) there has been discussion in industrial sociology and in the labour process debate as to the extent to which industrial and clerical work has been deskilled (e.g. Cooley, 1981). Braverman saw all management theory being based in principles of Taylorite Scientific Management. In particular he drew out three principles (Braverman, 1974, pp. 112-121):

1. The arrogation of craft knowledge by management. The essence of this principle is that management should become the repository of technical knowledge and craft skills rather than the craftsman.

2. The separation of conception from execution. Characteristically the craftsman makes his own decisions. He plans his own work, decides upon materials, tools and techniques, and then carries out the job. By contrast the average production line worker performs the same group of actions repetitively all day according to management's decisions.

3. The standardization and formalization of methods. Management aims to control craft knowledge according to Principle 1. Principle 2 separates out the creativity, the planning, indeed the thinking. It then applies this control by standardizing the methods of performing a given task, and by specifying it in such a way that there is a given approval...
process for any worker to follow. Ideally it is a simple algorithm that any fool (or automaton!) could apply correctly. Thus, for management the aim is seen to be the elimination or reduction of workers' skills, so that it can control as far as possible the whole labour process.

Braverman recognized that similar changes were affecting clerical as well as industrial work. At first glance the clerk and programmer appear very different occupational types, so we must ask what grounds there are for supposing that programming has been deskilled.

The case is outlined in some detail by Kraft (1977 and 1979) and Greenbaum (1979). Early programmers were 'computer experts' who used hardware and software as a tool and a medium for solving problems. Their public image was of all-solving technocrats. Ambitious projects bred uncontrollable complexity. It became apparent that the 'experts' were tardy in delivering the goods (on the occasions when they could deliver at all!). It would be unwise, unsophisticated and historically inaccurate to suggest that as a result management stepped in to apply the principles of Scientific Management. It was certainly more complicated. What is clear is that from the 1960s on there was a recognition among both computing personnel and business managers that the software crisis required action. Many of the changes that occurred were not independent but interlocking. Thus, for example, developments in programming languages, particularly the trend to higher level languages, made the programmer's work more comprehensible to managers and thus made possible the growing interest in project management, in resourcing and scheduling and subsequently in development methodologies. Block structured languages like Pascal were developed to support a top-down, modularised development methodology. The two went hand in hand. The result for programming was that management did indeed increase its knowledge of the technology and hence its control. Division of labour was applied through the creation of project teams along the lines of chief programmer teams. Systems analysts became the interface between the programming process and the organizational problem, the application. Programming became a collection of job design, coding, testing, implementation and maintenance. These could each be allocated to separate people, so that some would be the production workers of the software shop, the coders, while others would form implementation teams, and still others would be consigned to the contemporary task of Sisyphus, program maintenance. Thus, conception, creativity and planning were largely separated from the routine aspects of construction of software. In addition, management started to specify the technology, or at least to require that the use of specified technologies be justified by the specialists. Installation standards covering most facets of software development became commonplace. They were commissioned by and remained the province of management. (I myself worked in a department where standards running to several looseleaf volumes were developed in-house. They were mostly written by project managers, or by senior systems analysts and senior programmers who were either managers or closely allied in their work to management. Interestingly the looseleaf volumes were distributed on a highly selective basis according to the work an individual performed. Programmers did not receive the analysis and design volumes, nor the operations volume. Analysts were likewise deprived of certain volumes. Only project managers upwards received complete sets and thus, if only symbolically, were repositories of all knowledge.) Thus, through securing a hold on programming knowledge, dividing the tasks into smaller jobs, separating conception from execution, and by insisting on standardized working practices management engaged in the process of deskilling programming.

This was not all. The technology itself facilitated these changes. Both developments in languages and in methodology contributed. High level languages not only made programming more comprehensible to the layman, but removed areas of expertise; likewise developments in operating systems. I/O, for example, became the province of systems programmers, not applications programmers. And fourth generation languages take the process further, a process which Weinberg (1971) characterized:

Perhaps because of these high salaries, or because they cannot tolerate being at the mercy of a mystique they don't understand, computer executives have always been aware of the human element in programming. Their concern, however, has usually been with eliminating, rather than understanding, the human element.

Systems development methodologies and the disciplines of software engineering have made further inroads into programmers' freedom of action. They have brought prescribed ways of designing programs, specifying programs, common libraries of procedures, documentation standards and so forth. More discipline and less freedom. This leads us then to the current flurry of activity in the area of programming environments, but before we examine this technology it would be a good idea to qualify the argument as explained thus far in order to disarm some of the reader's likely disquiet and battery of objections.

First, not every person who programs, nor every person whose job is to program has been deskilled. Kraft (1977, p. 63) recognizes this, '... the most skilled software specialists are today without doubt more skilled than their predecessors; the deskilling trend refers to the average level of programmer skill'. Programmers who work in what are often termed COBOL shops are most likely to have been deskilled in some fashion. The label 'deskilled' is itself not uncontroversial. It is an emotive term which serves to denote changes that have reduced the freedom of programmers to control their own working lives, to exercise creativity and to use the full range of techniques potentially available to them. It should not be taken to imply that programmer skill levels have been reduced to
zero, nor that the average programmer is no more than a production line drudge. Rather, skill levels have been reduced and as a concomitant a once highly prized occupation, 'computer expert', is now less valued and has less status. Nonetheless, it is assumed (Assumption 2 above) that many readers who have programmed through the 1970s and 1980s will recognize a nugget of truth in this account of the changes that have taken place in programming.

Two further points need attention. It may be claimed by management apologists that all such changes are justified in the name of efficiency, organizational effectiveness or some other goal of economic rationalism. This interesting argument misses the point. The earlier analysis of how programming has changed as an occupation was not an accusation, so a justification is irrelevant.

The final point is that programmers have gained new skills to replace their old ones. They have exchanged ASSEMBLER for COBOL. Certainly programmers have acquired new skills, but whether they compensate for lost skills is not obvious. It has already been suggested that programmers lost a whole area of expertise, I/O management, with third generation software. Has there been adequate compensation? It is doubtful, given the changes to the organization of programming work as well. On the other hand, it might be argued that programmers are always learning something new and that as far as the technology is concerned it gives back more than it takes in terms of skill. Programmers now are able to tackle, with some real chance of success, problems they would not have expected to handle with earlier technologies.

PROGRAMMING ENVIRONMENTS

The aim of this section is to assess the degree to which arguments similar to those already outlined can be extended to programming environments. In the next section we shall criticise these arguments, but for the moment we shall concentrate on the apparently negative aspects of programming environments.

First, how do automated tools affect the programmer who uses them? Are they analogous to the introduction of machine tools into factories? As the task has become automated the workers have tended to become machine minders where responsibility is for the care and maintenance of the machine and the oversight of the actual task, rather than for carrying it out. As yet there is little sign of the programmer's work being automated to the extent that he simply feeds in key parameters to an intelligent programming environment and watches the results, which would be the sort of radical change required to make the analogy hold. Nonetheless, the programmer is affected by the tools available, if not as radically as the skilled craftsman turned machine minder. For one thing, the available tools clearly do affect and often constrain our thoughts. Maslow's famous line to the effect that to a person who has only a hammer, the whole world looks like a nail catches this insight. The point in relation to linguistic tools is the heart of the Whorfian hypothesis (Whorf, 1956). We ought not to be surprised if we were to find that the programmer was largely affected by what tools were available. Thus a syntax directed editor is most likely to result in a programmer seeing his program as a syntactic structure rather than as a piece of straight text, and clearly this is the intention of some designers (e.g. Teitelbaum and Reps, 1981). If a programmer has a choice as to which tool to use to solve a problem then that is the decision proper to a skilled technician. However, two points need to be drawn from this.

