Microcomputer PROLOG implementations: The state-of-the-art

by HAL BERGHEL and RICHARD RANKIN
University of Arkansas
Fayetteville, Arkansas

ABSTRACT

In this paper we discuss several characteristics of microcomputer PROLOG implementations including an overview of current products, a comparison of the range of built-in predicates, a description of the environment, and benchmark results.

From the collection of the Computer History Museum (www.computerhistory.org)

From the collection of the Computer History Museum (www.computerhistory.org)

INTRODUCTION

Although logic programming has a relatively short history in computer science, its impact has been significant. After the announcement that the Japanese Fifth Generation Project would standardize Japan's 1990s machines around the logic programming approach,[1] industry leaders and researchers began to devote considerable attention to this area. What follows is a general description of the various implementations of the logic programming language PROLOG that are available for microcomputers. These implementations are of considerable importance, for they allow virtually every interested person to enter the world of logic programming with minimal expense. It is our intention to acquaint interested readers with the current state-of-the-art.

PROLOG, as a logic programming language, developed from the early work of Kowalski[2,3,4] and Colmerauer[5,6] in the 1970s. As this work circulated, prototypes of the language appeared in France, England, Hungary, and Canada, and each was injected with some of its own design philosophy. As a result, there are at least three different models, each relying upon its own distinctive syntax, and each appealing to a particular subset of the research/development community. To standardize our treatment of a non-standardized language, we employ the Edinburgh nomenclature[7] in the following discussion.

One of the most significant aspects of PROLOG is that, at least in the ideal, it supports a clear distinction between the *logic* of the program and the mechanism of *control.*[8] This means that the programming is oriented toward the logic of the problem, leaving the control mechanism to the system. The important implication of this strategy is that the range of built-in predicates affects the convenience and speed of software development. However, since the various software houses have different design objectives, the predicates are not uniformly distributed over the entire range. Thus, some products may be better suited for certain applications than others. We provide a detailed classification of these predicates together with an analysis by product.

Of course, since the control mechanism is largely left to the implementation, differing strategies will have different effects upon performance. We also provide a series of benchmark results which shed light on the relative performance characteristics.

The products reviewed here are, alphabetically, Arity PROLOG, version 4.0 (Arity Corporation); micro-PROLOG professional (Logic Programming Associates); MPROLOG, version 2.1 (Logicware); PROLOG 2, version 1.2 (Expert Systems International); PROLOG-86+, version 1.0 (Solution Systems); Turbo PROLOG, version 1.0 (Borland International) and VML PROLOG, version 1.9m (Automata Design Associates). We believe these are the most current versions. Only one of the MS-DOS implementations that we know of, PROLOG-V, was not included (at the request of the manufacturer). One product from Applied Logic Systems was announced but not released as of this writing. This paper updates and integrates the results presented in earlier publications and reports.[9,10,11]

GENERAL DESCRIPTION OF THE IMPLEMENTATIONS

A general summary of the implementations appears in Table 1. As the table shows, two of the products provide compilers, and all but one provide interpreters. The lack of an interpreter for Turbo PROLOG is intended; the designers have developed a compiler that behaves as if it were incremental (!), therefore they believe the interpreter is not needed.

Three of the products support virtual memory (up to one gigabyte in some cases), and all but one provide shell support

