Pascal — The Language and its Implementation
Edited by D. W. Barron

10

Two 1900 Compilers

J. Welsh

Since the first definition and implementation of Pascal [1, 2], two compilers for ICL 1900 series computers, known as 1900 Pascal Mk1 and Mk2 respectively, have been produced at Queen's University, Belfast. This paper contrasts the techniques used in their production, and summarizes the significant features of the Mk2 system.

1. The Mk1 compiler

The Mk1 1900 Pascal compiler was produced by a six-month project in 1971 from the original Zurich compiler for CDC 6000 series computers, by the technique usually known as a half-bootstrap [3]. This involved rewriting the code generation parts of the existing Zurich compiler to produce 1900 object code. Two compilation cycles on the Zurich 6000 series machine then produced a compiler which was executable on, and generated code for, a 1900 machine.

Since the hazards of this transportation method are minimized by minimizing the changes made to the existing compiler, the structure and analysis logic of the Mk1 1900 compiler were carbon copies of those of the Zurich original. The new code-generation logic introduced was simple but effective, treating the 1900 as a single-computation register machine. The resultant compiler compared favourably with its Zurich parent in size and speed of compilation. The object code which it produced compared equally favourably with that from ICL's standard FORTRAN and ALGOL 60 compilers. The compiler also proved extremely reliable. In the five years following its creation it has been distributed to some forty other ICL 1900 installations, with only minor amendments in that time.

The most striking aspect of the Mk1 project, however, was the ease with which the original transport operation was carried out. The half-bootstrap technique was the method adopted for most early attempts to transport Pascal to other machines, with variable success. It has obvious disadvantages:

(a) The transport exercise takes place on the donor rather than the acceptor machine.
(b) The information transported is (possibly untested) machine-oriented code.

The Mk1 project showed that these disadvantages can be overcome. However, since the release from Zurich of the Pascal-P system [4], most implementors have opted for the more convenient full-bootstrap approach, which avoids these disadvantages. It is worthwhile therefore at this time to point out the less obvious advantages of the half-bootstrap method, which are:

(a) The effort and potential problems of implementing a fixed intermediate code on a possibly unsympathetic architecture are avoided.
(b) The fact that the success of the transport operation depends on the correctness of the code-generation logic introduced provides a powerful incentive for the implementors to get it right!

2. The Mk2 compiler

The Mk2 1900 Pascal compiler was produced by a project spread over a much longer period, 1974–76, with a distinctly different approach. The initial impetus to provide a new 1900 Pascal system came from the revised Pascal-language definition [5], but the objectives adopted for the project were considerably wider than a simple revision of the language implemented. They included the following:

(a) To improve the structure and clarity of the compiler as a vehicle for experiments in language design and implementation. In particular it was intended to achieve a structure which would facilitate modification of the compiler to generate different object code forms for different machines.
(b) To improve the quality of the object code generated.
(c) To improve the error handling and diagnostic facilities provided by the system during the compiletime, runtime, and post-mortem phases of program development.

It is worthwhile to examine the achievements of the Mk2 project in the light of these objectives.

2.1 The structure chosen

The Mk2 compiler was designed and written by following the stepwise refinement sequence outlined by Ammann [6]. At each stage existing compilers, and in particular Ammann's own Pascal 2 compiler, were critically examined, and the best structure and formulation chosen on this basis.

Up to the point of code generation no major departures from Ammann's struc-

ture were chosen, though the textual order, formulation, and in particular the identifiers used within this structure were systematically revised to improve the transparency of the final code.

For code generation, however, the scattering of imbedded generative sequences throughout the analysis logic was not considered consistent with objective (a) of the project. Instead a structure was sought which would allow a textual separation of the generative code within the compiler, while retaining the efficiency and flexibility of a one-pass parallel analyser/generator. The overall textual and functional structure sought for the compiler was thus as shown in Figure 1.

