This paper was originally published as:
A. Sloman, ‘The Evolution of Poplog and Pop-11 at Sussex University’ in
POP-11 Comes of Age: The Advancement of an AI Programming Language,
Ed J. A.D.W. Anderson, Ellis Horwood, pp 30-54, 1989.

THE EVOLUTION OF POPLOG AND POP-11 AT SUSSEX UNIVERSITY

Aaron Sloman
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QN

Introduction: influences from Edinburgh

I first met POP2 early in 1972 when the late Max Clowes, now best remembered for the "Huffman/Clowes" line labelling technique, allowed me to borrow his teletype and modem to dial up the Elliot 4130 computer in Edinburgh University. This enabled me to play with some elementary list processing programs.

I later got to know POP2 well when I spent the 1972/3 academic session in the department of Computational Logic in Edinburgh University, as a Science Research Council visiting fellow, learning about AI, at the invitation of Bernard Meltzer, founding editor of the AI Journal, who was then head of the Department of Computational Logic.

I had some wonderful teachers including Bob Boyer, J Moore, the late Julian Davies (who subsequently went to Canada), Jim Stansfield (who later moved to MIT), and Danny Bobrow (who also happened to be visiting for a year between leaving BB&N and moving to Xerx PARC). That was the year work began on the now famous Boyer/Moore string searching algorithm and their Lisp theorem prover. Harry Barrow also helped me learn to drive some graphical utilities he had implemented on a Honeywell computer. He and others in Donald Michie’s department were doing pioneering work on Freddie the robot, using POP2 as the main implementation language for both the image interpretation and the motion planning and control.

My first non-trivial POP2 program simulated a vertical two-dimensional world (called "Eden") in which a totally unintelligent robot (called "Adam") could be given commands to move around, push things, pull things, etc. The program made heavy use of an analogical representation in the form of a 2-D rectangular array. This made rotation difficult!

During that time I learnt LISP, POPLER (Julian Davies’ implementation of Carl Hewitt’s Planner system, a pre-cursor of Prolog), and LOGO (implemented in Pop-2 by Danny Bobrow while visiting Edinburgh). I also had a taste of Prolog, as Bob Kowalski was then developing his ideas about Prolog as a language with both a procedural and a declarative interpretation. David Warren had arrived as a research student and kept trying to persuade us all that Algol-68 was the best language ever. Something must have happened to him after I left.

Later, around 1975 when I was back at Sussex, I obtained a research grant for a vision project, and that gave me access for several years via a 1200 bd phone line to a DEC-10 computer in Edinburgh University. For a while we used the POP-10 system implemented by Julian Davies, but later moved onto the "WonderPop" system (known as WPOP) implemented by Robert Rae and Allan Ramsay at Edinburgh on the DEC-10. They were employed on a Science Research Council project to provide software support for AI researchers in the UK who had been given access to the DEC-10. This included people at Sussex University and Imperial College.

WPOP introduced a process mechanism, designed in part by Dave McQueen (now at Bell Labs), and various other syntactic and semantic extensions to POP2, including typed arithmetic identifiers and procedures. This made some compile time optimisation possible, achieving speeds comparable with PASCAL. The -conskey- procedure was introduced for creating new datatypes. WPOP also used a "caged" store manager, with different data-types stored in different parts of the address space, making it unnecessary for each record to have a field pointing to the key.

My impression is that the caged store manager caused more trouble than it was worth - e.g. users had to estimate cage sizes required for different classes of data, and garbage collection bugs proved very difficult to track down. Moreover the saving in space in individual records could be outweighed by unused portions of cages.

In the late 1970s John Gibson, who was later to play a major role in this story, was a research student at Sussex and he also made heavy use of WPOP on the DEC-10, for his work on natural language processing, later abandoned when he decided to concentrate on the development of Pop-11 and Poplog. He and I spent many hours testing new facilities in WPOP and communicating with Robert Rae about them via electronic mail.

As I was then chairman of the SRC Special Interest Group for AI, as well as being one of the major users of WPOP, I was able to influence some aspects of the Edinburgh design work, including the richer looping syntax. However I failed to convince others that provision of interactive development tools (including an efficient integrated editor) were at least as important as language extensions. The argument against me was that the limited address space on the DEC-10 was too limited to be cluttered up with things that were not required for the actual running of AI programs. The opposite view was taken at Sussex, as shown by the development of Poplog, sketched below.

At that time Frank O'Gorman was working as a research fellow with Max Clowes on a vision project at Sussex, and because they were using Algol68, and finding it very restrictive, Frank implemented a version of POP2 that ran on the ICL 1906 computer at the Rutherford Laboratory. This was, I believe, the first version of POP that included hashed property tables, subsequently taken on by WPOP and Pop-11. It included a compiler written in POP2. However, the days of the 1906 were numbered and that version of Pop-2 was not used much.

POP developments at Sussex

When I returned to Sussex University in October 1973, a group of us, including Max Clowes, by then Professor of Artificial Intelligence (probably the first in the world with that title), along with Margaret Boden, Alistair Chalmers, and several other colleagues started up what we called the "Cognitive Studies Programme". This was originally based in the *School of Social Sciences*, but very much later it was to evolve into what is now a separate *School of Cognitive and Computing Sciences*, including a range of undergraduate and postgraduate studies in Artificial Intelligence, Computer Science, Linguistics, Philosophy and Psychology. Our teaching and research requirements stimulated a varietyof developments of POP2 that led to what is now known as Pop-11, the core language of the Poplog multi-language system.

From the beginning we felt it was important to teach AI by giving students plenty of practical experience designing and implementing AI systems, so that they could learn at first hand some of the problems of analysing complex tasks and making systems that can emulate even very restricted aspects of human behaviour.

The most powerful language available on the Sussex Computing Centre machine at that time was Algol-68, and for a while we tried using it for teaching. However it was totally unsuitable for our students, most of whom were complete beginners and quite unused to mathematical formalisms or the kind of rigorous thinking required even to get a small Algol-68 program going.

We eventually managed to obtain a version of POP2 for ICL machines from Conversational Software Ltd (now defunct) and mounted it on the Sussex university computing service ICL 1904 computer. The machine was not available for interactive use, so I sat for hours punching cards to transform the POP2 system for use in batch mode: essentially getting the error handler to try to ignore everything up to the end of the current procedure definition after each error. I also punched in various library programs including the first POP turtle, a program that drove a simulated turtle around a 2-D array leaving a trail of "paint" that could then later be printed out. This provided a simple graphics capability that could be used on dumb terminals. It also allowed the students to draw pictures in an array and then write image interpretation programs that analysed the pictures (unlike the LOGO turtle, which produced images on a screen, or sheet of paper, but did not give the computer the ability to "see" them.

For several years drawing and analysing pictures in a 2-D binary array was how we introduced programming and AI to new students. Later we started using list-processing tasks (solving river-crossing puzzles or analysing and generating sentences) to start students off. I am still uncertain which is better.

Fortunately our students did not have to use punched cards: they had access to a primitive editor on a Modular One computer, which shipped programs over to the ICL machine. Turn around could be anything from a few minutes to a whole day, or longer!

This was intolerable, so we obtained funds to purchase our own computer. However, we had to choose a machine, and a programming language.

We did not want students to have to fight with impoverished languages or restricted programming environments, but our resources were very limited. At that time, typical introductory computing courses elsewhere either simply let students play with games or packages, or if they introduced programming they often used numerical programming problems, and impoverished or unfriendly languages like Basic or Pascal. LOGO was an attempt to break away from some of the restrictions of such languages, and for a time we seriously considered buying an Interdata computer and using the LOGO system developed by George Coulouris and colleagues at Queen Mary College London.

