POPLOG AND THE LEARNER:

An artificial intelligence programming environment used in education

Benedict du Boulay, Mark Elsom-Cook,
Tom Khabaza, Josie Taylor

1986

Cognitive Studies Research Paper

Serial No. CSRP.056

The University of Sussex,
Cognitive Studies Programme,
School of Social Sciences,
Falmer, Brighton BN1 9QN

Poplog and the learner: An artificial intelligence programming environment used in education

Benedict du Boulay
Mark Elsom-Cook
Tom Khabaza
Josie Taylor

Cognitive Studies Programme,
University of Sussex.

1. Introduction

This paper describes the Poplog system, developed in the Cognitive Studies Programme at the University of Sussex, together with some of the research on learning and using programming languages and environments that is currently in progress in this laboratory. The system is being used in many universities and polytechnics as well as in industrial research laboratories. Versions for both UNIX and VMS are available and it runs on VAX machines as well as a variety of systems based on the M68000 range of processors, such as the SUN and the Hewlett Packard workstations.

There are three current research projects of particular relevance to Poplog. First, because of the many on-line files a user may have real difficulty in locating the particular file that is believed to contain helpful information for the problem in hand. The "Help File Finder" project is attempting to address the issue of determining the mapping between a user query and a given help-file. The second project is concerned with the programming language Prolog. This project is investigating the difficulties that certain novices face in learning Prolog. This language is becoming widely used and there may be some disparity between the apparent simplicity of its syntax and the ease with which it can be learnt. The final project, "An intelligent program checker", is concerned with Pascal rather than Prolog. A system is being developed in Prolog and POP-11 to help novices in their early steps in Pascal by providing a program that can comment usefully about their programming errors, whether they be syntactic, semantic or logical.

The paper starts with a brief description of Poplog more from the point of view of students rather than that of systems programmers. The following sections briefly describe the three projects indicated above.

2. What is Poplog?

Poplog is a general purpose programming environment which provides incremental compilers for Prolog, POP-11 and Common Lisp integrated with a screen editor, VED, and other useful program development tools. The communication between these three core languages is intimate in that each compile down into a common virtual machine language (which itself is compiled). It is thus possible, as in the "Intelligent Program Checker" project, described below, to write short inter-communicating sections of code in say POP-11 and Prolog to exploit the best features of each. It is also possible to prototype a system first in Prolog (for clarity), and then move to a faster version written in POP-11 while staying all-the-while in the same environment. Sections of code in other languages, such as Fortran, can be linked in making it possible to write multi-language systems.

Much of the system is written in POP-11 itself, including the editor. This means that it is not only powerful but also easy to modify for particular users. POP-11 is an interactive language with capabilities similar to Lisp but with a syntax much more like Pascal. It can be used as a general purpose programming language, though its main use so far has been in research and teaching in artificial intelligence. Hardy (1984) provides a more wide-ranging description of the system, and especially its use as a POP-11 and multi-language environment. Gibson (1984) gives an excellent introduction to programming in POP-11 and a more complete description can be found in Barrett et al. (1985).

The screen editor plays a central role. It is used as a delivery vehicle for tutorial information and software as well as for the more traditional roles of constructing and editing both programs and text (such as students' assignments). Although VED is not a fully-fledged structure editor (in the "authoritarian" sense that would allow the contents of the edit buffer to be navigated only as structure) it does have knowledge of programs as well as text. So it is possible to call a pretty-printer to rearrange either text or programs to make them more readable or to compile some part or all of the buffer. Many of the most frequently used editor commands are associated with individual function keys, and VED also responds to commands entered on a special "command line" as well as to commands typed directly and preceded by typing the <escape> key. Both features can be tailored to suit different groups of users (or individual users). It is possible to communicate with the underlying operating system through VED, though many of the more useful commands are fielded by VED itself in a way that exploits the system to better advantage.

Incremental compiling

The notion of incremental compiling (as opposed to traditional compiling) is important in reducing the time and labour associated with the plan-code-test-debug cycle. By allowing pieces of program, either complete procedures and functions or even individual commands to be compiled and tested individually and then automatically linked into the growing code one reduces the wasted time consumed when other parts of the program are needlessly re-compiled after some small change to a subsidiary procedure. This gives very nearly the flexibility of an interpreted system but produces programs that run much faster. The compilers can be called from within VED, so that one frequently used mode of work is to "get into" VED and then conduct the whole program development process from there, editing and then compiling code as need be or using the other half of the screen to trace and debug the growing program.