First, technology is employed in social contexts where the programmer does not always have total control over his work. He may have certain decisions, about languages, methods and tools imposed upon him; and while that is not the fault, so to speak, of the technology in question, if it happens to constrain the programmer's use of his skills, the effect is that of deskilling. More importantly though is the effect that tools can have, not as a single tool to be chosen or not, but as part of an integrated set that embeds one specific methodology.

It is clear that the purpose of many programming environments is indeed to provide an automated version of a specific methodology or as support to some approved style of programming. Thus, Barstow and Shrobe (1984) point out:

In contrast, modern environments (to one extent or another) are constructed with a style of work in mind, taking advantage of what are believed to be naturally occurring structures of the software development process.

If they are correct in this, and examples such as MENTOR (Donzeau-Gouge, Huet, Kahn and Lang, 1984) and the Cornell Program Synthesizer (Teitelbaum and Reps, 1981) support their view, then it is clear that the adoption of such programming environments will constrain and reduce the programmer's scope for skilled decision making in the problem solving process. Perhaps the most graphic example where the technology is being built quite deliberately to reduce the range of programmer choice and to enforce conformity is the APSE (Stoneman, 1980). The whole ADA project has been motivated by the desire of the US DoD to achieve uniformity of software development through a single language and a single environment. In effect, if programming environments are integrated in such a way as to embed specific methodologies then the arguments of the previous section apply. It is also clear that the imposition of 'correct' methodologies upon students (as with Mentor and the Cornell Program Synthesizer) is likely to carry through from college to workplace thereby reinforcing the deskilling trend.

Besides the effect of embedded methodology we need to consider the influence and importance of individual software tools or types of tools. Three in particular appear to be possible deskilling agents: syntax directed editors, intelligent tools, and project monitoring and reporting software. Syntax directed editors, insofar as they reduce...
the programmer's need to be able to apply a syntax, clearly reduce skills. Templates, as in the Cornell Program Synthesizer, where reserved words cannot be edited, reduce coding to form filling, which perhaps does not matter to a program designer, but to somebody whose task is chiefly coding it reduces the skills and hence no doubt the status and marketability of the job that then becomes open to less skilled workers. The semicolon in Pascal may be an irritant to computer scientists, but when knowing its correct place and the effects of its absence are central to your job, the elimination or reduction of the need for such knowledge is a direct attack on your skills. Syntax directed editors will inevitably reduce the value of fluency in a particular language.

A discussion of intelligent tools might well be divided according to existing tools and future developments. Perhaps the most discussed example of a smart tool is Teitelman's (1972) DWIM (Do What I Mean) facility for Interlisp which is able to make appropriate deductions as to what needs to be done to correct an error be it a syntax error or spelling mistake. Thus, obvious bugs can be removed from programs with no programmer intervention other than perhaps agreeing to the proposed change. This appears to make life extraordinarily easy for the programmer. No more debugging. Of course, it is not that intelligent. And thoughtless use could exacerbate rather than solve problems if incorrect inferences are made by DWIM and the changes agreed to carelessly. So, it might be argued that DWIM, for example, is an aid which requires intelligence and skill to use effectively. Nevertheless, such tools do exhibit a tendency to deskilling even if there are mitigating factors.

Predictions as to the future of intelligent programming environments vary, the total elimination of the average programmer being one possible scenario. Barstow and Shrobe (1984, pp. 566-567) offer a vision with a different emphasis. They see intelligent programming environments facilitating the user's input to the design process. From this one might speculate that the elimination of the high-level systems designer or systems analyst will be the outcome rather than the disappearance of the programmer.

A further feature of many programming environments is project monitoring, configuration management and associated tools. While on the one hand automatic version control may relieve the programmer or project librarian of a tedious chore, the programmer is constrained to work according to some prescribed discipline for the version control system to function effectively. More importantly, the ability of management to monitor direct from the environment actual work done, rather than what the programmer records on a project reporting sheet, places the programmer more directly under the control of management. In itself, this is not deskilling, rather it improves management's capacity to ensure that prescribed methods are followed. So such software may not deskill programmers directly, but rather may contribute to their effective deskilling.

**JOB ENRICHMENT**

The case that programming environments are job enriching ranges from the provision of more versatile tool kits to the possible reuniﬁcatory effect on the division of programming labour. Of course, the actual effects of the technology will depend considerably upon the continuities of the social context in which it is implemented.

The first point to make is that programming environments offer programmers a diverse range of software tools from extensible editors to team communication systems to integrated documentation. Each such facility requires its own skills to use it effectively, so programming environments offer programmers the chance to increase the range and diversity of their skills. Secondly, the range of tools opens up the possibility of programmers exercising greater freedom of decision in choosing the appropriate way to develop a piece of software. Thus, for example, the UNIX shell is often used to build programs from its existing functions rather than writing a compiled program (see e.g. Kernighan and Mashey, 1984). Thirdly, the existence of new software tools may, and indeed should, make life easier for the programmer.

Programming environments remove the need to attend to mind numbing detail, particularly where that detail is irrelevant to the semantics of the problem at hand. To that extent they should prove beneﬁcial to the programmer. The qualification to this is that much depends upon the actual organizational context. They will only be beneﬁcial to programmers if they release programmers for more rewarding work. But, even if programmers in fact do not win more desirable work, it may be that insofar as programming environments increase the likelihood that systems, when completed, function correctly and robustly they will offer an increase in job satisfaction. In addition they may help reduce the maintenance burden by reducing the need for it and by making it an easier job when it is required.

The extent to which programming is made easier and is rid of tedium will be in part a function of the individual tools. Thus, while some tools appear to be predisposed to cause deskilling, others quite clearly are built to enrich jobs. EMACS (Stallman, 1984) seems only to offer the user choice and freedom to customize his own editing environment. Likewise, the increasingly common feature of recording the history of a terminal session, thereby facilitating the reaplication of an edited session or subset of a session, transparently aims to free the programmer from boring repetition, be it as a result of his own error or because several objects require similar treatment. A further example is of increased ease in switching activities through the use of divided screens. So, individual tools can indeed help enrich programming.

More speculatively, we might argue that the occupation of programming is itself a dynamic one. There is no standard set of skills that characterises the average, effective programmer. If anything the average programmer is, and has had to be, an adapter. The technology has never stood still. It is always required of
programmers that they be able to and be prepared to learn new skills. So it could be said that the chief characteristic of the programmer is the meta-skill of being an effective learner. Hence the development of programming environments is simply a further opportunity for the programmer to exercise his meta-skills.

Looking to the future it is possible that by appropriate contributions to the application and implementation of methodology, programming environments might contribute to a trend to reunite the tasks of design, coding and testing. As code production is integrated more with the design process it may become simpler to attempt to unify these tasks. In practice the decision as to whether to organize work thus is likely to be made according to the economic and political exigencies of the organization in question as much as upon the merits of the technology.

CONCLUSION

The aim of this article has been to highlight some features of developments in integrated programming environments in order to consider their likely effects upon the occupation of programming as it exists in most commercially oriented software production contexts, particularly in what would ordinarily be called data processing. Its emphasis on deskilling aspects is largely irrelevant to highly skilled, autonomous research and development programmers. As sophisticated tools and integrated environments become more commonplace, the actual effects to be observed will be determined by the wider interests of organizations and their management. The extent of their adoption will be affected by the extent to which the technology can be presented to programmers as job enriching while at the same time serving the objectives of management.

We have concentrated upon the effects of programming environments on programmers. This is in contrast to the conventional wisdom that assesses a technology almost exclusively according to its instrumental value, according to its contribution to organizational effectiveness. The idea of egoless programming (Weinberg, 1971) has implicitly or explicitly encouraged this abstraction from the humanity of the programmer. Nonetheless, given the legitimacy of the programmer's interests it has become clear that the technology can be presented to programmers as job enriching while at the same time serving the objectives of management.