In order to help me evaluate LOGO, Danny Bobrow arranged for me to have access to the MAXC computer at Xerox PARC (a micro-coded DEC-10 lookalike), Max Clowes allowed me to use his teletype and modem, and University College London accepted me as a user of their experimental transatlantic network service. I spent many hours sitting in Brighton typing in characters that went via a long round-about route to California and then were echoed back. What a way to use communications satellites!

These tests enabled me to decide that LOGO was really too impoverished for our purposes. The MAXC version did not even support list processing, so I had to implement a horrible list-processing package where every list link was represented by a LOGO "sentence" made of two LOGO "words". The concept of a "sentence" in LOGO was very ill defined, and it was not at all clear what the difference between a word and a one-word sentence was supposed to be, especially as the same double quote symbol ‘"’ was used for both words and sentences.

Moreover, the attempt to reduce the syntactic complexity by eliminating parentheses and commas had produced a language in which all but the simplest commands were unreadable! (For this reason I was later very surprised to learn that LOGO was being used to teach AI to undergraduates in Edinburgh University, especially when they asked me to be their external examiner. However, it turned out that they had little choice because there was no suitable AI language available on the machine their undergraduates had to use.)

Having rejected LOGO at Sussex we still wanted a teaching language that was powerful enough for us to use for our own research, so that the products of our more advanced work could be made available as library programs for novices to use as building blocks or demonstration programs. For our own sanity we did not wish to use one language for teaching and another for research. We also hoped that we could use a language that was suitable for absolute beginners but allowed a gradual transition to more and more sophisticated program development during the course of a student’s degree.

Since neither Max Clowes nor I had computer science qualifications or adequate knowledge and experience, Steve Hardy, who joined us in October 1974, was given the task of selecting a machine and providing the programming tools meeting our criteria. He had previously been a research student at Essex University, supervised by Mike Brady. He had met POP2 on the DEC-10 at Essex University, where he used it to implement sophisticated tools on which he built his PhD project on synthesising Lisp procedures from input/output examples, so he had a deep appreciation of its power. He also knew Lisp well, having built an interpreter for it as part of his post-graduate studies, as well as using it as the object of study in his thesis.

Steve could have used Lisp as the implementation language for his thesis work but he preferred POP2. One reason, I think, apart from the greater readability of POP2, was that POP2 provided a useful "state saving" mechanism that allowed him to write routines that attempted to solve a problem and then, if things got difficult, suspended themselves and only resumed working later on if new supporting evidence was turned up by other modules. These "cue-seeking" routines communicated via a global data-base using pattern-directed procedure invocation. Later this kind of mechanism, which I suspect was independently invented in a number of places, came to be known as a "blackboard", using terminology adopted at Carnegie Melon University in their speech understanding project.

After coming to Sussex, Steve investigated various computers (including the British machine called the Modular One). Eventually we decided to buy a PDP11/40. But we still lacked a teaching language.

We considered Lisp but decided its syntax was too impoverished for non-mathematical beginners. I still believe that a rich redundant syntax, although it requires more rules to be learnt, reduces the cognitive load on the user, as well as enabling the compiler to give more useful error messages if you make a mistake. Unfortunately, many designers of computer languages think only about more formal properties of the language, such as the parseability by computer, having a well defined semantics, and so on. Computer Scientists are not (or used not to be) concerned about cognitive processes in users of computer languages.

Anyhow, we did not find a Lisp that was available for any of the machines we had considered.

There were other serious objections to Lisp as a teaching language. For example it had no boolean data-type, and using the empty list to represent false was very confusing for beginners. Worse, many Lisp systems did not produce an error when CAR or CDR was applied to NIL, but simply returned NIL, making detection of certain kinds of program design errors difficult. We also felt that the use of two kinds of values for variables, the function value and the non-function value, was confusing and made it hard to treat functions in the same way as other objects, e.g. locally redefining them within another function. Moreover, the problem of forming functional closures in Lisp had been solved by clumsy and inefficient mechanism (funargs) whereas POP2 had an elegant and efficient, though less general mechanism, partial application.

Some of the defects that made us reject Lisp were subsequently remedied in the definition of Scheme, and its descendant T, though not, alas, in Common Lisp.

We thought about Prolog, but felt that it was too specialised and would give beginners a distorted view of computation, in addition to presenting them with a number of difficult conceptual problems. E.g. why can’t you have a rule saying that X is the brother of Y if X is male and Y is the brother of X? In logic there is nothing wrong with this. In most Prolog implementations it leads to infinite loops, as do other fairly natural constructs.

Eventually, after much discussion and heart-searching, Steve, Max and I decided that POP2 had many advantages over all the alternative languages that were available, both for beginners and for their teachers. It also had some flaws, including the fact that the open stack, while supporting a number of elegant constructs, could lead to obscure bugs. But we felt that the advantages easily outweighed the disadvantages.

The birth of Pop-11

Having chosen our machine and our language, we tried to get hold of a POP2 for the PDP11 computer. Various implementation projects were unearthed, including one in India, but they all had unpredictable finishing dates, so Steve sat down in the Summer of 1975 and by next January we were able to start using his POP system for teaching, under the RSX-11 operating system. He called it Pop-11 because Julian Davies had called his DEC-10 system POP-10.

The first Pop-11 was written entirely in MACRO-11, the PDP11 assembly language. It compiled Pop-11 source to an intermediate language, which was then interpreted. Despite the comparative slowness of an interpreter, and the small size of the machine, this system, and later the Unix version served us very well for several years.

Unfortunately the operating system provided with the machine (RSX11-D) did not provide proper time sharing facilities, so Steve had to implement those too.

Because the PDP11 had such a small address space (32K 16 bit words) we had to do without several of the facilities of POP2, especially sections, user-definable data-types, state-saving and jumpout. But most other features of POP2 were there, including (slow) floating point arithmetic, newarray (though not newanyarray), updaters, macros and partial application, though not dynamic lists. Access to the operating system was via a vector containing special procedures and other facilities.

Although Steve did most of the design and implementation there were continuous discussions of required features. Besides leaving out some POP2 features to save space, we took the opportunity to introduce various improvements to the language. Our objections to Lisp’s use of NIL for false was matched by our objections to POP2’s use of the integer 0 for false, so we introduced a boolean data-type, though we continued to allow any non-false entity to be equivalent to true in conditionals and loop termination tests. (Perhaps that was a mistake). We also introduced an "undef" data type, to be the default value of uninitialised variables, instead of following the POP2 use of a pair containing the word "undef" to indicate an undefined value. This new data-type allowed more errors due to uninitialised variables to be detected immediately, and helped considerably with debugging.

Although sections were not provided, a library package provided a "prefix" mechanism for automatically attaching a prefix to certain identifiers so that they would not clash with other uses of those identifiers. This remains available as a Pop-11 library package in Poplog.

