Correspondence

To The Editor:
The recent article by J. Paakki[1] contains controversial statements that are not sufficiently well documented and that might easily be misinterpreted. We think that the readers deserve a few additional comments.

One of Paakki's principal conclusions is that parsers written in Prolog (more exactly: in the DCG formalism[2,3]) are too inefficient. He supports his claim primarily by noting the inefficiency of his own parser, which he attributes mostly to the fact that the standard DCG implementation we employed yields unavoidably a nondeterministic parser.

It is not clear that this is indeed the principal reason. In Paakki's table 3 the execution time for the prolog parser grows more or less linearly with program size, until it explodes at some point. Curiously, the explosion occurs much earlier in the interpreted parser: according to Paakki's data, it becomes grossly inefficient for source programs of more than 60 and less than 300 lines. The compiled parser blows up on programs of more than 900 lines. The disparity strongly suggests that the enormous slowdown is related to memory management, for example to garbage collection. (A similar phenomenon can be observed in Paakki's scanner execution times.)

It is true that naïve DCG grammars are converted to backtracking top-down parsers whose execution time may grow exponentially with the size of the parsed string. However, programming languages for which it is customary to build top-down parsers (for example the Pascal family of languages[4,5,6]) allow deterministic parsing. In particular, a DCG for such a language can easily be constructed in a way that allows deterministic execution. If a language is described by an LL(1) grammar, a deterministic DCG parser can even be derived *automatically*[7] from a higher level formulation such as an EBNF grammar.[8]

We have experimented with such an automatically derived parser for the language Pascal-S[9] (definitely not a toy language). The parser builds a parse tree and has adequate error detection, although it does not attempt to recover from errors; the error detection is implemented rather inefficiently and accounts for about 80% of the time.[7] The Prolog system we used was Sicstus prolog (a good portable implementation of Prolog, but not the fastest), running on a SUN SparcStation 1. The timings do not include the cost of reading the list of tokens, nor do they include time spent in garbage collection, which was always smaller than the pure processing time. The following table lists the translation times for seven Pascal-S programs:

lines	tokens	CPU msec.
325	1313	1339
606	2620	2329
888	3927	3599
1452	6541	5929
2580	11769	10759
4835	22225	20540
7635	35295	33830

Over a wide range of program sizes, the speed is almost constant (about 0.9 of a CPU millisecond per token). The Prolog system (as installed) ran out of memory for a program of 9000 lines and over 40000 tokens, and the speed is certainly not breathtaking, but in our opinion these results contradict Paakki's thesis that the DCG formalism might be good enough for small toy examples but its conventional implementation is absolutely too naive for larger or more practical cases.

It is worth noting that a grammar need not be strictly LL(1) for top-down parsing to be deterministic. The context-free grammar of Oberon is even ambiguous, but the ambiguity is easily resolved – and the parser is wholly deterministic – if semantic analysis is carried out simultaneously with syntactic analysis. The DCG formalism is singularly convenient for expressing such combined analysers.

Admittedly, some languages are more suitable for bottom-up parsing. It is possible to implement a DCG as a bottom-up parser,[10] but in our opinion this is not really worthwhile: since we lose the ability to use inherited attributes, we might as well use one of the conventional parser generators.[11]

We will now turn to another point. Paakki says that implementing the symbol table as a Prolog term makes it awkward to handle details arising from block structuring. He also claims that it would lead to gross inefficiency (as large data structures they would consume too much space when passed as parameters through the analyzer – a strange thing to say, since the parameters are really pointers to the data structures, not their copies). He accordingly implements the symbol table in the internal Prolog database, even though such a representation is also inconvenient for block structuring and makes it difficult to perform updates in a clean way.

We think this was a wrong design decision. The conventional method of implementing a symbol table in Prolog is to keep it in an open unbalanced binary search tree. (In an *open* tree, where leaves are represented as Prolog variables, insertions do not require the overhead of copying that is usual for other declarative languages.) A stack of such trees is an elegant and convenient symbol table for a block-structured language. The data structure is very similar to what one would write in Pascal, and it exhibits similar behaviour.

We performed an experiment in which identifiers from a conventional compiler of Pascal-S[12] – 11160 occurrences of 1147 names – were found in (or inserted into) an unbalanced binary tree. Programs written in C and Pascal needed about 2 CPU seconds, and the Prolog program needed 5 seconds (note that Sicstus does not compile to native code). A version that kept the symbol table in the internal database was indeed faster: 1.2 seconds. When compared to the overall costs of compiling such a large program, the difference is certainly too small to justify a markedly less convenient representation.

A final comment concerns scanning. As Paakki correctly points out, Prolog is not very suitable for writing scanners (and – we might add – other programs that perform many very simple operations). We timed three versions of a program that just reads all the characters from a file. For a file of one million characters in 20000 lines, the CPU times were 1.9 sec. for C, 4.5 sec. for Pascal, and 31 sec. for Sicstus Prolog (timings for other file lengths show strictly linear growth in all three cases). This is hardly surprising: in the Prolog program each character-reading operation is accompanied by the overhead of a resolution step. Scanners in Prolog will always be slower, but there is really no need to use such a powerful language for such a simple application.

FELIKS KLUŹNIAK

Department of Computer Science,
Linköping University,
S–581 83 Linköping,
Sweden.
(On leave from Warsaw University)

STANISŁAW SZPAKOWICZ

Department of Computer Science,
University of the Witwatersrand
P.O. Wits, Johannesburg 2050
South Africa.
(On leave from the University of Ottawa)

REFERENCES

1. J. Paakki. Prolog in practical compiler writing. *The Computer Journal*, **34**(1): 64–72 (1991).
2. A. Colmerauer. Metamorphosis grammars. In *Natural Language Communication With Computers*, edited Leonard Bolc, pp. 133–189. Springer-Verlag, Berlin (1978). (Lecture Notes in Computer Science 63).
3. F. C. N. Pereira and D. H. D. Warren. Definite Clause Grammars for language analysis – a survey of the formalism and a comparison with Augmented Transition Networks. *Artificial Intelligence*, **13** (3) 231–278 (1980).
4. K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, Berlin, second edition (1975).
5. N. Wirth, *Programming in Modula-2*. Springer-Verlag, Berlin, third, corrected edition (1985).
6. N. Wirth. The programming language Oberon. *Software – Practice and Experience*, **18** (7) 671–690, (1988).
7. F. Kluźniak. Generating DCG parsers. Technical report TR-90-10, Computer Science Department, University of Bristol (1990).
8. N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, **20** (11) 822–823 (1977).
9. N. Wirth. Pascal-S: A subset and its implementation. In D. W. Barron, editor, *Pascal – The Language and its implementation*, pp. 199–259. John Wiley & Sons, Chichester (1981).
10. U. Nilsson. AID: An alternative implementation of DCGs. *New Generation Computing*, **4** 383–399 (1986).
11. S. C. Johnson. Yacc – yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ (1975).
12. M. Rees and D. Robson. *Practical Compiling with Pascal-S*. Addison-Wesley (1988)(?).