The extent of their adoption will be affected by the extent to which the technology can be presented to programmers as job enriching while at the same time serving the objectives of management. Much depends upon how far individual programmers or work groups are free to exercise choice. After all, it is one thing to impose a discipline upon oneself, quite another to have it imposed.

REFERENCES


BIOGRAPHICAL NOTE

Chris Sauer is a lecturer at Griffith University, Brisbane where he has been involved in the design and teaching of a new computing degree, the Bachelor of Informatics. His first academic post was as lecturer in Computing at the Open University, prior to which he had worked as a systems analyst for seven years. His interests are in the relationship between philosophy and computer science, particularly the ethical implications of new technology; the development and implementation of large computer systems especially those resulting in failures; and the sociology of the occupation of programming.
Letter to the Editor

Comments on the Influence of Academics
It has been remarked that the Australian Computer Society is not unduly influenced by academics as they number less than 2% of the membership. I would like to suggest that their influence is out of proportion to their numbers.

Academics are seen as more neutral in a community that can be broadly divided into suppliers and users. Users tend to be 'vertical' specialist people with a narrower scope of work, usually limited to a particular application area, whereas suppliers tend to be 'horizontal' generalist people who (with luck) see a broader range of products, tools and applications. Academics, although users, have the broader outlook (we hope) without having to push a particular product. They can keep in touch with the latest developments without the need for loyalty to a particular brand.

The vast majority of journal articles are contributed by academics. These articles often arise from their particular line of work, and they are accustomed to publishing papers. Non-academics, if they were to publish any reports, are often restrained by proprietary concerns and the need to respect confidentiality.

Perhaps the disproportionate influence of academics is not altogether unwelcome. If they keep in touch with the latest developments, then they serve a role in bringing these to our attention. Problems may arise when academics forget that the rest of the computing world may not speak their language, and does not always understand their articles published in the journals. If an article is supposed to help one develop professionally, it would help if one could understand it.

To broaden the content of our journals, contributed articles could include reports on projects in the commercial or public sector. Suppliers could also contribute presentations on e.g. some AI language or an automated system design tool. This could be done without pushing 'X' brand computers. The August, 1985 issue on Office Automation has some good examples, and is much more readable than most other issues.

One the subject of membership requirements, could we have, in addition to academic qualifications and entrance examinations, some method for entry for experienced personnel who find it difficult to sit and regurgitate answers on an exam script. The ACS (like the British Computer Society) could allow the submission of reports on specific projects undertaken or some kind of entrance interview, where the applicant could present his or her work experience.

Finally, if we talk about being a professional body, how are we to be professional? A member of the BCS has raised the question of the Computer Society supporting a member when he or she is required to do something that is professionally unsound. Will the ACS be willing to disqualify a member who breaches the ACS code of conduct?

I am not sure that any single body can represent all data processing professionals, as we have such a diverse and sometimes conflicting range of interests. In other countries, this has resulted in the creation of a number of different organizations to represent the computer industry.

John Yeo
Glen Waverley, Victoria 3150

Editor's Reply
Many of Mr Yeo's points have been made before, and many need making again. It is widely understood that academics are expected to conduct original investigations and report their results in the open literature, so it is not surprising when they do so. Where the work reported is new and original, many of the ideas and notations may be unfamiliar and appear difficult. This is not always entirely the fault of the authors.

It is also widely understood that professional people need to develop and maintain their skills in a variety of ways, including reading journal articles. This Journal covers a very wide range of topics: as editor, I am satisfied if most readers can find an average of one article in each issue that interests and informs them. I do not know of anybody who digests every article in every issue—I certainly do not! As editor, I rely heavily on expert help from referees in deciding when to publish or not. However I do try to insist that, for each article, for the first page at least, the author should cater for the curious as well as the experts, so that all readers should be able to understand the problem addressed and the general direction taken for its solution.

Not so widely understood is the duty of all (true) members of a profession to share their experiences and insights as widely as possible including via the written word. Writing serious articles for the permanent record is an exacting task for which skills need to be developed and practised. Not all initial attempts will be successful. As editor, I would be delighted to see a hundred-fold increase in the number of articles contributed by professional, non-academic members of the Australian Computer Society. I am still waiting to be amazed.

John Lions
Editor

Book Reviews


This book does not succeed as a history of computing. The author is a silicon valley journalist whose writings seldom rise above the journalistic level. The material is insular and parochial with a very American perspective.

I have ceased to hope that a book on the history of computing will refer to Trevor Pearcey or the CSIRAC, though the latter was apparently the fourth working computer in the world and the first in the Turing/NPL tradition as opposed to the Von Neumann/LAS tradition. Augarten's book did not disillusion me, therefore, in ignoring all antipodean contributions to computer hardware and software.

Rather harder to digest is the dismissal of the EDSAC at Cambridge, the first stored program computer in production use, in one scant paragraph devoted mainly to the mercury delay line storage. There is no mention at all of Maurice Wilkes, nor of the revolutionary programming environments, the first assembler programs, that contributed so greatly to EDSAC's success. Manchester fares a little better with two photos and a page of text.

When dealing with material whose source is nearer to home, as in the last two chapters on 'The Integrated Circuit' and 'The Personal Computer', the book is very much better. I found the former chapter fascinating, though the latter indulged too much gee-whiz hero worshiping.

As an 'illustrated' history the book is somewhat better. It does contain some good pictorial material although the format of the book does not do this material justice. It is not a coffee table book, and must live in the book shelves.

In many ways this is a disappointing book, especially when one considers what it might have been. But if you don't take the history too seriously, it will give a fair measure of pleasure and enjoyment for the price.

Allen G. Bromley
University of Sydney


As its preface states (twice), this is an 'introductory handbook for people who want to find out how expert systems work' and it aims to bring computer users who are unfamiliar with developments in Expert Systems to the point of being able to start their own knowledge engineering projects. It is one of the most readable of the books that introduce Expert Systems at the level of computing practitioners.

It is in four sections, entitled Background, Inference, Knowledge Engineering and Learning. The first two chapters give a clear and concise introduction. Although I liked the other two chapters of the Background section less, Chapter Three did give a fair description of the Fifth Generation. The second section, on Inference, includes a good treatment of the theory and of forward and backward chaining mechanisms. The section on Knowledge Engineering actually describes three systems and never addresses the issue of taking an expert and encoding his or her knowledge. The final section, on Learning, addresses the problem of the 'knowledge bottleneck', getting knowledge into an expert system. It starts with a good overview of the problems and some of the major approaches, but from there, it is strongly oriented towards one particular approach, that of genetic adaptive algorithms. For a practitioner's book, one might have expected more detail on systems like ID3 that are commercially available.

The thirteen chapters of this book have been produced by ten authors. One advantage of this is that it does offer different perspectives on the subject. For example, Forsyth does not mince words when discussing Prolog as a 'much-touted logic programming language' of which the kindest thing he can say 'is that it is ahead of its time.' Then follows a list of objections to Prolog. To balance this, however, there is a chapter on Prolog in the section on Knowledge Engineering and this includes a toy example that illustrates how Prolog works when used to encode rules.

Several of the chapters, especially those written by Forsyth, the editor, are very pleasant reading. Many of the authors are practitioners, describing their own systems. As one might expect, there is some unevenness in quality and the reader is sometimes left with difficult links to forge between chapters. For example, both Chapters Six and Eight deal with inference engines without comments to compare the two. On the other hand, once you have read the two introductory chapters, you can read any of the other chapters in any order.