Having seen how useful a pattern matcher was in his own research, Steve easily convinced us that it was essential to extend the language to include a built-in pattern matcher with segment variables, together with extended list syntax using "ˆ" and "ˆˆ" for inserting the value of a variable into a list or splicing in a list of items. He wrote the pattern matcher, like everything else, in assembler. This allowed many list processing programs to be much clearer: a pattern could often be used to *show* what was intended, instead of an obscure looping procedure that would take much longer to write and debug. For example the following pattern would match a list of lists containing a list containing ITEM1 and ITEM2 and would assign a (possibly empty) list of all the intervening elements to the variable FOUND (the symbol "==" will match any number of items):

```
    [== [== ˆitem1 ??found ˆitem2 ==] ==]
```

So if the value of "item1" is "e" and the value of "item2" is "i" then the above pattern could be matched against the list

```
    [[a b c] [d e f g h i j] [k l m n] [o p q]]
```

and would bind the list [f g h] to the variable "found".

Writing a program to do this in Lisp or Pascal would require at least three loops and a tricky control structure, and would be hard to get right first time, and probably even harder to understand when read. A Prolog version would also be less clear than the Pop-11 version because the Prolog matcher (i.e. the unifier) does not support segment variables except for the final segment of a list. Using "==" and "??variable" to match arbitrary list segments simplifies many programs.

The use of a pattern matcher made possible a library package providing a simple pattern-directed database mechanism, and this proved to be a useful core for a variety of library programs and student projects concerned with interesting AI tasks, as well as more general introductions to computing techniques.

Another important extension introduced by Steve Hardy was the provision of numerical indexing of lists and other data-types, so that the expression "item(5)" could be used instead of "subscrv(5, item)" or "subscrs(5,item)" to access or update the 5th element of a vector or string and instead of

```
hd(tl(tl(tl(tl(item)))))
```

or something like Lisp’s "CADDDDR", for accessing or updating the fifth element of a list.

This was the pre-cursor of the "class_apply" facility introduced in Poplog Pop-11, which allows any object of any type to be treated as if it were a procedure. Occasionally this use of a data-structure as a procedure causes problems, but it permits the construction of elegant general purpose utility programs that access or update the N’th element of a structure, one of many ways in which a language like Pop-11 that lacks typed variables, but has typed objects, can facilitate re-use of code.

Other generalised (overloaded) facilities for a range of data types were introduced, including explode, which put all the contents of a data-structure on a stack, and fill which "filled" a data-structure with objects from the stack.

The pattern matcher and numerical indexing enabled students to learn to use list processing in quite complex programs without having to learn that lists were really binary trees: an irrelevant implementation detail for many purposes.

Several other extensions to Pop-11, some added after the transfer to the VAX 1981, allowed programs to be more general. For example, the original Pop-11 list concatenator, the infix operator "<>", was generalised (by John Gibson I think) as a concatenator for strings, words, vectors, and even procedures. So if proc1 and proc2 are procedures the expression

```
proc1 <> proc2
```

denotes the procedure that first applies proc1 and then applies proc2 (each of which will take whatever arguments it requires from the Pop-11 stack, and put back any results).

The extension of <> to a range of vector-like data-types, like the numerical indexing of structures, makes it possible to write generic programs that can be re-used in a variety of contexts. This generalises the fact that most programming languages allow some "overloaded" procedures, such as the equality operator and the printing function. The new Pop-11 features introduced later in connection with the class mechanism, also extended this "overloading" capability in several directions. The overloaded procedures all use the data-types of their arguments to select appropriate procedures to run. The benefit is that it is possible to define generic procedures that can be used on a variety of data-types, facilitating re-use of well tried and tested software.

On the other hand the numerical indexing could sometimes lead to obscure errors. There’s a difficult trade-off between language features that increase the chances that you will get your program right first time because high level general facilities are available, and features that make it easier for the system to help you detect mistakes, for instance by doing more compile time checking. Strongly typed languages select the second option. We preferred the first.

Another language extension in Pop-11 was the ‘chain’ facility which allowed the implementation of exitto, exitfrom, the catch and throw pair, and other useful control structures for dealing with abnormal exits from procedures. This made it unnecessary to provide the jumpout facility of POP2. However, these control facilities were potential sources of obscure programming problems, as were the process mechanisms introduced later, until John Gibson later introduced "dynamic local expressions" (with the keyword "dlocal") into Poplog Pop-11, making it possible for a procedure always to trap abnormal exits so that everything that had to be tidied up when a procedure finished, was tidied up. Similarly a procedure that could be resumed and suspended as part of a process was also able to ensure that environments were saved and restored appropriately, using dlocal.

The problem of limited address space on the PDP11 was partially solved by providing an auto-loading library mechanism that allowed us to add a host of useful utilities to Pop-11 without requiring them to take up some of the precious address space for users who did not want them. We also added more and more AI demonstration programs that students could either run and play with, or incorporate as subroutines in their own programs, or copy and extend. This helped to make Pop-11 very popular for teaching AI.

Another change from POP2 allowed macros to return results on the stack, which were concatenated onto proglist, the compiler input stream, instead of having to make a list of new program text items to give to the macresults procedure. This change made it much easier to trace and debug macros, as well as making macro definitions easier to understand. Pop-11 also made it possible to define a macro as a list of text, or a single text item.

I think the main contribution of Max Clowes at this stage was the invention of highly motivating exercises and examples, and a style of teaching that always encouraged students to look beyond the technical niceties to assess their relevance to understanding the nature of human intelligence. It is very easy to forget about these long term objectives in teaching programming, especially when some of the students enjoy learning about programming tricks.

Before we switched to Unix, there were no text-formatting facilities on the PDP11, so while the students were writing their noddy Pop-11 programs, Steve Hardy and I (mostly Steve) implemented a quite sophisticated formatter in Pop-11, which we used for producing our teaching documentation. We had very limited laboratory funds and I can still remember how we used to save paper (and trees) by printing teaching documentation on the back of old line-printer output.

Ever since then we have continued to use Pop-11 for a variety of non-AI purposes. For example one of our research students built a four-voice electronic tone generator, and John Gibson wrote Pop-11 programs to take in a readable notation and compile it into instructions to drive the generator. I wrote a Pop-11 program to help with production of indexes for books, and it has since been used by several authors at Sussex ever since. Roger Evans, much later, used Pop-11 to write a TROFF previewer on Sun workstations. Pop-11 programs, combined with C programs or shell scripts, have been in use by the Poplog development team for many years for configuration control and distribution of library files across a mixed network of computers. And so on. The full potential of Pop-11 as a general purpose programming language for non-AI tasks has yet to be appreciated by the world at large!

The move to Unix on the PDP11

We were very pleased that we (and our students) no longer had to use the ICL machine, but RSX-11 gave us many headaches, and the machine crashed on the slightest provocation. We then heard from George Coulouris that the Unix(tm) operating system was far more flexible and more robust (if a disk access failed it tried again instead of crashing).

So with the help of DEC UK we switched to Unix in 1976, and in about 6 weeks Steve re-wrote his system in Unix assembler. Unix made a huge difference to the development environment. Previously we had used a very primitive editor written in Pop-11, but now we were able to suspend Pop-11, invoke a more sophisticated editor, then resume Pop-11 and compile the file that had been changed. At first we used the unix 'ed' then later a screen editor. For our naive users the 'ed' error messages (usually only "?"), and the bizarre pattern elements for searching (e.g. "." would match any character), were very confusing, so we had to produce a modified version more suitble for ordinary mortals.

I remember a DEC maintenance engineer being amazed by the number of students who were able to use our PDP11 simultaneously, and get a reasonable response. This was mainly because the use of an incremental compiler substantially reduced the number of different programs that had to be active at once, and reduced the amount of context-switching between editor, compiler, linker, user program and debugger. I think we eventually got up to about 20 simultaneous users on a PDP11/40 with 256 Kbytes main memory and 15 Mbytes disk space. Users mostly switched between Pop-11 and ed.