TABLE I—Overview

Product:	P2	AR	LPA	MP	P86	VML	TUR
Version:	1.2	4.0	PRO	2.1	1.0	1.9	1.0
Interpreter	+	+	+	+	+	+	-
Compiler	+	+	-	-	-	-	+
Virtual Memory	+	+	-	-	-	+	-
Shell Support	+	+	+	+	-	+	+
DOS Services							
Time/Date	+	+	+	-	+	+	+
Interrupt Facilities	-	+	+	-	-	+	+
Directory Facilities	+	+	+	-	+	+	+
Keyboard Facilities	+	+	+	+	-	+	+
Internal Clock Timing	-	-	-	-	-	+	-
Editor	+	-	+	+	+	-	+
Interactive	+	+	+	+	+	-	+
Multiple Windowing	+	-	+	-	+	-	+
Screen Control	+	+	+	+	+	+	+
Modularization	+	+	+	+	+	+	-
Module Privacy	+	+	+	+	+	+	-
Export/Import	+	-	+	+	-	+	-
Multiple Worlds	-	+	-	-	-	+	-
Multiple Theories	-	-	-	-	-	+	-
Database Indexing	+	+	-	-	-	+	-
Clause Indexing	-	+	-	-	-	+	-
Hashing	-	+	-	-	-	-	-
B-Trees	-	+	-	-	-	-	-
Optimization							
Cyclic Structure Checking	-	-	-	-	-	+	-
Garbage Collection Control	+	+	-	+	-	-	-
TRO	+	+	-	+	-	-	+
Stack Control	-	+	-	+	-	-	-
System Information							
LIPS count	-	-	-	-	-	+	-
Heap Used/Remaining	+/+	+/-	-/+	-/-	-/+	-/+	-/+
CPU Time	+	-	-	+	-	-	-
DCG	+	+	-	+	+	+	-
Structured Programming	-	+	+	-	+	-	-
# b-i preds (approx.)	255	170	90	150	155	210	90

From the collection of the Computer History Museum (www.computerhistory.org)

which allows users to suspend PROLOG and execute independent object modules without altering the state of the interpreter (see the Built-In Predicates section). Among the DOS services supported are those concerning the time/date information in the control information area, the BIOS/DOS interrupt facilities, directory-related function calls (e.g., DIR, MKDIR, CHDIR, RENAME, and COPY), and keyboard facilities for retrieving scan codes and information from the keyboard status byte. In addition, some products allow the use of a programmable timer.

By *interactive editor,* we refer to an automatic re-invocation of the editor upon determination of compiler/run-time error. *Multiple windowing* refers to the ability to define different screen functions for the available window partitions. Screen control allows the user to configure the system for the desired video attributes beyond the DOS specification for video mode.

Six products support modularization of procedures. Module privacy is a technique whereby predicates are hidden from users, frequently to prevent name conflicts between modules. Worlds and theories are similar entities that are used to accomplish roughly the same thing. Technically, a world is a region within a database. One normally would use individual worlds to avoid backtracking through the larger database of which the world is a part. A theory is a database region which may involve the physical separation of the clauses into separate files.

Database indexing refers to the way in which the clauses are indexed and accessed. Clause indexing involves addressing a clause by its internal reference number. Hashing and B-trees increase the efficiency of searching. All of these features are extremely important for large clause sets.

The sub-category entitled *optimization* is a grab-bag of features which in one way or another relate to the efficiency of the implementation. A cyclic structure is created when a variable is unified with a term which contains that variable. The result is the generation of an infinite term as the unification repeatedly instantiates the term's variable with itself. It is not at all clear that the procedural interpretation of this phenomenon is consistent with the semantics of first order logic. Occur checks anticipate this behavior, but do so at considerable cost in efficiency. As a result, occur checking is not supported (as far as we know). A compromise is cyclic structure checking. In this case, the variable responsible for the infinite loop is returned in lieu of the infinite term. This surrogate does not appreciably decrease performance. Garbage collection control and stack control allow a programmer greater latitude in speed/space trade-offs. TRO stands for tail recursion optimization.

The system information features are useful for benchmarking and program development. DCG refers to the mechanism for translating definite clause grammars into PROLOG clauses. Finally, a ' + ' for structured programming indicates that such control structures as "if then ... else ...," "case," and so forth, are available.

We note that the number of built-in predicates specifically excludes a count of logical and arithmetic operators. Further, the numeric tally of the built-in predicates should be interpreted as an estimate of the number of substantially different predicates, rather than the total number. For example, since the distinction between "get0(term)" and "get0(handle,term)" is one of input type rather than functionality, both would be subsumed under one predicate. However, the capabilities of redirecting the standard input would be noted in the feature tables. Other cases of essentially duplicate functionality include predicates related to I/O, clause handling, formatting, string manipulation, and so forth. We believe that this "selective tally" approach provides a more reasonable first glance estimate of overall functionality than those which overlook the fact that some predicates are extremely narrow in scope, and that predicates are not distributed uniformly over the range covered in our classification.