To meet the project objectives the analyser/generator interface had to express the generative actions required in an object code-independent form. This was achieved as follows:

(a) The generative attributes of whose existence the analyser must be aware are abstracted as values of appropriate Pascal-defined types, e.g.

```
type type representation = . . . . .;
     runtime address     = . . . . .;
     code label          = . . . . .;
```

The detailed definition of these types is provided by the generator in a form appropriate to the object code being generated. However, the analyser requests, stores, and passes back values of these types from and to generator procedures, without any assumptions of their detailed representation or significance.

(b) The code to be generated is abstracted as a sequence of operations on a hypothetical stack machine. Each operation is requested by the analyser as a call to a corresponding generator procedure, passing relevant attributes as paramenters when appropriate, e.g.

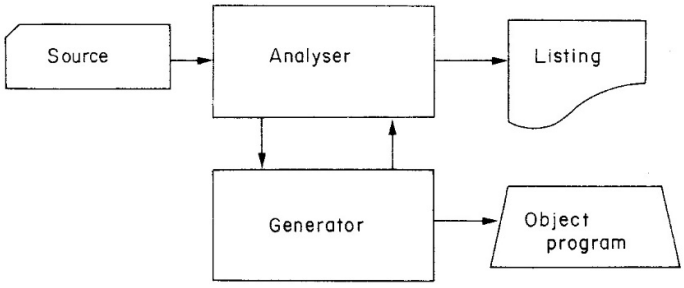

Figure 1. Functional and textual structure of Mk2 compiler

procedure *Stackreference* (*address:runtimeaddress*);
procedure *Jumponfalse* (*label:codelable*);

By controlling the types and representations of these parameters the generator implementor in effect tailors the abstract interface code to the needs of his particular machine.

In this way both storage allocation and code generation is modelled in the Mk2 analyser source without any knowledge of the actual object machine involved. Compilation of this analyser in the context of an appropriate generator which defines the generative types and procedures required will produce a compiler for any chosen object code. The generator can likewise be programmed without any knowledge of the analyser's internal functioning, beyond its interface specification.

On the basis of this interface the Mk2 project was completed by programming a generator for absolute 1900 machine code. The resultant compiler has now replaced Mk1 as the distributed 1900 Pascal system. The same analyser and interface is being used by the University of Southampton to produce a Pascal compiler for the very different architecture of ICL's 2900 series machines [7]. While it undoubtedly deserves some further refinement the Mk2 analyser and the generative interface which it incorporates offers an interesting and flexible base for Pascal implementors. For half-bootstrap transportations it has the advantage of isolating those parts of the compiler which must be rewritten, while providing a flexible abstract model of the generators required. Equally it might be used as a base for cross-compilers of object code for mini- and microcomputers, an avenue which is currently being explored in Belfast, or simply as a means of providing alternative object-code options on a single machine.

2.2 The object code produced

The Mk2 generator produces absolute 1900 code maintaining parallel simulations of the runtime states of the hypothetical stack and actual 1900 machines, delaying generation of actual 1900 code as long as the equivalence of these simulations permits. Within this general framework the following particular strategies are consistently applied:

(a) All operations involving constant operands are folded or optimized whenever possible.
(b) All available 1900 registers are exploited to minimize the use of temporary locations during expression evaluation.
(c) Existing register contents are exploited in certain critical contexts, e.g. parameter passing and block entry. (A more general exploitations of previous register contents could be supported by the simulation but has not yet been implemented.)

The quality of the resultant object code, compared with that produced by the Mk1 system, has been measured for the following five programs:

(a) Three programs given by Wirth [2], which perform real matrix multiplication, integer sorting, and character frequency counting, and thus typify simple numeric and character processing.
(b) The Ackermann function, advocated by Wichmann [8] as a test of the procedure and parameter mechanisms in recursive block-structured languages.
(c) An intermediate version of the Mk2 compiler itself, which employs the full range of Pascal's facilities in a typical systems programming application.

The length and execution time of the object code generated for each program by the Mk1 and Mk2 compilers (on a 1906S computer) are shown in Table 1.