Unix pipes also allowed us to write programs that overcame the address space limitation by having several Pop-11 programs running in parallel and sending messages to each other. This was not popular with other users of the machine!

One use of Unix facilities was a built in Pop-11 "help" macro that invoked the Unix formatter nroff to format and print out a help file. I later introduced a "teach" program written in Pop-11 that allowed users to print out a portion of a text file, try out some Pop-11 commands, print more of the text file, try more commands, etc. These later evolved into the Poplog TEACH files.

As Unix was spreading on PDP11s and more and more other universities wanted to teach AI, we started distributing Pop-11 on RK05 disks, for a nominal charge. I don’t recall how many sites obtained it from us, but centres of expertise developed in a number of universities including Nottingham, Aberdeen, Warwick, Queen Mary College and Sheffield, as well as some places outside the UK. It was also successfully used at Marlborough College, a "public" (i.e. private!) school for boys, as an alternative to BASIC.

Further developments in the 70s

At Sussex, Pop-11 started spreading into other Schools. Steve Isard (now in Edinburgh) used Pop-11 for teaching programming and AI to Experimental Psychology students in the School of Biological Sciences, and Jim Hunter (who later moved to Aberdeen University) introduced Pop-11 for teaching and research in the School of Engineering and Applied Sciences. It was Steve Isard who persuaded us to drop the "function .... end" format for defining procedures, in favour of "define ... end".

Around 1978 Jim Hunter, Keith Baker (now Professor of Computer Science at Reading) and I set up a project to develop a distributed Pop-11 system. Allan Ramsay and David Owen were employed as research fellows on this project and a simulated version ran (slowly) on a PDP11/34, but during the time available for the project we never managed to get the Cambridge Ring system to work so that we could distribute the Pop-11 programs over the network of LSI-11 microcomputers, as planned. However, the project did produce potentially useful programs for specifying (and then generating) a network of Pop-11 processes, the processors to which they should be allocated, the communication channels to be used, etc.. For instance, it was possible to assign more than one Pop-11 process to a particular processor, in which case they would be time-shared, but each process was defined so that all it needed to know were logical names for its communication channels, whether pipes or real communication ports between machines. This made it possible to reconfigure a network easily (though not while running). This was one of many examples of the potential uses of Pop-11 for non-AI tasks.

For teaching purposes, we obtained the PDP11 Prolog that Chris Mellish had implemented while in Edinburgh, and Steve Hardy wrote a Lisp system in C, so by around 1980 we were able to teach Pop-11, Lisp and Prolog to our students, all on the PDP11/40.

By then Bill Clocksin had implemented a POP2 system for the PDP11/40 in Edinburgh University, and for a while it was used for teaching there. It had more of the DEC-10 POP2 facilities than Sussex Pop-11, including user-extendable record and vector classes, so it was well suited for anyone wanting compatibility with older POP2 systems. But it lacked some of the extensions we thought essential for our purposes, such as the built-in pattern matcher, and the boolean and undef data-types. So we never used it.

Jon Cunningham, who had been a reserch student at Essex University, joined us as an AI lecturer in October 1980. He had started his PhD work using LISP to implement a program for checking consistency of naive physics axioms, but was quickly converted to Pop-11 after coming to Sussex, and then he played an important part in our design discussions, and subsequently implemented a number of extensions to Pop-11, and some useful library programs and teaching programs. He later implemented a program that translated LISP into Pop-11, which was useful for teaching Lisp to students who had learnt Pop-11. This is still available as the (unsupported) Poplog library program LIB OLDLISP.

Later he was to implement the first Poplog LISP compiler, a subset of Maclisp, subsequently replaced by Poplog Common Lisp.

Chris Mellish joined us in 1981 and also played an important role in Poplog development, especially when he wrote the first Prolog system in Pop-11.

The birth of Poplog Pop-11

In 1981 the Sussex University Computing Centre replaced its ICL machine with a group of VAX11/780 computers, at last providing a good interactive service, and the opportunity to develop programs with a big address space (though each machine initially had only 2.5Mbytes memory).

By then our courses had grown and the PDP11/40 was unable to cope, so we desperately needed to transfer our teaching to the bigger machines. We hired John Gibson in the summer of 1981, and by a tortuous route he managed, at amazing speed, to re-implement Pop-11 on the VAX running VMS, in time for us to start teaching in October. The bootstrapping process made use of Steve Hardy’s PDP11 Pop-11 written in Unix assembler, another Pop-11 he wrote in C, and finally John Gibson’s Pop-11 written in Pop-11 to run on the VAX. Programs to translate the new Pop-11 system sources into VAX VMS assembler were written in Pop-11, so a Pop-11 system was required to run these programs in order to do the translation. The output files could then be assembled, linked and run. The full story of that extraordinary bootstrapping operation should be told in print some day.

All this would not have been possible without the co-operation of the Experimental Psychology laboratory, which allowed us to use their VAX for some of the development work before the new machines were available in the Computing Centre.

The VAX Pop-11 compiler was not completed until a few hours before our first batch of students came into the terminal room and attempted to compile and run our "Eliza" demonstration program. Being a first draft compiler it was very slow, and with so many students all simultaneously asking it compile a lengthy file we almost ground the VAX to a halt. But it worked, and, once compiled, the Eliza program ran at a reasonable speed, using a new version of the pattern matcher written in Pop-11.

Both compilation and execution speeds have been enormously increased since then.

At that time we decided to enrich the syntax of Pop-11 with a wide range of looping constructs, and also decided that all syntax closers should be formed by prefixing "end" to the opener. Thus having played with forms like "while ... do ... enddo" we eventually settled for "while..do...endwhile", "define....enddefine" etc. However, there were still plenty of POP2 users that we wished to convert, so some of the POP2 (and WPOP) syntax was still accommodated, including "function.....end" and the use of "close" to terminate loops and conditionals. Alas these continue to bug some Pop-11 users, for instance because "end" and "close" are syntax words.

There were two very important changes introduced in VAX Pop-11. The first was that the language no longer used an interpreted intermediate language. Instead procedures were compiled all the way to machine code. Secondly the new VAX Pop-11 system was mostly written in Pop-11 plus a few assembler files. This meant that from now on development work was going to be much easier as many extensions could be tried out interactively in a running Pop-11 system and then later built in to the system. This enormously speeded up subsequent development, and allowed many extensions to the system to be thoroughly tested before they were added.

The editor VED was such an extension. Steve Hardy wrote a first draft in Pop-11 in about three weeks, around August 1981, testing it on the PDP11 as the VAX Pop-11 was not yet ready. I then took VED over, moved it onto the VAX and gave it the ability to handle more than one window on the screen. Later this version was built into the Pop-11 system so that users did not have to recompile it each time. This split-screen version of VED was meant to be a temporary patch until we had a proper multi-window editor (as in the Poplog LIB WINDOWS library), but somehow the temporary patch remained the standard VED interface until the Poplog window manager (PWM) arrived in 1986. Even now many people still use it. (It should be replaced by a multi window interface on the X Windows system.)

Released from the 32Kword address space limitation John Gibson was able to restore many of the features of POP2 and WPOP, including dynamic lists, user definable record and vector classes, hashed properties, newanyarray, saved images, "lightweight processes", sections and other features, some of them described below. Many of these facilities were generalised in Pop-11, and some of the generalisations are described below.