Two other features of Poplog are relevant here. The first is the very large amount of tutorial text and software that has been built up over several years by the lecturers who have used the system for courses at various levels, from entertaining school pupils on Open Days to providing examples of advanced parsing techniques for graduate students. The tutorial text files are called "Teach" files and the software files are called "Library" files. The Library files enable complex new programs to be built out of existing programs with a consequent saving of labour and possible increase in modularity and clarity. They also enable students to do interesting things fast. The contents of the Teach and Library files are very easily accessible and can be scanned, edited and cannibalised at will using the screen editor.

The other useful feature is the large amount of online documentation about the system itself and about the languages that can be used. It is easy to ask for help about some language feature (if you know how to find the name of the appropriate file) and so get some text and some pointers to other files that might be useful.

All these files are not without their drawbacks. Their sheer number make the task of finding what one wants difficult (especially if as a beginner one is not sure of what exactly one wants to know or of how the information might be arranged in the system). A related problem is that of trying to keep the system under proper control by providing useful cross-links and making sure that new features are correctly integrated into the existing complex web of helpful references. Work on these issues is described in section 4.

3. Using Poplog in Teaching

The large amount of online documentation, together with the many library and utility programs makes Poplog a very useful teaching device. Many of our courses have a strong practical element in that students are expected to be able to understand, extend and eventually formulate programs to carry out AI tasks such as parse English sentences, interpret representations of visual scenes or demonstrate some other expertise. The Cognitive Studies Programme runs a number of undergraduate degrees including "Computing and Artificial Intelligence". It also povides a context for this degree and others in Linguistics, Philosophy and Psychology by introducing them to the study of human cognition from all these different perspectives. Various post-graduate degrees are also offered including a 1 year SERC sponsored "conversion" M.Sc. in Knowledge-Based Systems as well as degrees based on research.

Our students arrive with widely varying levels of experience in computing and technical subjects in general. This has meant that the Poplog development team has had to take seriously the notion that the system would be used both by beginners as well as by very experienced people who use it for their research in artificial intelligence. Many of our students have no background in computing and have always studied "Arts" subjects. One way that we try to induct the more timid of them into computing is by introducing them to the Poplog editor. This they use both as a word-processor and as a means of navigating around some of the initial teaching material. While this may seem an inefficient use of resources and time it is a much less threatening way to start than being plunged into writing programs. Another advantage is that students can progress at their own pace through the materials, becoming more familiar with the system all the while.

There are no "intelligent" teachers lurking inside Poplog. Many of the teaching files suggest activities that the students might try, give them pieces of program to run and so on, but the system itself does not evaluate the outcome of their experiments. There are good reasons for this choice. First it is extremely difficult to build effective computer tutors. Second we wish to emphasise that the system is a tool for the students' (and our) use and not muddle up the interactions with issues to do with evaluation. Given the difficulties of building tutors it seemed better to put effort into *reeking* the facilities powerful and convenient to use. The intelligence is in their design rather than in their responsiveness to the user. However we are interested in intelligent systems as such as the later discussion on "help file" *finding* and program checking will show.

Tailoring the environment

Once the students are logged in the system executes a "login" file which immediately puts them in a specially tailored Poplog environment that almost completely instates them from the underlying VMS system or UNIX system if they wish. In some cases the chosen environment is the screen editor VED working on a special file called "output" in which all I/O from their programs is stored. There are login files provided for all the different courses that we teach. They may put the system a special state at the start (as above) or set up default pathways to preferred Teach and Library files. This means that the system scans the Teach and Library directories in a given order depending on the value that has been set, thus making certain files available to some courses but not to others. Similar start up files are associated with other parts of the Poplog system. For example the system will execute a user's file associated with VED when the editor is first called after login if such a file exists. By this means many of the faculty (and some of the more advanced students) interact with a version of VED, POP-11 or Prolog tailored to their own needs.

Staff and students have access to cursor addressable terminals with limited graphics capabilities (such as Visual 200's and Visual 55's). VED enables two files to be independently displayed on the screen at once, which makes both word-processing and program-development easy. This is the best that can be achieved at present with the terminals that we have available but we are in the process of developing a full windowing mechanism for Poplog implementations on machines with bit-mapped graphics, such as the SUN workstation. It is possible to have more windows on a dumb VDU but they are too small.

The first work that most students do on the system is to call onto the screen a Teach file that, explains how to navigate around files by moving the cursor in any direction in varying sized leaps. This is achieved by typing the command "teach" on the VED command line. This will have the effect of filling up the half of the screen with tutorial text. The cursor will be placed at the beginning of the new file and all subsequent editing commands and keys take effect on that file. Thus the earliest experience that such a student has is really about using the system as a word-processor.