The longer time taken for character counting under Mk2 is attributable to the fact that in the Mk2 system the binding of I/O devices to program file variables is delayed until runtime, with a consequent overhead in the file access routines. Otherwise the results show a consistent improvement of 10–20 per cent. in the length and execution time of the programs used.

While any improvements in object-code efficiency are welcome, the relative modesty of those achieved clearly demonstrates that a multiplicity of registers for expression evaluation does not radically improve the performance of most programs. One may in turn question the justification of the more complex code-generation strategy incorporated in the Mk2 compiler. However, this strategy should not be evaluated on the basis of these measurements alone since:

(a) it is capable of further exploitation in code optimization and
(b) it readily supports other generation features such as the runtime error checking described in the following section.

2.3 Runtime error checking

The third objective of the Mk2 project, namely to improve the error-handling and diagnostic facilities provided, has been tackled in three ways:

(a) The syntax and semantic error-recovery techniques used by Ammann in the Pascal 2 compiler has been incorporated with minor modification, to provide improved recovery with few repetitive or spurious messages.
(b) The runtime error-checks option has been implemented in a way which exploits the information provided by the program as to the values which may occur, in the manner outlined below.
(c) The provision of a diagnostics package implementing execution profiles, traces, and symbolic post-mortem dumps has been undertaken by the University of Glasgow [9].

Table 1. Comparison of object code produced by Mk1 and Mk2 compilers

Program	Code length			Execution time		
	Mk1	Mk2	Mk2/Mk1	Mk1	Mk2	Mk2/Mk1
Multiplication of 50 X 50 matrices	82	65	0.79	1.76	1.60	0.91
Sorting 1000 integers	58	49	0.84	4.7	4.1	0.87
Counting 1,000,000 characters	29	24	0.83	12.5	13.6	1.09
Ackermann [3, 8]	43	37	0.86	13.2	11.0	0.83
Mk2 compiler	24.8K	22.4K	0.90	58	45	0.78

A significant feature of Pascal is the increased precision with which the range of values to be taken by program variables can be specified–as enumerated or subrange types. The programmer may expect that the system ensures that this specification is observed, either by compile-time or (possibly optional) run-time checks. Provided these checks are made, however, the compiler may in turn exploit these guaranteed properties at other points where the variables occur.

In the Mk2 compiler an assertion is constructed about each operand in an expression indicating the range of values it may take, and hence whether overflow may occur in its evaluation. For constants the assertion is trivially determined by the value itself; for variables it is derived from the declared type of each variable; for operands which are themselves the result of some operation, the assertion is computed from the operation and from the assertions for the operands to which it is applied.

In this way an assertion is available to the compiler about the value of each expression compiled. Where this expression occurs in a context where a limited range of values is acceptable, e.g. as a subscript, a case index, or a value for assignment to a subrange variable, this assertion can be used to determine whether a runtime check is necessary, and what form it should take. The need for checks against arithmetic overflow are decidable in the same way.

Thus the amount of runtime checking which is incorporated in a program depends on the precision with which the programmer declares the variables involved, and in general the checks may *decrease*, rather than increase, as more precise declarations are used.

To demonstrate this the text programs given earlier have been remeasured, with the runtime check options selected, to determine the degradation in code length and execution time which these checks introduce. Where appropriate the programs have been run in two forms–the first using variable declarations without particular regard to the range of values taken, the second using the most precise subrange declaration possible for each variable. The resultant degradation factors are shown in Table 2.

The results for the first two programs, which depend heavily on subscripted variable access, show that while careless declaration of index variables produces a

Table 2. Degradation factors produced by Mk2 check option (a) with careless variable declarations, (b) with careful variable declarations

	Code length		Execution time	
	(a)	(b)	(a)	(b)
Matrix multiplication	1.60	1.03	2.68	1.04
Integer sort	1.67	1.04	2.24	1.07
Character counting	1.08	–	1.04	–
Ackermann function	1.24	–	1.11	–
Mk2 compiler	–	1.03	–	1.09

two- or threefold increase in execution time when error security is requested, the same security can be achieved at very little cost by careful declaration of these variables and the use of a compiler which exploits the information provided.