The Pop-11 process mechanism provided the main process facilities that were in WPOP on the DEC-10, such as procedures to create, run, suspend, resume, or kill a process. However additional mechanisms extended the power of Pop-11 processes, including consprocto, a facility for creating a new process from part of the current calling chain, e.g. for state-saving programs. Dynamic local expressions also enhanced the process mechanism, as mentioned above, and discussed further below.

It was also decided to replace the old operating system interface, which in POP2 had used a special function called "popmess" and in PDP-11 Pop11 had been a vector of special facilities. Instead, we started using a collection of procedures, such as sysopen, sysread, syswrite, sysspawn, syssleep, and so on. This required more space, but was far more convenient.

He was also able to add several novel mechanisms of which perhaps one of the most important was a collection of procedures made available to users, for planting instructions for the Pop-11 virtual machine, which could then be compiled to machine code and run. (I think the original idea for doing this kind of thing in Pop-11 came via Steve Hardy, from a language called "CLU", about which I know nothing.)

These code-planting procedures enabled users to define new syntax words that extended the language as required for different applications. They were also later used to provide incremental compilers for other languages, first Prolog, then a teaching Lisp, then Common Lisp, then ML. Users elsewhere have implemented other languages in Poplog.

The code-planting procedures gave more power than the old macro facility, since macros could be used only to define new constructs that were translatable into legal Pop-11. The same limitation applies to Lisp macros: using macros in Lisp, you can define only new constructs that are translatable into Lisp. By contrast, with direct access to the Pop-11 virtual machine, users could define any syntactic form that could be translated into virtual machine instructions, allowing a richer set of possible language extensions than macros do.

Without that power it would have been hard to implement incremental compilers for Lisp, Prolog, ML, and other languages, although slower interpreted versions would have been possible.

However, the initial set of Pop-11 virtual machine instructions did not prove adequate for all applications, so over the years they had to be extended into what is now known as the Poplog virtual machine, which supports special facilities for Prolog, provides full lexical scoping, and other features that were not in POP2, WPOP, or early versions of Pop-11.

While language development went on, we were also able to extend the teaching and programming environment. This included provision of user-extendable search lists for program libraries and documentation libraries. We also extended the editor so that it could be a general purpose interface both for browsing code and documentation and for interaction with programs: the editor buffer was a data structure that both users and programs could write into and read from. This made it a suitable interface for a variety of tools including simple graphics tools, an electronic mail front end, a text formatter, and others.

As new languages were added to Poplog it became necessary to tailor the environment, including the editor, so that it could provide equal support for all languages, and switch its defaults automatically depending on which language was currently in use. This raised a number of deep problems that have not yet been fully resolved, including the problem of coping elegantly with a file that includes commands in different languages: e.g. it can be a problem for the editor to decide automatically precisely which language is the "current" one. E.g. if you ask for help file while editing a prolog file with the cursor currently in a Pop-11 sub-section, should the help file be retrieved from the Pop-11 area or the prolog area? Of course, any automatic inference system will get it wrong sometimes, so the user has to provide the answer.

The addition of Prolog

During 1982, Chris Mellish and Steve Hardy devised a model for implementing Prolog, using Pop-11 closures to represent Prolog continuations, and Chris implemented a Prolog in Pop-11, while he was learning Pop-11. It worked, but was somewhat slow compared with high performance Prolog systems.

One reason for comparative slowness was that we decided that it was particularly useful to enable Prolog to share data-structures with Pop-11. This meant that it was not always possible to infer that because a Prolog program could no longer access some structures they were inaccessible. They might still be accessible if they had been handed to a Pop-11 program and stored somewhere for later use. This meant that structures that could be allocated using a stack in a stand-alone Prolog, had to be on a garbage collectable heap in Poplog. The use of garbage collections could slow things down, though the more memory was available the less this mattered, since having more memory reduced the frequency of garbage collections.

There were other inefficiencies in the original implementation, which led John Gibson to extend the Pop-11 virtual machine to provide additional mechanisms specifically to support Prolog. For example, instead of Prolog continuations being Pop-11 closures allocated on the heap (and therefore requiring garbage collections), they were allocated on a special stack reserved for Prolog continuations. Additional changes were made to speed up back-tracking and unification.

It was as a result of such changes for Prolog that we started referring to the Poplog virtual machine rather than the Pop-11 virtual machine.

Since the basic facilities are available as Poplog virtual machine instructions and calls to built in Pop-11 procedures, it is possible in principle to implement an extension to Pop-11 that acts as a dialect of Prolog, using a totally different syntax.

We therefore discussed at length whether Pop-11 should be extended to include a logic programming subset, but I argued that most users would prefer their Prolog programs to be compatible with other versions of Prolog. So we left Prolog as a quite distinct language, while defining procedures for calling Pop-11 from Prolog and vice versa.

Steve Hardy did implement a simplified version of Prolog using Pop-11 syntax and the Pop-11 database mechanism and the pattern matcher syntax. It is still available as LIB SUPER (for "superdatabase") in the Poplog library.

Another important problem was whether to leave Pop-11 and Prolog sharing data-structures given the inevitable efficiency cost. We consulted a variety of users and the verdict was clear: people wanted the flexibility of shared data-structures more than they wanted increased efficiency. This choice between flexibility and efficiency has been and always will be a difficult tradeoff in the development of high level languages.

The addition of Common Lisp, and its implications

During 1983 it became clear that Common Lisp was going to be some kind of international standard. Moreover the firm that was responsible for commercial marketing of Poplog (then called Systems Designers Ltd, later renamed as SD, and later still SD-Scicon), decided that the small Lisp system produced by Jon Cunningham was not adequate for the market place.

So we arranged to implement a Common Lisp in Poplog under the supervision of Jon Cunningham, partially supported at first by a grant from Systems Designers and later by a research council grant. One of our recent graduates John Williams was appointed to work with Jon, and after some initial help from Jon, gradually took over the main design and implementation task himself, though some aspects of Common Lisp required John Gibson to make changes deep in the system, such as the introduction of new categories of numbers (indefinite precision integers, ratios, complex numbers), and above all extensions to the virtual machine to support lexically scoped identifiers, non-local gotos and the "unwind-protect" mechanism.

Some additional extensions to the Poplog virtual machine were needed because Lisp did not have a boolean data-type and treated the empty list, NIL, as false, and also because of the use of Poplog's open stack for passing arguments and results.

The addition of full arithmetic facilities meant that any other language implemented in Poplog could also use them. For example, Poplog Prolog immediately acquired indefinite precision arithmetic, ratios and complex numbers. Similarly when ML was added.

Useful extensions to the Poplog virtual machine made for the benefit of Common Lisp, generalised in some cases (e.g. provision of file-local lexical variables), were made conveniently accessible to Pop-11 programmers by extending the Pop-11 syntax. For instance new syntax was introduced for declaring variables as lexically scoped (lvars, lconstant, dlvars).

Lexical (static) scoping led to an important evolution of Pop-11 programming style. In particular, by using it in the Poplog system sources we were able to reduce the frequency of bugs. Over the last few years most of the Pop-11 libraries were also transformed to use lexical scoping, or

    Lconstant'ed and lvars'ed

as the Revision notes often put it.

Lexical scoping did not make quite as big a difference to Pop-11 as to Lisp, since Pop-11 had always included partial application, which provided a subset of the facilities of lexical closures in a more efficient and compact form. However, partial application is not as general, and sometimes it can be more obscure, since in order to make a nested procedure access the syntactically enclosing value of x lexical scoping allows you to write:

```
   procedure(); ..... x .....endprocedure
```

whereas using partial application to do the same thing you have to write something like:

```
   procedure(x); ..... x .....endprocedure(%x%)
```

Moreover, whereas a lexical closure can directly update the enclosing lexical environment the partially applied procedure cannot, unless a reference is explicitly created to hand down to the sub-procedure as the value of x.