For a book that is aimed at practitioners, it is unfortunate that some of the references are relatively inaccessible. There are also some slips where systems like ACLS, ROSIE and OPS-5 are mentioned without any further reference being given.

In spite of some unevenness between chapters, the book is a good introduction—it indicates what an expert system is, and some of the possibilities, choices and problems. The authors speak from experience and express many of the central notions very clearly.

Peter Chubb
University of New South Wales
The European Conference on Optical Communication (ECOC) is one of the major international conferences on optical fibres and their applications. The tenth ECOC was held in Stuttgart in September, 1984 and this volume contains all the papers from the conference proceedings, together with post-deadline papers and the text of speeches from the opening session.

Contributed papers at the conference were divided among the following sessions: Planar waveguide devices; Laser diodes; Fibre measurement; Sources; Wavelength division multiplexers; Coupling and launching; Special fibres; Fibre characteristics and design; Detection; Broadband systems; Subscriber network and LAN; Fibre and cable design; Fibre manufacturing; Hydrogen in fibres; Poster session (miscellaneous).

There was also a series of invited papers on:

— Activities of the Deutsche Bundespost towards the introduction of Optical Fibres in the Subscriber Line Network
— The Technical Lessons of the Biarritz Experience
— Stacked Planar Optics by the use of Planar Microlens Array
— Characterization Methods for Single-Mode Fibres
— Progress in Dynamic Single-Mode Lasers
— Influence of Hydrogen on Optical Fibres—Implications and Potential Solutions
— Fibre Nonlinear Optics: Problems, Limitations and Opportunities
— The Incidence of New Epitaxial Techniques on Future Opto-Electronic Integrated Circuits

In 1984, for the first time, the conference program included a series of tutorial papers. Unfortunately these papers, which might well have been the most useful for readers of this journal, have not been reproduced in this volume.

As the list of session titles indicates, the 130 papers presented cover the whole spectrum of optical fibre activity as it relates to communications. Moreover, although the title of the conference is "European", participation was much more diverse and there is a large number of contributions from Japan and the USA. So the volume gives a very good overview of the state of the art in 1984. Since each of the papers is restricted to four pages, however, it should be regarded as a guide to further reading, rather than a source of detailed information.

The one very important area which is not covered is that of Optical Fibre Sensors, as this was the subject of a separate conference held in conjunction with ECOC.

On the aesthetic side, the volume was produced from camera-ready copy supplied by authors and reproduction is generally very good. In order to spare the bishops of those attending the conference, the proceedings were printed "sideways" i.e. two reduced manuscript pages per page. While I appreciated this at the time when I was sprinting between parallel sessions, the reader in the comfort of his/her office might find it a little annoying and it might have been preferable to revert to conventional format for this post-conference, clothbound edition.

R. A. Samsuat
University of New South Wales


The book aims to provide both a theoretical and practical exposition of relational database systems and succeeds quite well. The book is a translation from French of Bases de Donnees et Systemes Relational (Relational Databases and Systems), Bordas, Paris, 1983. In a rapidly evolving field such as database management, a new edition of a good book is required every few years and this book loses somewhat because of the time lag between the appearance of the French and English editions. For example, although this is a 1985 edition, all the papers cited date from 1981 or earlier (with one exception) and similarly all the systems described date from 1981 or earlier.

The book is divided into three sections. The first contains a general introduction to the database concept and provides a theoretical framework for the N-ary relational model and the binary relational model. The second section deals with implementation issues. An illustration of the difference in approach between the two sections is that whereas tuple and domain relational calculus are introduced in the first section, the corresponding implementation languages, QCEL and QUERY-BY-EXAMPLE respectively, are described in the second section. The second section contains quite thorough treatments of the data structures and query optimization techniques used in relational systems. Other topics covered in this section include the architecture of relational systems (data definition, view definition, catalogue description etc.), integrity constraints, concurrency and security. The third section offers a theoretical treatment of data dependencies and normalization theory. Throughout the book, well chosen examples are used to illustrate points made in the text.

It is not usually necessary to mention issues such as the standard of presentation, but the number of minor errors such as typing errors, together with the poor quality of printing, detract from this otherwise good book.

The book would be suitable as a text for an undergraduate course on relational database systems that aimed to cover both theoretical and practical issues. As a text though, it would have to supplemented by student exercises drawn from other sources since no student problems are contained in the book itself.

R. Sacks-Davis
Royal Melbourne Institute of Technology
The title of this book is misleading. It is really about program transformations, in particular transformations from recursive programs to iterative ones. The author argues strongly that it is easier to write provably correct recursive algorithms, often without assignment statements, than to discover the appropriate loop invariants to construct a correct iterative algorithm. If you disagree with this then you need read no further; I find it a very plausible argument. Having found a correct recursive algorithm, one can convert it, by transformations which preserve its correctness, to an iterative form. Some people might argue that this transformation is unnecessary since we can run the recursive form but the author shows that simplifications become obvious in the transformation process and also that the choice of transformations can produce surprisingly different versions from the same recursive original.

Three main types of transformation are discussed. Firstly an intermediate type of program, ‘recurrent’, is used which still has no assignments but instead refers to vectors of successively computed values; this is often easy to construct from the recursive algorithm and in turn easily converted to an iterative form. Secondly there is a detailed study of recursive functions where a single recursive call is made and its value combined with some non recursive function of the arguments to obtain the final result; this is a very general schema and various common special cases are shown to have simple transformations to efficient iterative forms. Thirdly sets of mutually recursive subprograms are analysed to determine when an explicit stack is required and when a simpler data structure, or none at all, suffices to determine the control flow of the iterative version.

The interesting ideas in this book are unfortunately made harder to grasp by an amount of idiosyncratic terminology and notation. For instance loops written in a form such as

\[
\text{DO f((EXIT(0),EXIT),EXIT) OD}
\]

take a while to get used to, writing ‘:’ for the integer division operator is confusing and the apparently important term ‘generalised action’ is introduced without any clear indication of where it is defined.

For these reasons the book is likely to appeal only to keen students of program transformations and not to those who might like to browse in hope of finding useful ideas. F. Duncan’s translation is unobtrusive, apart from the odd use of ‘method of dichotomy’ for ‘binary search’, and there appear to be only a few typographical errors. I was surprised to discover that the program on page 154, which is mentioned in several places as a good example of a simplified program obtained by the methods described, could be further simplified by the omission of one of the three tests from the main loop.

J. M. Robson
Australian National University


A great deal of rubbish has been written about Ada Lovelace and her relationship with Charles Babbage. Particularly prevalent are the myths that she was one of the great women mathematicians of the nineteenth century and the world’s first programmer. Both are the myths that she was one of the great women mathematicians of the day. Perhaps no great deal of this is new, but the writing through Stein’s essentially scientific eyes and the coherence lent by the focus on the one individual meant that the book communicated effectively to one, such as myself, not steeped in the social and humanities traditions.

References


Allan G. Bromley
University of Sydney


With their rapidly expanding range of applications and with the theoretical and experimental challenges they pose, amorphous semiconductors form the new frontier of solid state electronics. The Japanese have been at the forefront of the exploration of this frontier. This volume summarizes Japanese progress in this area to mid-1984. Two earlier volumes in the series (Volumes 2 and 6) report progress in the same area in the previous two years.

This volume consists of 27 well selected reviews of developments spanning the entire field by those closely involved with these developments. Topics covered range from the basic material and device physics through material and device characterization to device applications. Although the use of amorphous silicon in solar cells has been responsible for the growth of interest in this area and the contents of the volume reflect this, considerable space is also given to reviewing developments of more direct relevance to the computer field. For example the volume includes reviews of developments in the area of thin amorphous silicon MOS transistors for large area displays, in the use of tellurium based amorphous films for optical disc memories capable of storing very large quantities of information with rapid random access, in the use of amorphous silicon in contact linear imagers for reading printed material and the use of the same material as a photo-
conductor for electrophotography in digitally based copying machines and laser line-printers.