The screen editor works on the principle that what you see is what you get and that anything typed in using the ordinary keys is entered in the file at the cursor position, other text moving along and down to make way for it as the default (though text can replace what is on the screen *iT* one wishes). A file can be traversed in any direction at will in either small or large jumps, including scrolling sideways. There are function keys to delete characters, words and lines and to carry out other editing tasks. A simple "undo" mechanism is available for accidental deletions. A contiguous portion of a file can be marked with a vertical bar that appears down the extreme]eft hand side of the file. This marked range can then be deleted, moved elsewhere, moved to another file, compiled and so on with ease.

At some stage the student will have two files on the terminal screen, the output file (say) and a teach file. The student can jump **the** cursor back and forward between the files in order to work on one or the other. It is a!so possible to arrange for the window on one of the files to grow to fill the whole screen (the other file is still active and can be called back if necessary). Text can be copied from one file to another if needed using the range marking facilities mentioned above. More than two files can be made active at any one time, possibly a teach file, a file containing a student program and the output file, for example. However only two can be shown at once and means are provided for choosing easily which these two should be.

Once the student has become reasonably proficient in navigating around the file system and within files the work on programming in Prolog or PO?-!1 itself can begin. There are a number of library packages to help students build powerful systems quickly as well as to provide examples of standard techniques in artificial intelligence. Thus there are tutorial versions of various expert system types, production rule interpreters, visual scene interpreters, parsing and grammar 'packages and so on. On occasion it has been possible to "bolt" together a number of these library packages to build, relatively easily, a fairly powerful system. An example is a system which can interpret commands for a simulated robot stated in English, undertake the necessary planning, carry out the actions commanded and display the results using simple graphics.

4. Finding the documentation that you need.

Al Environments and the help file finding problem.

AT programming for research purposes is usually exploratory in nature; that is, the activity of programming is used to explore problems which are ill-defined, to clarify these problems, and to explore possible solutions. This is in contrast to the approach traditional in computing, where good practice requires that a problem be well defined, and its solution well understood, before this solution is embodied in a program.

In computing environments designed for this style of use, it has been found particularly useful to provide extensive on-line documentation. This allows the users to explore the on-line environment in the same exploratory fashion in which they use the environment to explore some other area. This applies both to the novice programmer, who may be learning the basic concepts of computer programming, and to the expert programmer who may want to use some facility of the environment whose details are unfamiliar.

Pop:og's on-line documents, collectively referred to as **the** "help files", number some one to one and a half thousand. Each **help file** has a "type", which is one of "teach", "help", "ref" or "doc" - these specify different categories of help files, and a name which specifies the subject of the document. Tor example, the help file **called** "teach define" would be a tutorial guide to defining procedures in POP-11. A name such as this can give only so much information; for example, "teach define" might have been about defining POP-11 procedures, but it might also have been about defining Prolog predicates, defining production rules in a production system, or defining the action associated with some key sequence in VED. Furthermore, if the name of the required file is not known, it is very difficult for an inexperienced used to guess what its name might be, especially for a novice who may not be familiar with computing terminology. Even an expert may have great difficulty with terminology, because taking a technical term out of context can deprive it of most of its meaning; for example a help file called "help delete" might explain how to delete files from a directory, lines from a file, items from a data-structure, or one of many other kinds of deletion.

To some extent, problems with the names of help files can be alleviated by the systematic use of cross-references between help files. These effectively form a menu system embedded in the help file system, which allows the the help file author to produce help files that contain, in addition to ordinary text, menus of other help files. Thus the documentation can be structured so that the user can arrive at the required document by selecting a succession of menu entries, instead of having to know the name of the required document in advance.

However, there are further problems in using the help file system, due to the manner in which the Poplog system was developed. Like other AI environments, Poplog is an extensible system, and the languages that it provides, the environment and the documentation have all "grown" with use, rather than having been carefully planned from the outset (see Khabaza, 1985 for further discussion of this feature). This lack of organisation in its development has led to the documentation being "lumpy", or uneven; some features of the system are well documented, some poorly, and some not documented at all, or documented in surprising places.

The problems associated with not knowing whether a feature exists, whether it is documented, and what the documentation is called are collectively known as the "help file finding" problem.

The Help File Finder project.

This project was conceived as investigating the ways in which AI techniques could help to solve the help file finding problems described above. There are two basic threads in its motivation: one practical, to solve the problem of providing good on-line documentation, and one theoretical, to study the cognitive processes of computer users.