One unfortunate consequence of following Common Lisp was that whereas previously dividing two integers always produced either an integer or a decimal number (e.g. 10/3 produced a decimal 3.33333) integer division could now produce ratios. Although for some purposes this was an improvement, because they had absolute accuracy, they could also slow programs down: a trap for programmers who are used to integers being coerced to reals on division. I now think a new distinct operator should have been introduced for the production of ratios. But at least Pop-11 is no worse than Common Lisp in this respect.

Dynamic local expressions

A very important generalisation of Lisp's unwind-protect mechanism provided 'dynamic local expressions', which specify actions to be performed on entry or exit to a procedure, including abnormal entry (such as resuming a suspended process) and abnormal exit (using chain, exitfrom, or procedure suspension. The syntactic word "dlocal" was added to Pop-11 to indicate the use of dynamic local expressions. When combined with the use of procedures that have updaters this allows elegant constructs for automatically saving and restoring the contents of data-structures, or performing other actions. For example, whereas in a procedure definition

```
    dlocal foo, baz;
```

simply specified that the values of the variables foo and baz should be saved on entry to the procedure and restored on exit, the declaration

```
    dlocal % hd(list) %;
```

specified that on entry to the procedure (or re-entry in a process) the expression ‘hd(list)’ should be evaluated and the result stored, and on exit (normal or abnormal) the expression should be evaluated ‘in update mode’, i.e. as if ‘-> hd(list)’ had been written. This would ensure that whatever was done to the head of the list by the procedure, the original value would always be restored on exit.

In addition, it was made possible to specify different actions depending whether the procedure was being entered normally, left normally, left abnormally, resumed in a process, or suspended in a process. This enabled the use of processes to be cleaner and more modular, reducing the frequency of bugs caused by unexpected interactions between environments, and enabling process resumption and suspension to be traced easily.

The introduction of "active variables" that could store multiple values, and ran procedures whenever they were accessed or updated, integrated well with this dlocal mechanism.

This work on dynamic local expressions led to a clarification of the distinction between lexically and dynamically scoped identifiers. John Gibson pointed out that the use of shallow binding for dynamic variables essentially meant that these should best be thought of as "permanent" variables: they always point to the same memory location, though the contents of that location could, if required, be saved and restored on procedure entry and exit. This saving and restoring was shown to be simply a special case of the notion of dynamic local expression, evaluated in access mode on procedure entry (to get values to save) and in update mode (to restore the values) on exit.

This analysis showed that the previous use of "vars" for non-lexical local variables had mixed up two roles: declaring a permanent variable, and specifying entry and exit actions. These roles were now separated in that "dlocal" could be used for the latter purpose. It would be cleaner to force the declaration of permanent variables always to be done globally, but this suggestion was resisted on the grounds that it would stop a lot of programs working. So local "vars" declarations with their dual role were retained.

Other extensions to Pop-11

One by-product of the extensions required for Common Lisp was the introduction of generalised properties which allowed complex items to be used as indexing keys to retrieve stored information, unlike the old Pop2 properties, which indexed on the address of a pointer. The new mechanism required the user to provide a hashing function for the generalised properties, but it soon became clear that it would be useful to provide a built in default hashing function that could be redefined by users for particular data-types, to suit their needs. Thus was born "syshash" and "newmapping" the simplified version of "newanyproperty". The syshash procedure was one of several "overloaded" procedures that used the data type of its argument to select an appropriate procedure to run, making the new association mechanism very general and modular, since suitable hashing functions could be provided for different data-types as they were introduced.

A major generalisation was later introduced by Roger Evans - a "destroy" property, which could be used to associate with an object a procedure to be run when the object becomes garbage and is about to be discarded: a ‘destroy action’.

This makes it possible to do things like telling a window manager that a particular window is no longer needed because the data-structure with which it was associated is now garbage. Similarly in a network of distributed communicating processes, it is now possible for the information that some item has become garbage to be communicated automatically to other processes that need to know. I suspect that a variety of important applications of destroy actions will emerge as experience with use of the facility grows.

A Poplog Pop-11 library program shows how the generalised association mechanisms can be used to implement a "views" package, in which different values are associated with objects in different views. A particular case is associating different truth values with the same proposition in different hypothetical contexts, e.g. while exploring alternative possible plans to solve some problem.

Classes and object oriented programming

The notion of allowing each data-type to have its own hashing function was an instance of a more general facility, the Pop-11 "class" mechanism. There are several built in classes, such as integer, ratio, procedure, vector, word, pair, reference etc. In addition users can define additional record classes and vector classes. Each class has associated a collection of information, including specific procedures, for instance procedures for equality testing, for printing, for accessing and updating components, etc. The information associated with a data-type is encapsulated in a special type of record, a "key". There is one key for each type (and keys have their own key, whose key is itself).

This notion of a class is an extension of the POP2 data-type mechanism in that several aspects of the information associated with a class can be changed by the user, for example the class_print, class_=, class_hash, and class_apply, the latter being the procedure to be invoked whenever an instance of the class is treated as a procedure, i.e. applied to something.

This mechanism provides a simple but useful, one-layered, single-inheritance object oriented system.

From 1983 we began to experiment with a variety of different object oriented extensions to Pop-11, some including multiple inheritance, meta-classes and mixins. One of them used Pop-11 processes as class instances and the process variables as slots. Mark Rubinstein, a former student of ours who worked on Poplog for a while, implemented a far more sophisticated OOP system. This is known as LIB FLAVOURS, and has been used by many projects.

A more efficient, though possibly less general, OOP system is being designed and will be added later.

Sections and identifiers

Sections in POP2 made it possible to have different portions of programs developed by different programmers, without fear of clashes between identifiers, since only the identifiers ‘exported’ from a section were accessible outside it. These sections were tree-structured, unlike the "package" notion of Common Lisp.

In Poplog Pop-11 the notion of a section was extended in various ways to make the mechanism more generally useful. In particular, it was made possible for the compiler to re-enter a section, so that, for example, a package defined in a section could have optional extensions that were compiled in the same name space. This required the introduction of section path-names so that a deeply nested section could be entered directly. It was also made possible to refer explicitly to an identifier defined in a section by prefixing the section pathname. Thus if foo is defined in subsection SEC2 of subsection SEC1 section SEC0, then it can be accessed from outside these sections using the format "$-SEC0$-SEC1$-SEC2$-foo": not particularly pretty, but at least it makes possible what was not previously possible.

Another extension was the introduction of a "global" declaration type making it possible to declare that an should be automatically imported into all lower level sections.

In fact, some of the need for sections was removed by allowing "top level" lexically scoped identifiers. Thus global variables and procedure names defined in a file could be declared to be lexically scoped, meaning that they could be accessed only by other procedures whose definitions were syntactically nested within that file. This made it possible in many cases to avoid the overhead of a section. More precisely the scope of "top level" lexical identifiers was made to be a particular compilation stream, and a facility was provided to enable one file to be "included" in another as part of the same compilation stream, by analogy with " include" in C.

This section mechanism made it possible to have two prolog databases within Poplog, running in different sections.