The volume would be of most interest to those directly involved in, or responsible for, research and development in the amorphous semiconductor area or in areas which benefit from this technology. In such a rapidly evolving field, any review must be quickly overtaken by developments. The present volume avoids this to a certain extent by the scope of its contents.

Martin A. Green
University of New South Wales


Containing 10 programs for acceptance sampling, process control and process capability study, this is on the whole a very useful package for the QC practitioner. Outputs include data histograms, OC, AOQ, average cost and average fractions inspected curves for attribute acceptance sampling and control charts, with action limits, for both attributes and variables inspection. A demonstration training program for production of random numbers and sampling times and illustration of attributes inspection and the limitations of 100% inspection is also provided.

The book gives step-by-step guidance for using the IBM PC and detailed instructions for running the programs, including the reproduction (not always quite accurately) of what will be seen on the screen or print-out at the successive stages of execution. Some of the data entry could be simpler, e.g. in the Attribute Sample Size and Precision program 7 keys have to be hit to enter 99.71% confidence level, despite this menu item being displayed on the screen.

Graphics capability is good, permitting the display of bar charts, curves and control charts. The outputs can be readily examined on the screen and then printed out in the same form.

Capability, precision and speed are reasonable. For the binomial distribution the Normal or Poisson approximations are used for sample sizes above 125. For the hypergeometric distribution the maximum sample size is 33. With simple data sets the computer response is fairly immediate. For more complex calculations the wait can run up to minutes. During some lengthy calculations the screen was empty, a frustrating experience. A message ‘CALCULATING ...’ would have been welcome.

The authors are much less expert at writing a manual than in writing program code. There is no consistency about the assumed level of knowledge of the reader, e.g. we are expected to understand the terms of ‘degrees of freedom’ and ‘Student’s t distribution’ but must have ‘factorial’ explained to us. Chapter headings and section headings are not always relevant to the contents. For example, Chapter 1 is entitled Techniques of Quality Control but less than half actually deals with this, in a rather sketchy way, – the rest discusses how to run the program, move files etc.

There are many instances of slipshodness in writing and proofreading. Thus the equation for sigma on pages 26 and 27 is wrong, omitting division by $\sqrt{N}$, although the calculation actually performed by the computer is correct. In the first paragraph of the same chapter it is stated that ‘the program assumes sampling with replacement, or large sample sizes’. It really assumes large *N* sizes. There is considerable confusion between AOQ and AOQL, illustrated by the Index entry ‘AOQL, see Average outgoing quality’ and by the listing under Average Outgoing Quality Limit of only two introductory pages, 5 and 6, of which the latter only refers to AOQ.

Some explanations are very unhelpful. What, for example, is one to make of the last paragraph on page 108 in connection with known or unknown sigma plans: ‘The difference is relative to the degrees of freedom for a given confidence level. When known sigma plans are asked for, the degrees of freedom are set at a very number.’ Some explanations are missing, for example the meaning of the symbols in the control chart print-out in the Tool-Wear program. (The figure on page 118 incidentally is not a complete reproduction of the actual program output.)

Amongst the errors in the manual is a major inconsistency between the tabulated output data on page 103 and the printed-out control chart on the following page. Both are inconsistent with the input data (e.g. see samples 7-10). Furthermore the chart does not show the control limit variations due to sample size variation. None of these errors appeared when the program was actually run, using the suggested illustrative input data and sequence.

The title of the program of Chapter 8, Acceptance Sampling By Variables, (see Sect. 8.2) is entirely misleading. More relevant to process control, it produces only action limits for a given variables specification and sample size. No OC curves can be produced for a given sampling plan, or acceptance criteria determined for a specified AQL and sample size.

The other programs are however apt. This is a useful package to have for a large range of quality control tasks. One just wishes more care had been taken with the manual.

Henry S. Blanks
University of New South Wales


This text book contains many statistical algorithms that should be part of the repertoire of any competent programmer. It is not, however, a book for beginners in either statistics or Pascal programming; it requires a rudimentary knowledge of both.

The book begins with some introductory chapters on programming techniques and style. Particularly welcome here is the section on the effects of errors in the representation of real numbers in computers, a topic which is revisited during discussion of particular algorithms later in the book. Some simple algorithms for the sorting and presentation of data are also given in the introductory chapters.

Statistical techniques presented include variance, correlation coefficients, significance tests, regression and analysis of variance. For each of these, a brief introduction is given, followed by a presentation of algorithms, usually in the form of Pascal procedures, for the technique. Little theoretical background is given, and there is little information provided about how to interpret the results of the algorithms. For these, the reader is referred to standard statistics texts, and in particular to a previous book by the same authors (Clarke and Cooke, 1983). References to this latter book are frequent and detailed, and in order to get the best out of the volume under review, it is really necessary to have both it and the previous book.

Another group of techniques presented is concerned with simulation, particularly the generation of pseudo-random numbers from various distributions. Various tests for randomness are also described.

The many programs contained in the book are well structured and laid out. They are well documented, so that most of them can be understood, without reference to the reference text. Each chapter concludes with a set of exercises, mainly programming exercises. The book concludes with appendices for the longer algorithms and for test data. A bibliography and an index are also provided. It is difficult to think of a course which could use this book as its only text. However, in combination with an introductory statistics text, it would be useful for a practical, Pascal-based statistics course. Programming students who wish to gain expertise in numerical processing would also find much to interest them. A third possible readership is researchers in other fields who wish to gain expertise in using computers for simple statistical analysis.

References


Bill Beaumont
University of Adelaide


This book gives a broad overview of parallel array computers, pipelined architectures and multi-computer networks. Particular emphasis is placed on computer architectures which match the structure of flow of information of the algorithms which will be executed on them. The discussion includes computers which have been built and architectures
The problem is that of incorporating, in a natural and meaningful 'Lemma' and 'Remark'. While this leads to a very concise presentation it has the disadvantage of being rather technical and therefore not suitable for a wide audience amongst engineers and moreover, mechanical engineers since they are not familiar with the terminology and concepts used in the field of optimization. In tertiary mechanical engineering teaching institutions for some time, there has been a growing demand for a book that provides a comprehensive treatment of the subject.


This is indeed a book on the theory of multiobjective optimization. Apart from a few motivating remarks in the introductory chapter, the text is largely devoid of examples of applications. The material is presented using the headings: 'Proposition', 'Definition', 'Theorem', 'Lemma' and 'Remark'. While this leads to a very concise presentation it sometimes becomes a little too terse for easy reading.

The first half of the book has a 5-page introductory chapter, a 25-page chapter on mathematical preliminaries (convex analysis, point-to-set mappings, preference orders and domination structures), a 60-page chapter on properties of solutions, and a 35-page chapter on stability of solutions with respect to perturbations.

The second and more interesting half (to this reviewer) consists of a 40-page chapter on Lagrange duality, a 40-page chapter on conjugate duality, and a 70-page chapter on methodology. In this last chapter a concise coverage of most of the recently developed methods for multiobjective optimization is given. A reasonable discussion of the usefulness of interactive programming as a way of attacking multiobjective problems is given, and this area would be of most interest to computer scientists.

This book would sit on the shelves as a very useful reference work for both the scholar and practitioner. The mathematics is quite deep although the authors claim that only an elementary knowledge of linear algebra and mathematical programming is required; some background in convex analysis would be helpful.