One general consideration does not appear in the table. This concerns the issue of whether one of the products is a legitimate PROLOG. We do not enter into the controversy here beyond mentioning that Turbo PROLOG is a strongly typed language that does not support general unification. Further, it lacks the metalogical facilities normally associated with PROLOG environments. For further details on this issue, see Weeks and Berghel[10] and Pereira.[12] Additional discussion of Turbo PROLOG can be found in Rubin[13,14] and Shammas.[15]

BUILT-IN PREDICATES

The classification of predicates used here is an emendation of the taxonomy employed in Weeks and Berghel.[9] The scheme is somewhat arbitrary and is simply the approach to the classification we find convenient. We call attention to the fact that the categorization is intended only for ease of use. For example, creating a separate category for strings does not imply that strings are separate data structures. No predicate was counted unless it appeared in the documentation for the product. Since the tables are self-explanatory, we make only very general comments regarding anomalies within the classification.

LPA's micro-PROLOG is distinctively different in terms of built-in predicates. In this case, there are multiple program environments, each of which has its own set of predicates. The environments are SIMPLE, micro-PROLOG, and DECsystem-10. Both SIMPLE and micro-PROLOG use syntax based upon the Marseilles implementation, whereas DECsystem-10 is essentially the Edinburgh syntax. Further, as a simplified interactive version of micro-PROLOG, SIMPLE has its own character: it supports infix notation. This makes the classification difficult because the range of built-in predicates depends upon the environment.

Although SIMPLE and micro-PROLOG are compatible to the extent that any module written in SIMPLE can be included in micro-PROLOG, neither is completely compatible with the DECsystem-10 environment. To illustrate, one can access micro-PROLOG clauses from the DECsystem-10 mode, but not the converse. As a result, such features as DCG's, which are supported in the DECsystem-10 environment, are not available under micro-PROLOG. Thus, the question becomes one of which environment should be compared. Since the DECsystem-10 predicates are only a subset

From the collection of the Computer History Museum (www.computerhistory.org)

TABLE II—I/O predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
PROGRAM/CLAUSE I/O							
save ws by predicate(s)	-	+	+	-	-	-	-
delete ws by file	-	-	-	-	+	+	-
replace ws w/file	-	-	-	-	+	-	-
update file from ws	-	+	-	-	-	+	-
load/save binary image	+	+	+	-	-	-	-
load/save state	+	-	-	-	-	-	-
CHARACTER I/O							
get char from stream/file	+/+	+/+	-/-	+/+	+/+	+/+	+/+
get pr char (stream)	+	+	-	+	+	-	-
get w/o echo (stream)	+	+	-	-	-	+	-
skip to char (stream/file)	+/+	+/+	-/-	-/-	+/+	+/+	-/-
skip w/o echo (stream)	-	-	-	-	+	+	-
put char to stream/file	+/+	+/+	-/-	+/+	+/+	+/+	+/+
newline (stream/file)	+/+	+/+	-/-	+/+	+/+	+/+	+/+
newpage (stream)	-	-	-	+	-	+	-
write spaces (stream/file)	+/+	+/+	-/-	+/+	+/+	+/+	+/+
STRING I/O							
get string from stream/file	+/+	+/+	-/-	+/+	+/-	+/+	+/+
put string to stream	+	+	-	+	+	+	+
TERM I/O							
read term from stream/file	+/+	+/+	+/+	+/+	+/+	+/+	+/+
read token from str/file	+/+	-/-	+/+	+/+	+/+	+/+	-/-
read number from str/file	+/+	-/-	-/-	-/-	-/-	+/+	+/+
write to stream/file	+/+	+/+	+/+	+/+	+/+	+/+	+/+
write quoted to str/file	+/+	+/+	+/+	+/+	+/+	+/+	+/+
write ops prefix str/file	+/+	+/+	-/-	-/-	-/-	+/+	-/-
write formatted	+	+	+	+	+	+	+
declare operator	+	+	-	+	+	+	-
remove operator	+	+	-	+	-	+	-
get info about operator	+	+	-	+	-	-	-
define a prompt for I/O	+	-	-	-	+	-	-
direct file access position	+	+	+	-	+	+	+
fixed length file access	-	-	+	-	-	-	-
report on output environment	+	-	+	+	+	-	-

of Clocksin/Mellish, we evaluated micro-PROLOG. We emphasize that without complete compatibility, representing the product by an "inclusive-or" tally of each of the three environments would be misleading.