Closely related to the development of the section mechanism was the introduction of a clear division between words, which are the globally accessible entries in the Pop-11 dictionary, and identifiers, which are the entities that have syntactic properties and denote values. A section is then simply a mapping from words in the dictionary to identifiers. A collection of procedures was provided to enable operations on identifiers to behave in a manner independent of the current section, unlike, for instance valof(word), which can associate different values with the word in different sections, causing surprises when programs defined in a section use valof, and then run when another section is current.

In addition, the syntactic properties of identifiers were enriched to cope with a variety of new facilities. For example besides ordinary identifiers, macros, and infix operators, all provided in POP2, Pop-11 now allows active identifiers with an associated integer (the "multiplicity"), syntax words, and syntax operators with a numeric precedence. Along other dimensions, identifiers can be specified as lexical or permanent, constant or variable, global to sections or not, and of type procedure or not. It may be possible later to introduce new identifier types, in the interests of efficiency and improved compile time checking.

Extensions to the store manager

The extension of Pop-11 to permit external procedure calls, together with the need for more user control over store management led to extensions to the store manager. For example there are now two garbage collectors, a faster one using the technique of copying all non-garbage to a new area then copying it back, and a slower one for use when not enough space is available for the copying version. In addition, users can "lock" the heap when it is known that everything in it at a particular point is non-garbage. This substantially reduces the effort required from the garbage collector, and the space required for the copying version. For example, when compiling a large program it is possible to re-lock the heap after each file is compiled, thereby dramatically reducing the total time required for garbage collection.

It is also possible to create "layered" saved images, so that users can share certain programs in a saved image, and then build their own saved image relative to the shared one. On some operating systems (VMS, Dynix, SunOS 4.0) the shared images will map into shared memory, with substantial efficiency gaines on multiprocessing systems.

In order to accommodate externally linked procedures that could dynamically allocate store, the Poplog store manager was generalised to accommodate a segmented heap, where "holes" in the heap were reserved by external programs for their own use.

The emergence of a two-level virtual machine

As the demand for Poplog to be ported to more machines grew, it became clearer that the porting task should be simplified. To achieve this John Gibson devised a mechanism using not just one virtual machine, but two, a high level Poplog virtual machine (PVM) and a low level Poplog implementation machine (PIM). The PVM provides powerful facilities making it a suitable target for compilers for high level languages. The PIM is a far more primitive, at a level similar to the VAX instruction set, and therefore is much easier to translate to a variety of machine languages for different architectures. A language-independent and machine-independent compiler bridges the gap between the PVM and the PIM. Because this compiler is both language independent and machine independent a lot of effort can be put into making it fast and enabling it to do optimisation while compiling.

Thus it is (relatively) easy to add a new language that will run on all architectures supporting the PIM, and it is easy to port the PIM to a new architecture whereupon all the languages targeted at the PVM are immediately available (along with a rich software development environment, integrated screen editor, sophisticated store manager, full operating system interface, etc). Further, new languages added to Poplog using the tools provided will tend to be more robust than a stand-alone compiler built from scratch, for which all the design and implementation work has to be done specially. Of course a specially tailored compiler can be more efficient.

This design has kept the development and maintenance costs of Poplog far lower than they would have been for four separately implemented incrementally compiled languages running on a range of machines, as well as supporting tightly-coupled mixed-language programming.

The actual mechanism is more complex than I have indicated as the incremental compilation of user procedures to machine code ready for execution requires slightly different mechanisms from the batch compilation of system sources to produce assembly language files for rebuilding or porting the system.

Miscellaneous developments

Over the last few years, under the pressure of requirements from users, a number of further extensions were made, such as provision of built in mechanisms for vectors of signed integers, new "fast_" procedures for improved efficiency, a generalised signal handling mechanism, improved tracing and debugging facilities, extensions to the external language interface, more flexible versions of the recordclass and vectorclass data-type declaration facilities, extended string handling, extended printing procedures, a facility to enable a property to invoke a procedure to simulate an association if one is not found in the table, more flexible terminal handling and input/output facilities, new facilities to control the behaviour of the compiler, for instance so as to allow the user to vary the trade-off between speed and safety.

Fortunately, the autoloadable library mechanism makes it possible for some of the extensions to be put into the library rather than into the main system, so that they do not add to the overheads of users who do not require them.

Ports to new machines - back from VMS to Unix

From mid 1984 a VAX running Berkeley Unix(tm) became available, so the Poplog system was ported to that, and subsequently versions were made available on a variety of machines running different kinds of unix, starting with the Bleasdale, then Sun-2, Sun-3, GEC-63, Hewlet Packard workstations, Apollo, Orion-2 (with Clipper Processor), Sequent Symmetry (multiple 80386 processor), Sun-4, Sun-386i. Additional ports are likely, e.g. to the Decstation with MIPS processor.

In order to facilitate transfer of programs between VMS and Unix, the VMS version of Poplog (and therefore Pop-11) was altered to permit the use of filenames in Unix format. Since many of the same file manipulation procedures are available in both VMS and Unix Poplog, this means that many programs can now be ported between the two operating systems without change, provided that they all use the Unix format for file names.

In 1987 a subset of Pop-11 known as Alphapop was ported to the Apple Macintosh by Cognitive Applications Ltd, and very nicely integrated with the Mac user interface. This implementation received a glowing review in *Byte* May 1988. Most Alphapop programs will run in Poplog Pop-11, but since Poplog Pop-11 provides a much wider range of facilities, the converse is not generally true. However, a number of teaching programs developed in Poplog Pop-11 are available for use with Alphapop.

Future developments

Although it is possible for users to employ Pop-11's language extension facilities to produce extensions that are tailored to their requirements, we have continued to try to abstract from commonly required types of extensions to find ways of making them easier to integrate with the language and its support environment, including the editor.

For example users have always been able to define macros or syntax words with their own opening and closing brackets, for creating new program or data modules or their own looping constructs for iterating over special purpose datatypes. If one introduces these new constructs the use of different words can mask the relationship with existing constructs, and prevent the existing tools (e.g. editing aids) from working straightforwardly with the new forms.

In order to overcome this we recently introduced the notion of a syntax form that has user-definable syntax words to extend that form. Thus the "define ... enddefine" construct can now be used with a user-specifiable role, by including a colon followed by a sub-sytax word after "define", as in

```
  define :class .....
      .....
  enddefine;

  define :instance ....
      .....
  enddefine;
```

The behaviour of define will then be controlled by a user-defined procedure associated with the keyword. Similarly although Pop-11 comes with a rich collection of looping forms, including:

```
 for <variable> in <list> do <actions> endfor

 for V1 V2 ...Vn in L1, L2,...,Ln do <actions> endfor

 for <variable> on <list> do <actions> endfor

 for V1 V2 ...Vn on L1, L2,...,Ln do <actions> endfor

 for <variable> in <structure> using_subscriptor <procedure> do
     <actions>
 endfor

 for <variable> from <number> by <number> to <number> do <actions> endfor

 for <variable> from <number> to <number> do <actions> endfor
         (Default: by 1)

 for <variable> by <number> to <number> do <actions> endfor
         (Default: from 1)

 for <variable> to <number> do <actions> endfor
         (Defaults: from 1, by 1)

 for <action> step <action> till <condition> do <actions> endfor
```

it is useful for users to be able to use "for ... endfor" to specify their own iterative constructs tailored to particular forms of data, for instance iterating over entries in an association table. Ian Rogers designed and implemented an extension allowing a user-defined keyword (a sub-syntax word, like "in"), following the variable list, to specify the action to be taken.