B.A. Murtagh
University of New South Wales


This book comprises twenty papers selected from those presented at the 50th Annual Meeting of the Pacific Southwest Section of the American Society for Engineering Education which was held in the United States in October, 1984. The book therefore has its primary audience amongst engineers and moreover, mechanical engineers since the papers address a problem which has occupied the minds of educators in tertiary mechanical engineering teaching institutions for some time.

The problem is that of incorporating, in a natural and meaningful fashion, an understanding of computer usage (in a broad sense) amongst students.

This task is not as simple as it might first appear. This is particularly the case in the critical area of design where many of our most competent educators have developed their expertise remote from the impact of computers. There is, firstly, a reaction against encouraging mechanical engineers to become electrical engineers. However, a certain level of familiarity with circuit fundamentals must be achieved before designers can accept digital electronic devices as a natural and integral part of their array of design options.

A second problem relates to the incorporation of computerised solutions to problems of design optimisation. This is often seen as an exercise which not only complicates the teaching process but also tends to obscure the recognition of fundamental concepts and leads students to be uncertain of solutions they develop. Finally, although it is recognised that computerised drafting is a first step to the important concepts of computer integrated manufacturing, the benefits, using currently available packages, do not reveal themselves, at the conceptual design stage, to engineers accustomed to developing their ideas with pencil and paper.

The papers cover all of these topics and the authors offer a variety of tentative solutions. The book is, in my opinion, well worth reading and, for mechanical engineering faculty, it offers a valuable source of pump priming material to allay fears of this new technology.

Jacob A. Carmel
University of New South Wales


This book is for everyone. It is an Australian magazine-style publication available from newsagents. Its aim is to increase 'awareness and understanding of the uses of information technology in today's society' and 'to cut away the mystery and the myth from computers'. The book addresses both issues in a constructive and enthusiastic manner. The current and future uses of computers in sport, health, welfare, industry, communications, arts, schools and homes are presented. The language used is clear and there are many colour photographs used to illustrate each topic. The book can be readily understood and enjoyed by all people including the 'computer literate'. The book will extend the knowledge of almost everyone as its scope is wide and each section has been prepared by people with considerable understanding of the topic and with a vision of the future.

The section on Arts and Entertainment is a good example of the approach taken in this book. The use of computer graphic systems to design logos and construct advertisements is explained clearly. There are beautiful pictures and excellent text explaining the processes involved in animation, the control and production of theatrical news. The technical content is useful but never overwhelming. Costs are also given. There are cleverly devised headings throughout the book. Daily issues such as home computers and the use of computers in teaching are brought into perspective. Computers are shown as part of complex systems such as medical CAT scanners, electronic sporting scoreboards with action replay capability, environment control for plant nurseries and sheep shearing robots.

A totally enthusiastic view of computers is presented. We are told all the reasons why computers are useful but the book omits any mention of issues which concern the community (the Australia Card debate is a good example). The often anti-social nature of computers in the home could be contrasted by the reader with the happy family groups depicted in the book. Computers demand our attention and our children respond with aggressive glee or tears. In the work place they alter people's jobs - and not always for the better. They are used as an excuse for poor human performance. We need a lot of good system designers and a well-informed 'user community'. Men and women, boys and girls must be involved. There are not many girls or women involved these days and this gets no mention. You and Computers has a limitation by omission.

Benefits, however, outweigh limitations in this book. It is up-to-date in its perspective and provides a lot of useful information. Many people want to learn about computers and don't know where to start. This is a very good starting point. Children and adults will enjoy this book which I recommend to you.

Penelope A. Collings
Canberra College of Advanced Education

This book contains the proceedings of an IFIP Working Conference organised by the IFIP Technical Committees on Education and on Computers and Society. It is probably best described as a book on computer sociology.

The avowed aim of the conference as set out in its summary was as follows:

"In order to achieve systems which more adequately meet the requirements and ambitions of users, there have recently been many attempts to get users to participate more actively in the design of systems. However, cooperation between users and designers has not always proved easy, partly because of the disparity of knowledge between the two; ..."

This conference aims to tackle the problems of facilitating the cooperation of users and designers through considering in particular the educational processes which might contribute removing the above-mentioned disparity of knowledge."

Later in the same conference summary it is essentially acknowledged that this aim was not met, mainly due to 'lack of either a firm theoretical base or an extensive body of empirical knowledge to serve as a framework for ordering our ideas.' This is certainly the impression gained from reading the conference papers, nearly all of which contain little of substance which could be of use to a computer professional involved in the day-to-day practicalities of dealing with users.

The apparent lack of substance is possibly exacerbated by the brevity of the papers, many of which run to under two printed pages. The papers as delivered at the conference may well have been valuable, but by the time they are summarized to two pages, any depth is lost.

The value of the conference to practitioners could have been enhanced if it had contained some component of computer science mixed in with the sociology. There exist practical methodologies (such as CORE, SADT and Jackson Structured Design) which can be used to great benefit in achieving the aims of the conference. They were completely overlooked by the conference.

In summary, the proceedings contain a rich choice of research topics for computer sociologists but little of benefit to practitioners.

N.D. Birrell
Electronic Publishing Division,
John Fairfax & Sons Limited


This is one of a collection of guides within the BCS Monographs in Informatics series, and deals with the legal, accounting and control features which should be considered in payroll software. Three other guides are available dealing respectively with financial accounting software, purchases software and sales software. The purpose of the publication is to help those involved in buying payroll software, particularly for mini and micro computers, and to highlight the risks involved in this area.

The booklet is divided into three main sections and concludes with a useful glossary of computer terms used. The first section deals with requirements under the following headings:

function requirements — what the system will be required to do;
data requirements — the information that will be required to do it;
data updating — how this information is input to the system and kept up to date;
links to other systems — identification of other systems which may provide payroll input or require payroll output.

The second section deals with controls under the following headings:

the need for controls — the penalties of inadequate control;
access control — who can use the system and for what;
management trail — how to trace what has happened in the system;
input control — input checks, batch control and input validation;
processing control — ensuring that the system is generally secure;
output control — providing the right environment for the system.

The third section covers buying a package under the following headings:

general rules — points to consider before starting;
evaluation of payroll packages — steps involved in determining suitability;
capacity — establishing whether the computer is big enough for the job;
cost comparison — determining which system provides best value;
implementation — steps involved in implementing the system including documentation to be obtained or prepared.

In summary this plainly written booklet would be an excellent investment of both time and money by anyone considering purchasing a payroll software package. The principles enunciated and detailed checklists provided would form an excellent basis for payroll software acquisition in the small to medium sized organization. Naturally there are some matters of legal and taxation detail that require translation from the British situation to the Australian equivalent but this should not detract significantly from the worth of this publication.

R.C. Reeve
University of New England


Originally published by the National Computer Centre in the U.K., this is a guide to the current ANSI and ISO standard (ANSI X3.23-1985). The standard itself is not easy reading; it is often hard to see the wood for the trees. Brown and Gwillim have reorganised the material in the standard and added new material, to help the reader understand Cobol's central concepts and the motivation for its features. The authors' style is less formal but more concise than the standard; similar in level to a well-written compiler reference manual. Although it contains some useful explanations which will help an experienced programmer learn about new features of the standard, it is not intended as a text-book.

The book contains a chapter describing its relation to the standard, a chapter on Cobol concepts and facilities, 10 chapters defining the Cobol language, and a final chapter on program portability. There are 6 appendices, one of them a glossary, but no index. Although the book is shorter than the standard, it contains almost as much information. How has this trick been achieved? Part of the answer lies in its organisation. Cobol is implemented in 11 modules, most of which affect more than one division of a program. By grouping eight of the modules together, the authors are able to devote separate chapters to program structure and to the three main Cobol divisions, bringing related material together, and saving the reader needless cross-references. The three modules treated separately are relatively self-contained; Debugging, Report Writer, and Communications. They also offer a commentary on each Cobol feature to explain its purpose and usage, avoiding the legalistic prolixity of the standard.