For all of the programs measured the degradation produced by the Mk2 systems's check option, used on well-coded programs, is a small price to pay for the additional error security achieved. Indeed for four of the five programs tested, the degradation is less than the improvement in execution speed which the Mk2 object code showed over that of its Mk1 predecessor, so that the Mk2 programs *with error security* are more efficient than the equivalent Mk1 programs *without*!

2.4 Counting the cost

As the preceding sections show, the Mk2 system offers some significant advantages over its predecessor in terms of structure and adaptability, object-code quality, and cost of error security. At what price are these advantages obtained? Table 3 shows the compiletime requirements of the Mk1 and Mk2 systems in compiling an intermediate version of the Mk2 system.

The significant increase in the object-code length of the Mk2 compiler is its least attractive feature. It is attributable to:

(a) the additional language features implemented, in particular *packed* data structures;
(b) the structural separation of analysis and code-generation logic within the compiler; and
(c) the more structured and more powerful code-generation logic used.

The lack of reduction in working storage, despite the use of the new *packed* facility, is also attributable to these factors.

The increased storage requirement may, however, be offset against the considerable increase in compilation speed. If compilation cost is measured as a space × time product then the appropriate formulae for the cost of compiling 100 lines of Pascal source under each system are of the form

Table 3. Comparison of compiletime requirements of Mk1 and Mk2 systems

Compiletime requirement	Mk1 system	Mk2 system
Fixed object code and data storage	20.5K	30.3K
Working storage	12.6K	12.7K
Time taken to compile (9000 lines)	76 sec	45 sec

Mk1 : $(17.3 + 0.84T)$ *K*words × sec
Mk2 : $(15.2 + 0.50T)$ *K*words × sec

where T is a measure of the working storage requirement of the source program determined chiefly by the program's complexity or depth of nesting and the number of identifiers used. Thus the Mk2 system has a basic advantage in cost of compilation over Mk1 and this advantage increases with program complexity.

3. Conclusions

While the first generation of Pascal implementations represented a significant breakthrough in language implementation and portability, second-generation systems show clearly that further advances can be made. The Mk2 1900 system's error-checking strategy only indicates that the low or zero cost error security, which Pascal makes possible in principle, can become a practical reality; but much remains to be done. Likewise there is much that can be done in other directions, and the language Pascal is likely to remain both a challenge and a source of guidance for language implementors for some time to come.

Acknowledgements

The author is deeply indebted to Colum Quinn and Kathleen McShane, without whose design and programming skills neither 1900 Pascal system would have come to fruition.

References

1. Wirth, N., The programming language Pascal, *Acta Informatica*, **1**, No. 1, 35–63 1971.
2. Wirth, N., The design of a Pascal compiler, *Software–Practice and Experience*, **1**, No. 4, 309–333, 1971.
3. Welsh, J., and Quinn, C., A Pascal compiler for ICL 1900 series computers, *Software–Practice and Experience*, **2**, no. 1, 73–77, 1972.
4. Nori, K., *et al.*, Pascal-P implementation notes, Chap. 9 in this volume.
5. Wirth, N., *The Programming Language Pascal* (revised report), Nr. 5, Berichte des Instituts fur Informatik, ETH, Zurich, November 1972.
6. Ammann, U., *The Method of Structured Programming Applied to the Development of a Compiler* (Eds. A. Guenther *et al.*), International Computing Symposium 1973, pp. 93–99, North Holland, 1974.
7. Rees, M. J., *et al.*, Pascal on an advanced architecture, Chap. 13 in this volume.
8. Wichman, B. A., Ackermann's Function, A study in the efficiency of calling procedures.
9. Watt, D. A., and Findlay, W., A Pascal diagnostics system, Chap. 11 in this volume.