Thus the use of the keyword "for" can continue to be used to indicate an iterative control structure, while allowing new versions to be introduced for particular problem types. This is analogous to the change that allowed "define" to be used as a syntax word to indicate the general notion of a top level declaration of a new object or type of object, while allowing a user definable sub-syntax word to indicate a particular specialisation.

There are probably other examples of this general notion waiting to be discovered. It appears to be an extension to syntax of some of the ideas of inheritance previously associated with data and procedures. However I expect there is still work to be done to devise a good clean general form of this idea.

I do not know how far this kind of attempt to identify and provide useful abstractions can go or should go. The evolution of Pop-11 needs to be slowed down, and the formation of a Pop91 standards committee must be an excellent thing, especially as there is no danger that a frozen standard will get in the way of the mechanisms that allow users to define their own higher level application-specific constructs, in order to aid readability and therefore program development, testing, and maintenance.

One of the issues that still needs to be addressed is provision of new type declarations to enable more efficient programs to be written and to enable the compiler to do more checking.

It is possible that some of the need for this is reduced since Robert Duncan and Simon Nichols added Standard ML to Poplog. Since ML is a strongly typed functional language, in which types can be polymorphic which overcomes some of the restrictions of typed languages like Pascal, it would be possible in principle to use ML to define those procedures that require the compile time checking and and run-time efficiency of a typed language. Efficient procedures written in ML could then be invoked as needed by procedures using the more general and flexible mechanisms of Pop-11.

However, in order to take advantage of this it may be necessary to improve the facilities in Poplog for compiling languages like ML. For example, it should not always be necessary to convert integers or decimals between their standard machine representations and the standard Poplog representations, which require two bits to be used for type identification.

Facilities for fast integer arithmetic on machine integers are already provided in SYSPOP, the extended dialect of Pop-11 used for Poplog system development, which also provides C-like pointer manipulation and other facilities required for very efficient programs. SYSPOP and the POPC compiler, which is used to compile the Poplog system sources, are not at present available to users, but will be shortly (probably early in 1990). This will enable users to compile and link Pop-11 and SYSPOP programs required for an application, which can then run without unnecessary system development tools normally built into Poplog Pop-11.

Modified versions of POPC could be used for cross-compilation for embedded systems, opening up a host of new potential applications for Poplog.

Pop-11 and Lisp

It is very interesting for Pop-11 users to note that whereas Common Lisp was supposed to provide the standard that would eliminate the problems arising from the wide diversity of Lisp systems developed previously, in fact a number of divergent Lisp systems continue to be used, and it looks as if Lisp dialects based on Scheme, such as the language T, are growing in popularity. There is also a move to produce a European dialect of Lisp that is smaller and cleaner than Common Lisp.

The Scheme-based dialects are much closer to Pop-11 than Common Lisp is since they treat functions as ordinary values of variables, and this allows more general and elegant procedures to be written. However, all dialects of Lisp share a major problem, name the very simple and elegant syntax that is very attractive to computer scientists and mathematicians, but is often very unattractive to working programmers, and very confusing for some learners.

Sometimes this is simply because the programmers are accustomed to other languages. But some of the resistance to Lisp can be justified by the fact that a syntactically impoverished language imposes a greater cognitive load on the user.

For example in Lisp the significance of an expression will depend on its position in a list structure, requiring the reader to examine a larger context in order to interpret the expression, wheres in Pop-11 and other languages with a Pascal-like syntax the significance is indicated by local key-words. This is illustrated by the difference in multi-branch conditionals.

In Pop-11 the form

```
if      .... then  ....
elseif  .... then  ....
elseif  .... then  ....
elseif  .... then  ....
elseif  .... then  ....
else    ....
endif
```

would correspond, in Lisp, to

```
(cond
    ((....)(....))
    ((....)(....))
    ((....)(....))
    ((....)(....))
    ((....)(....))
    (( t )(....)))
```

This structure is clear enough when it can all be taken in at a glance, but in a real program one may be confronted with a line of the form

```
((........)
```

whose significance can be ascertained only by looking some way up, whereas the corresponding line of Pop-11 would be

```
elseif ..... then
```

making it very clear that this is a condition in a multi-branch conditional.

The extra syntactic redunancy also helps the compiler to generate more helpful error messages if, in the course of writing or altering a program a bracket is put in the wrong place, though the likelihood of this is reduced by a good editor.

This is simply one example to illustrate the point that although syntactic richness may increase the number of different syntax words that users have to learn, and may complicate the tasks of both syntax-driven editors and parsing programs or compilers, it nevertheless can play an important role in increasing intelligibility for humans, and this may be important both for learning to use the language and for production and maintenance of code in large teams where people often have to read and work on programs written by others.

Of course, these syntactic limitations are not as important as the semantic limitations of some other languages. I would not, for example, use the more readable syntax of Pascal as a reason for preferring it to Lisp, which has far more flexibility and power. However, when semantic limits are not the issue, the syntactic differences may be decisive in selecting a language.

If the need to write, read, or modify programs disappears in the future because all the work is done at a higher level of abstraction, with code handled only by programs not people, then these differences between languages will become irrelevant. However, the need for human programmers will probably remain with us for many years to come, even if the proportion of non-programmers using computers continues to grow.

My own conjecture is that *one* of the reasons (and there may be others) why AI programming languages and techniques have not been widely accepted by non-AI programmers is simply the syntax of Lisp, the only AI language many of them have looked at. This conjecture is supported by the number of programmers in industry who were put off by Lisp and yet liked Pop-11 despite its similarity in power. This difference in acceptability was shown by a survey of Alvey-funded projects in the UK conducted by Ken Hartley at the Rutherford Appleton Laboratory. Only three languages were rated "good" by *all* users, on a "good/indifferent/poor" scale: POP-11, PARLOG and C++. Of these three, POP-11 had the most users. The complete list of languages mentioned included PROLOG, LISP, C, Pascal, Fortran, Ada, and others.

By offering programmers familiar with conventional languages the opportunity to appreciate the advantages of AI tools for rapid prototyping and thorough testing, without the shock of unfamiliarity that Lisp gives them, Pop-11 has the potential to make a large difference to the number of programmers willing to use AI development environments. Many who have made the transition to Pop-11 can then, if necessary, turn to Lisp without having to learn so much all at once.

Acknowledgements

The University of Sussex has provided a congenial and accommodating environment for this work for the last fourteen years, and some of the far-sighted administrators supported us morally and financially. The Science and Engineering Research Council and the Alvey Directorate helped with funding for some of the developments. Staff at SD-Scicon, now Integral Solutions Ltd, made important suggestions from the marketing point of view, did some of the porting work, and collaborated with us on support and maintenance.

Robin Popplestone must, of course, be acknowledged as the originator of many of the features that have made the POP family of languages so pleasant to use. Many colleagues and students at Sussex and some users in other places have influenced developments by providing ideas and sometimes code, including especially, at Sussex, Andrew Casson, Jon Cunningham, Robert Duncan, James Goodlet, Steve Hardy, David Hogg, Steve Isard, Tom Khabaza, Jonathan Laventhol, Rudi Lutz, Chris Mellish, Aled Morris, Simon Nichols, Allan Ramsay, Ian Rogers, Mark Rubinstein, Ben Rubinstien, Chris Thornton, John Williams, and David Young, and at Hewlett Packard Laboratories Steve Knight and Chris Dollin.

However, there is one person without whom Poplog Pop-11 could not have happened, and who must take the credit for all the most important developments since 1981, namely John Gibson.