It took some 5 years for Cobol 85 finally to be standardised. This means that its concepts are almost eight years behind the times; for example, it does not offer any specific facilities for screen-oriented output. Brown and Gwillim were obviously anxious to avoid any further delay in making their translation available to the public. Unfortunately, although most of the book is of a high standard, it shows signs of hasty preparation. I noticed a confusing error in a flow-chart that is intended to explain the PERFORM verb and in a discussion of contained programs there is a reference to a section on the scope of names, which seems to have been omitted in the rush to publish. Also, although the authors state that they devote most of their commentary to the new features of Cobol 85, this is not really true. The chapters on the Report Writer and Communications modules, which are not new features, contain the most examples and most detailed explanation by far, and perhaps belong in a text-book rather than a reference book. Their level of detail would have been better used in the discussion of program structure and the scope of names. Some difficult syntactical constructs were illustrated by useful examples, but explanations of their meanings were not always included. The book could be improved by adding an index.

Despite its occasional flaws, this is still a very useful book. The authors are clearly interested in the development of Cobol, and the pro-
tection of its users through standardisation. But they are not afraid to
criticise the standard when they see fit, and to point out those things
that the standard has left unclear or undefined – which are often hard to
discover from the standard itself. The chapter on portability is especially
useful. If you want to write Cobol programs that are not locked-in to
particular hardware, or you are an educator teaching standard Cobol,
you need this book. You are likely to extract far more truth from it than
from the standard itself. In its hardcover form, it is durable, and will
perhaps outlast Cobol 85.

Barry Dwyer
University of Adelaide

Computer-aided Data Base Design - the DATAID Project. Elsevier
Science Publishers, B.V. (North Holland), Amsterdam, 221 pp.,
US$44.50 (hardcover).

The book is a collection of papers, in various type fonts, written by
researchers and developers from the DATAID project consortium of
Italian universities and industry. Their subject is computer-aided design
of business information systems.

The project is important in terms of the effort and focus applied. The
project, by virtue of its participants, has a developmental emphasis. I
suggest that anyone engaged in research and development of data
(information) analysis and database design methodologies should at
least be familiar with this project. It may also be useful to specialist
practitioners as a reference book.

The material in the book is poorly integrated, somewhat in contrast
to the stated objective of database design in Chapter 10, namely
'Designing the logical and physical structure of a database in a given
database management system to contain all the information required by
the users in an organisation'. One has to search hard to find a coherently
stated framework within which the DATAID project is cast. However,
with too little help from the prefacing chapter, one discovers that:
1) an organisation must first be divided into environments which are
'homogeneous so far as operations are concerned' (p. 186) and that this
process has been called 'enterprise analysis' or 'business' functional
analysis' by others;
2) the conceptual schema (or model) can then be defined which consists of
data classes (entities and relationships), integrity constraints,
opérations on data classes and (database) applications dynamics, which has
to do with causal or time ordering of events that affect the system;
3) finally, with some additions for distributed databases, logical and
physical database design can proceed (p. 159).

Fortunately there is an emerging ISO standard from ISO/TC97/SC5
(van Griethuysen, 1982) to assist us understand what we are trying to
talk about.

The various contributions are quite mixed in their standard and
approach. The major difficulty is that in order to describe a methodo-
dology or automated tool it is necessary to state what one is trying to do
and why. The difficulty is apparent from the previous paragraph.
A second difficulty is using a (necessarily) static text, even with many
figures and diagrams included, to describe an interactive tool, parti-
cularly a tool that uses graphics. This caused me to skip through three
chapters that described such tools.

However, contributions on Galileo a very high level language for
schema definition, suggest the nature of what lies ahead. Two chapters
dealing with physical database design for CODASYL network and re-
tional DBMS are very interesting reviews and describe the DATAID
project methods being developed in this complex area that is so
important for computer performance. A chapter on database dynamics
describes the theory and background to this topic but does a poor job of
exemplifying it. This is probably because database dynamics is really
only interesting for systems too complex to be described adequately in a
short chapter.

Perhaps this review is a little harsh. It is and always will be an
extremely difficult task to describe simply the extremely complicated
information systems we are engineering to-day. It will be quite a lot
easier when the international standards referenced below are completed
and companion information systems and database engineering methods
are developed and become accepted in the specialist community.

References
TC97/SC5/WG5 (obtainable from the Australian Standards
Association).

C.N.G. Dampney
Macquarie University

Information Systems: Theoretical and Formal Aspects, Proceedings of
the IFIP WG 8.1 Working Conference on Theoretical and Formal
Aspects of Information Systems, Barcelona, April 1985, North-
Holland, 236 pp., US$35.25.

The title of this book provides a good indication of what's in store for
the reader. 'Theoretical and formal aspects of Information Systems'
suggests that the content will be data modelling since it has been the
database area, and particularly the data modelling aspect of database,
that has been generating much of the theoretical interest in information
systems over the last five years.

The book is broken down into five sections entitled: Systems and
Languages, Modelling Approaches I, Modelling Approaches II, Logical
Frameworks and Modelling Approaches III. Overall, it provides a good
up-to-date summary of the type of work which is being done in Infor-
mation Systems modelling. The papers chosen for the conference reflect
a good cross-view of the techniques and approaches which are popular
at present in the theoretical literature. The reader should not expect to
find in this collection of readings, a set of techniques which could be
applied to the task of designing and building information systems today,
but rather a glimpse into some of the problems we face as designers and
some of the potential solutions to those problems.

Most of the papers in the proceedings are concerned with data
modelling, usually at the conceptual level. One exception to this is the
paper by Mark and Roussopoulos entitled 'The New Database Archi-
tecture Framework - a progress report' in which the process of mapping
from the internal data model to the physical data level is considered.
In this paper they propose a database architecture aimed at providing a
standard structure upon which the current internal models can operate.
The framework presented is based on the two-dimensional classification
of data:
1) 'The point of view dimension' as presented in the ANSI/Sparc model
and
2) The meta data dimension (data about data).

The architecture discussed aims at providing a database environment
upon which any DBMS model (network or relational) can be applied,
using the data language interface as the communicator. The paper also
provides a clear over-view of the two dimensional data classification and
outlines the progress to date of the ANSI/Sparc Data Base Architecture
Framework Task Group.

The majority of the papers are concerned with the upper level of the
ANSI/Sparc model. The Demo et al. paper, 'An entity relationship query
language' outlines the broad detail of a query language based on the
entity relationship (ER) semantic data model. It therefore uses the ER
constructs of entity, relationship, and attribute. The query language is
built on three concepts:
1) The query view (entity or relationship types)
2) The query population (the occurrences involved in the specific
query, e.g. selection criteria)
3) The resultant entity type (the query result massaged to satisfy the
query requirements).

Anderson and Claghorn in their paper 'ADE: Mapping between the
external and conceptual levels' also concentrate on the upper levels of
the ANSI/Sparc architecture. They provide a system and grammar to
incorporate the various external level views of the database in a formal
grammar. The motivating force behind this is to allow the sharing of the
additional data embodied in the external views between different user
groups. The modelling technique presented is based on the binary data
model and allows both the representation and format of particular
entities to be described. In this way, different representations of the
same object, for example, date, can be accommodated and different
formats of that data can also be modelled.
Other papers look at different problem aspects of the data modelling area. For example, Horndasch et al, describe a modelling technique aimed at generating a conceptual model. It is a two phase process in which the first phase, described as requirements modelling, attempts to model both the functional behaviour of the requirements and the information flow firstly, and secondly to model the data structures using a semantic binary modelling technique. These two models, which could be described as the process model and the data model, are then formalised by transforming the requirements model into a high-level net.