The project has suggested a number of directions for research on improving the Poplog documentation system. The simplest of these is to experiment with more sophisticated keyword systems than that currently provided. The current help file naming scheme can be thought of, roughly, as "one keyword/one file". Some improvements might be gained by allowing many keywords to be associated with a single file, and the keywords associated with different files to overlap. For example, a help file on deleting files might have the keywords "delete file", while a help file on deleting items from a list might have "delete list".

The system of cross-references between help files, mentioned above, leads to a "network-structured" menu system. A major problem with such systems is that when introducing a new node (in Poplog, a new help file), the authors must insert cross-references to the new entry in all the relevant files, many of which may not be known to them. To aid this process, a system for examining the network of help file cross-references is being developed. This will form a tool for help file authors, to help them integrate a new help file into the menu system, or improve the existing menu system.

A third suggestion to arise from the project is that of "dynamic help files". These are help files which, rather than consisting of static text, are generated and transformed by an active system, using some stored framework. This allows the information displayed in a given help file to be tailored to the requirements of the user. The system of dynamic help files would form an "augmented discrimination net"; a discrimination net because the ultimate aim of traversing the net would be to find the required help file, and "augmented", because traversing the arcs of the net triggers some work, such as the creation of a tailored menu.

Finally, the project has been considering the feasibility and problems of an expert system with knowledge and inference capabilities in the domain of programming systems. This endeavour is particularly relevant to the aim of studying the cognitive processes of computer users, because producing such a system requires the modelling of human computing expertise. However, reasoning about the activity of programming is still poorly understood; preliminary study has shown that the amount and variety of knowledge used by a human expert, even in limited areas such as list processing or editing, is very large, comparable with the knowledge bases of the most advanced expert systems.

5. An Intelligent Program Checker

Everyone has their own favourite horror stories about the awful, incomprehensible error messages that novice programmers have to cope with (du Boulay & Matthew, 1984). One of the advantages of a system like Poplog is that it is entirely feasible to develop tools for helping students learn conventional languages like Pascal.

The "Intelligent program checker" project is a move in the direction of incorporating Intelligent Tutoring Systems into Poplog. Because of the limitations of our current understanding of tutoring system design (mentioned earlier), the program checker does not attempt to carry out the whole tutoring task; instead it focusses on mechanisms for reasoning about programs and diagnosing bugs. This is a central component for any tutoring system for programming.

The program checker is intended to provide the novice programmers with the sort of information about errors which they could expect to receive by consulting an expert. By embodying such expertise within the system, the pupils have the advantage of immediate feedback on errors, without the intimidating (and expensive) overhead of having an actual expert programmer looking over their shoulders.

This task is not suited to a standard expert-system approach, since it involves reasoning using many different sorts of knowledge (and metaknowledge), and providing explanations of this reasoning process which are meaningful to the pupil.

At its lowest level, the program checker isolates and identifies syntactic bugs in the program. By using techniques related to natural language processing methodology, the system manages to accurately locate errors and, in most cases, to suggest corrections which are consistent with the intentions of the pupil. This is in direct contrast to traditional compilers which often report errors to be some considerable distance from their actual location, and do not offer corrective advice (or if they do, a least-errors correction is used, which is typically far from the program which the pupil intended).

Other errors which the program checker detects are less easy to separate, but may loosely be regarded as forming a continuum of "semantic errors" of an increasingly problem-dependent nature. The simplest semantic errors involve comparing a formal semantic description of each statement with the way in which that statement has been used by the programmer. Following this, analyses of interactions between statements and of algorithms used by the programmer are performed. When this has been done, the actual behaviour of the program is compared with the stated goals of the program (normally supplied by a human teacher).

At present, only the syntactic level of analysis has been fully implemented. It exists within Poplog and is currently oriented towards teaching Pascal. Since the language infomation is separable from the rest of the system, versions of the system for other languages are easy to produce. POP-11 and C versions are currently being added.

The program checker itself is written in a mixture of POP-11 and Prolog. Essentially, the well-defined algorithms are implemented in POP-11 for speed, while the facilities for reasoning about the state of the parser are implemented in Prolog. Extensive use is made of the applications interface to the VED editor: the pupil is shown the program in one window while maintaining an interactive dialogue with the program debugger in the other window. This permits a much richer debugging interaction (for example, both pupil and program checker can "point" to pieces of the program which they wish to discuss).

6. Understanding Prolog Programmers

Prolog has suddenly become a most fashionable language accompanied by some rather ambitious claims about the ease with which it may be learnt. While these kind of claims are quite usual for any language, Prolog seems to be something of a special case because of its apparent relation to logic and English (Taylor, 1984).