In a similar vein, M PROLOG has a distinctive way of supporting predicates. Some predicates, such as those for program/clause I/O and debugging and tracing, are supported only within the professional editor, PDSS. As a result, the tally of predicates refers only to those predicates in the language, although the features supported include those supported in PDSS as well. We believe this is the most reasonable way to describe M PROLOG.

With regard to program/clause I/O (see Table 2), the kernel is the pair of predicates which loads and stores a file (variations of consult and reconsult). However, the enhancements mentioned in Table 2 can save an enormous amount of work. One must remember that only consult and reconsult were present in the original PROLOG specification, so the variation between products is quite wide. For example, some products offer load options that are not cumulative and others use buffered I/O which is user-transparent.

Since the control predicates for success and failure are part of the language standard (such that it is), they are not included in the comparison (see Table 3). We note, however, that Turbo lacks the success predicate. Further, it is now quite common for products to include limited cuts (e.g., "snips"), which are useful but not part of the original language. In

TABLE III—Control predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
STREAM/FILE CONTROL							
create a file	+	+	+	+	+	+	+
open a stream/file	+/+	+/+	+/+	+/+	+/+	+/+	+/+
close a stream/file	+/+	+/+	+/+	+/+	+/+	+/+	+/+
temporary redir stdin	+	+	+	+	+	+	+
temporary redir stdout	+	+	+	+	+	+	+
turn on/off error calls	-	+	-	+	-	-	-
BACKTRACKING							
cut	+	+	+	+	+	+	+
repeat	+	+	-	+	+	+	-
logical set	+	+	+	+	+	+	-
explicit procedure call	+	+	-	+	+	+	-
special termination	+	+	+	+	+	-	+
number of solutions	+	-	+	-	+	+	-

Table 5, *full relational set* refers to the set of operators $\{<, >, <=, =, >=\}$ or their notational equivalents.

Structure manipulation (see Table 7) is important if one is to take full advantage of symbolic programming. Particularly important are such predicates as the ability to unify on arbitrary tree structures, decompose, compose, and convert between structures.

We also wish to note that, in contrast to earlier reports,[9,10,11] the present comparison indicates that a great deal of attention is being paid to extensions to the language. We believe that this reflects a desire on the part of the developers to establish PROLOG as a complete language environment rather than simply an experimental tool. To illustrate, the number of built-in predicates in the products under study that are not directly related to PROLOG typically constitute between 25 percent to 35 percent of the total.

PERFORMANCE CHARACTERISTICS

Traditionally, performance assessments fall into two categories. In some cases, the analysis is based upon an abstract model of the environment. Simulation and stochastic modeling illustrate this sort of evaluation. In other cases, the actual performance of the system in use is measured. These are usually called "benchmarks" or "workload models." In either case, one seeks to extract from the analysis some estimate of

TABLE IV—Term predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
CLASSIFICATION/CONVERSION							
is a variable	+	+	+	+	+	+	-
is a non variable	+	+	-	+	+	+	-
is an atom	+	+	+	+	+	+	-
is a number	+	+	+	+	+	+	-
is either atom or number	+	+	-	+	+	+	-
is a list	+	-	+	+	+	-	-
is quoted	-	-	-	+	+	+	-
is name	-	+	-	-	+	-	+
COMPARISON							
matching plus unification	+	+	+	+	+	+	+
does not match	+	+	-	+	+	+	+
equivalent	+	+	+	+	+	+	+
not equivalent	+	+	-	+	+	+	+
relational inequalities	+	+	-	+	+	+	+

From the collection of the Computer History Museum (www.computerhistory.org)