Other papers concerned with conceptual modelling include the Morgenstern paper which uses the 'hypergraph' concept to provide a modelling approach which will allow multiple mappings between the same attributes. The data maps defined, using the technique presented, are said to allow a simple approach to the problem of defining and using multiple relations with different roles involving the same attribute columns, without the need to rename those attributes. This is achieved by defining the data items or entities which participate in a named role.

The Pletat paper defines formal semantics for representing a data model using the entity relationship approach, and the Lipok et al paper provides a method for expressing data base integrity constraints.

The book ends rather abruptly with sections on power supplies and electromagnetic compatibility. The text does not contain any section on magnetic recording or display systems, all important to instrumentation and control.

In summary, I was disappointed that this book was so superficial in its treatment of such an important and growing subject area. On the other hand the book is quite readable, with operations and principles clearly explained. I hesitate to recommend it to a beginner as it may give an outdated and inappropriate view of the field.

D. H. Mee
University of New South Wales


This book is designed to support all United Kingdom Year 11 Computer Studies courses, and is relevant to Australian Year 11 computer work. The book has now been revised to reflect changes in the various Computer Studies syllabi, and to present computing as part of the broader discipline of Information Technology. It is set to concern computing, data communications and automatic control, and the latter two topics are also mentioned in the book. Other new topics are chip fabrication, software engineering, and the Fifth Generation topics of artificial intelligence and expert systems.

The book deserves the popularity implied by its 5 reprints, since it is easy to read. This is partly because its coverage is very wide (presumably due to its being the union of all the Computer Studies syllabi), and hence the subsections devoted to each topic are very short. This sometimes leads to shallowness (the section on Teletext runs to only 56 words), to unsupported detail (as in references to pn junctions) or to lack of balance (ferrite cores get four times as much space as Teletext). A teacher using the book would have a good working basis, but would have to supply quite a lot of ancillary breadth or depth. The book is therefore not ideal for reference but chapters can (to some extent) be read out of sequence.

There are plenty of exercises (with answers) and a large number of photographs and figures. The short glossary is useful and the index is adequate, albeit with some gaps. There are several books which address Years 11 and 12 computer work, but this one is worth the specified price as an introductory reader in Computer Studies at Year 11.

David Woodhouse
La Trobe University

AWA CONTINUES SPONSORSHIP OF AUSTRALIA'S PREMIER YACHT RACE

Australia's largest electronics company, Amalgamated Wireless (Australasia) Limited, is to sponsor the annual AWA Sydney Hobart yachting classic until at least 1989.

AWA and Race organisers, the Cruising Yacht Club of Australia, announced the extension of the sponsorship following recent talks in Sydney.

AWA became the major sponsor of the Race in 1984. CYCA Commodore Mr John Brooks said today that the Company had provided magnificent support for Australia's most prestigious yacht race.

Mr Brooks said the Board of the CYCA was unanimous in its decision to accept AWA's offer of a further three years of sponsorship after the 1986 AWA Sydney Hobart Yacht Race.

AWA then has the option of a further three years of sponsorship after 1989.

In addition the Company will continue to sponsor the AWA Southern Cross Cup, the biennial international teams racing series which culminates in the AWA Sydney Hobart classic.

The 630 nautical mile annual AWA Sydney Hobart yachting classic ranks among the top three long distance ocean races in the world. It began in 1945 and last year attracted a record 179 starters including entrants from Great Britain, Hong Kong, Bermuda, France, Papua New Guinea, New Zealand and all Australian States.

AWA has been involved with the Race for the past 18 years, providing the sophisticated communications equipment and operators for the radio relay ship which escorts the fleet to Hobart, relaying regular position reports back to race organisers in Sydney and Hobart.

Upon becoming the major sponsor, AWA developed a unique computer analysis system for instant updating of yachts' progress on line honours and corrected time throughout the Race. In addition it provides race officials and safety authorities with further specific position information in the event of an emergency.

Mr John Hooke, the Chairman and Chief Executive of AWA, said the Australian-owned company was delighted to continue the sponsorship, with sub sponsors Ampol and TAA, of what he described as a vital part of Australia's sporting heritage.

MacCOBOL NOW IN AUSTRALIA

For the first time in Australia computer programmers and systems developers can use COBOL on the Apple Macintosh Computer.

MacCOBOL provides access to 386 of Mac's 512 ROM routines. High-level features include compiling Calls directly from ROM routines, the Micro Focus Level II Compiler, an ANSI '74 compiler and ISAM file handling. Sophisticated tools such as Generate, Run and Build are there, and Micro Focus' existing applications are cross-compatible for easier hardware changes.

The importers, whose team has many years' Cobol experience in DP environments, fully support MacCOBOL. Call Tim Harvey at Computer Connection, Miranda NSW (02) 526-1404.
OFFICE BEARERS: President: R. Christie; Vice-Presidents: M.L. Cattermole, J. Goddard; Immediate Past President: A.W. Coulter; National Treasurer: R.G. Heinrich; Chief Executive Officer: A. Kelly, P.O. Box 319, Darlinghurst, NSW 2010 telephone (02) 211 5855.


SUBSCRIPTIONS: The annual subscription is $20.00. All subscriptions to the Journal are payable in advance and should be sent (in Australian currency) to the Australian Computer Society Inc., PO Box 319, Darlinghurst, NSW 2010. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are now four issues per annum. The price of individual copies of back issues still available is $2.00. Some are already out of print. Issues for the current year are available at $5.00 per copy. All of these may be obtained from the National Secretariat, P.O. Box 319, Darlinghurst, NSW 2010. No trade discounts are given, and agents should recover their own handling charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $2.00 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries; Canberra: PO Box 447, Fyshwick, ACT, 2609. NSW: 1st Floor, 72 Pitt Street, Sydney, NSW, 2000. Qld: Box 1484, GPO, Brisbane, Qld, 4001. SA: Box 2423, GPO, Adelaide, SA, 5001. WA: Box F320, GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005. NT: GPO Box 213, Darwin, NT, 5794.

CONTRIBUTIONS: All material for publication should be sent to Associate Professor, J. Lions, Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington, NSW 2033. Prospective authors may wish to consult manuscript preparation guidelines published in the February 1986 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews. Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of Journal readers, or describe important computing experience or insight. Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal or discuss a computing topic of current interest. Descriptions of new software packages are also published to facilitate free distribution.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees. Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor's invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW, 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typed in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear bond-paper copies, should be submitted. References should be cited in standard Journal form, and generally diagrams should be ink-drawn on tracing paper or board with stencil or Letraset lettering. Papers and Short Communications should have a brief Abstract, Key word list and CR categories on the leading page, with authors affiliations as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

This Journal is Abstracted or Reviewed by the following services: Publisher Service; ACM Bibliography and Subject Index of Current Computing Literature; ACM Computing Reviews; AMS Mathematical Reviews; CSA Computer and Information Systems Abstracts. Data Processing Digest.

ENGINEERING INDEX INC. Engineering Index.
INSPEC Computer and Control Abstracts.
INSPEC Electrical and Electronic Abstracts.
ISI Current Contents/Compumath.
ISI Compumath Citation Index.
SPRINGER-VERLAG Zentralbatt fur Mathemamm und ihre Grenzgebiete.

Copyright © 1986. Australian Computer Society Inc.
Production Management: Associated Business Publications, Room 104, 3 Small Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007). Tel: 212 2780, 212 3780.

All advertising enquiries should be referred to the above address.

Printed by: Ambassador Press Pty Ltd, Parramatta Road and Good Street, Granville, NSW 2142.