Proponents of Prolog claim that as Prolog is based on first-order predicate logic it is easier to learn than other more conventional programming languages, because this type of logic has developed from the formalisation of human rational thinking, rather than from the logic governing machine operations. Furthermore, the logical basis allows for a declarative interpretation of programs which can be expressed in natural language. These are dubious claims since independently conducted psychological experiments have demonstrated (a) that people have considerable difficulty understanding and using logic and (b) that using natural language within the formal domain of logic leads people to commit various kinds of reasoning errors. In fact there is considerable overlap in the literature on the psychology of deductive reasoning and the Human Factors literature which looks specifically at people trying to learn programming languages. This may be attributed to the fact that in both areas we are look:rig at people trying to work within the constraints of a sometimes counter-intuitive formalism, and they do not find this *ei^he:^* easy or 'natural![1].

We have instigated a project at Sussex, using the version of Prolog in the Poplog system, which aims to gather empirical data, focussing on the following major issues relating to the learnability of Prolog. First, are the declarative semantics really of any use to the beginner, or are they merely a red herring? Second, does the presence of natural language in the programming domain help, or obstruct the learning process by obscuring the underlying formalism? Third, what kinds of models of Prolog do beginners construct for themselves, and how far do they diverge from an adequate model?

In order to put these questions into a broader context we are also examining the behaviour of Prolog experts and experts in other languages (e,g. POP-31 or Pascal) who are learning Prolog. For data collection we have written a program within the Poplog system which automatically logs end stores interactive sessions at the terminal, inside the editor (VED). A second program Is used to replay the logfile by substituting the characters stored in the file for the normal keyboard input to VED. Thus, at replcy time, VED behaves as it did for the experimental subject when the logging took place.

Our data *K:11* be of use to the varied *?ro2og* communities and any software and online teaching materials developed as a result of this work *vcijl* be made available as part of the Poplog system to UK. educational establishments.

7. Conclusions

Poplog has the following nice features. It provides a powerful screen editor integrated carefully into a system containing incremental compilers for Prolog, POP-11 and Lisp. It is used a both a general software development environment as well as a specialist environment for teaching and research in artificial intelligence. Probably its most pleasing characteristic is the way that it enables working prototypes to be quickly built and tested by allowing new sections of code to be both tried out and linked in quickly and easily.

Of course there are still things to be done. There is extensive on-line documentation but its coverage can be patchy and the cross-referenecing could, and is, being improved. The split screen facility is an enormous improvement on systems that allow only a single activity to be displayed at one time, but it is only a step towards a fully-fledged window mechanism. This will realised on the single user workstations with br't -mapped graphics that are now being used extensively. At the time of writing {July, 1983) the Common Lisp implementation is in the final stages of completion. This wil: replace the existing small Lisp implementation in the system. Though even this small system has been used for useful development work by at least one of our customers.

Because we use the Popiog system ourselves for teaching and research, there is a constant drive to provide better features and to worry about how to teach its use to beginners. The research projects that have been described in this paper are just three from a comprehensive programme of research in artificial intelligence and related topics currently uderway in the University of Sussex.

8. Acknowledgements

The Pop!02 system was developed by a team of people including Aaron SIoman, Jon Cunningham, Steve Hardy, John Gibson and Chris Mellish. It has been adopted as part of the SERC/Alvey IKBS Infrastructure and is marketed by System Designers. The "Intelligent Program Checker" project, the study of "Novices Learning Prolog" and the "Help File Finder" project are all supported by grants from SERC/Alvey. The "Help *Tile* Finder" project is directed by Professor Aaron Sloman.

9. References

Barrett R., Ramsay A. & Sloir.an A. POP-:! 1: a practical language for artificial intelligence, Chi Chester: Ellis Horwood, 1985

du Boulay, B & Matthew I. Fatal error in pass zero: how not to confuse novices Behaviour and Information Technology, 3, 2, 109-118 (1984).

Gibson J. POP-11: an A.I. programming language, in New horizons ^n £f*iL£IL£i22JLL computing, edited by Yazdani M. , Chichester: Ellis Ilorwood, 1984.

Hardy S. A new software environment for list-processing and logic programming, in AXLIJMCJ.CLI int.eH,i^erice: tools, techniques and applications, edited by O'Shea T. & Eisenstadt M., New York: Harper and Row, 1984.

Khabaza T. Towards and intelligent help file finder, in Artificial jntelligence programming environments, edited by Hawley R., Chichester: Ellis Horwood, 1985.

Taylor J. Why novices will find Prolog hard, in proceedings of ECAI-84, European conference jji artificial intelligence, Pisa, 1984.