TABLE V—Arithmetic evaluation predicates and operators

Product:	P2	AR	LPA	MP	P86	VML	TUR
PREDICATES							
evaluate and unify	+	+	+	+	+	+	+
arithmetically equal	+	+	+	+	+	+	+
not arithmetically equal	+	+	+	+	+	+	+
full relational set	+	+	-	+	+	+	+
OPERATORS							
arithmetic operators	+	+	+	+	+	+	+
X**n	+	+	-	+	+	+	+
int(f or n)	+	-	-	-	+	+	-
float(f or n)	+	-	-	-	+	+	-
log2(X)	+	+	-	-	+	+	-
log10(X)	-	-	-	-	+	+	+
lognat(X)	-	+	-	-	-	-	+
abs(X)	-	+	-	+	+	+	+
round(X,N)	+	+	-	-	-	-	-
sqrt(X)	+	+	-	-	+	+	+
sin(X)	+	+	-	-	+	+	+
cos(X)	+	+	-	-	+	+	+
tan(X)	+	+	-	-	+	+	+
asin(X)	+	+	-	-	+	+	-
acos(X)	+	+	-	-	+	+	-
atan(X)	+	+	-	-	+	+	+
floor(X)	-	-	-	-	+	+	-
greatest integer	+	+	+	+	-	+	-
atoi(<ascii>,<int>)	+	-	-	+	+	+	+
stof(<ascii>,<flt>)	+	-	-	+	+	+	+
bitwise AND	+	+	-	+	+	+	+
logical AND	-	-	-	-	+	+	-
bitwise OR	+	+	-	+	+	+	+
logical OR	-	-	-	-	-	+	-
bitwise EXCL-OR	-	-	-	-	+	+	+
logical EXCL-OR	-	-	-	-	-	+	-
bitwise NEGATION	+	+	-	+	+	+	+
arithmetic NEGATION	+	+	-	+	+	+	+
n-bit shift(left)	+	+	-	+	+	+	+
n-bit shift(right)	+	+	-	+	+	+	+
random number	-	+	-	+	+	+	+
random seed	-	-	-	+	+	+	-
counter	-	+	-	-	-	-	-

TABLE VI—Database control predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
CLAUSE CONTROL							
list all clauses	+	+	+	+	+	+	-
list specified clauses	+	+	+	+	+	+	-
assemble/disassemble clause	+	+	+	+	+	+	-
add a clause to the database	+	+	+	+	+	+	+
remove:							
first clause for predicate	+	+	+	+	+	+	+
all clauses for predicate	+	+	+	+	+	+	-
report presence of predicate	+	+	-	+	-	+	-
TERM CONTROL							
record term	+	+	-	-	-	-	-
erase term	-	+	-	-	-	-	-
report term	-	+	-	-	-	-	-
replace term	-	+	-	-	-	-	-
manipulate reference #	-	+	-	-	-	-	-

TABLE VII—Structure manipulation predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
structure unification pred	+	+	-	+	+	+	-
get the Nth argument	+	+	-	+	+	+	-
convert list/structure	-	+	-	+	+	+	-
convert list/atom	+	+	-	+	+	+	-
convert list/string	+	-	+	+	+	-	-
length of a list	+	+	-	+	+	+	-
sort list	+	+	-	+	-	-	-
append	+	-	-	+	-	+	-

TABLE VIII—Set predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
set unification	+	+	-	+	+	+	-
findall/bagof	+	+	+	+	+	+	+
membership	-	-	-	-	-	+	-
intersection	-	-	-	-	-	-	-
union	-	-	-	-	-	-	-

TABLE IX—String predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
search for substring	+	+	-	+	+	-	-
get substring	+	+	-	+	+	-	-
get position of substring	+	+	-	+	+	-	-
get length of substring	-	+	-	+	-	-	-
get length of string	+	+	-	+	+	-	+
concatenate strings	+	+	-	+	+	+	+

TABLE X—Debugging and trace predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
trace program execution	+	+	+	+	+	+	+
trace single goal	+	+	+	+	+	+	-
trace multiple goals	+	+	+	-	+	-	-
report goals to be traced	+	+	+	+	+	+	-
goal ancestry	+	-	-	+	+	+	-

TABLE XI—Shell support predicates

Product:	P2	AR	LPA	MP	P86	VML	TUR
EXEC	+	+	+	+	+	+	+
report file existence	+	-	-	-	-	-	+
rename a file	+	+	+	-	+	-	+
erase a file	+	+	+	-	+	+	+
link files	-	+	-	-	-	-	-

system performance in terms of responsiveness, throughput, and cost.[16]

Ideally, the programs used in benchmarking are known *a priori* to be relevant to the intended application of the computer resource. From our experience, this ideal is seldom realized. Instead, general-purpose and "home-grown" programs which anticipate patterns of usage are used. Of course, if the anticipated patterns are unrealized, the benchmark results are likely to be unreliable. We mention this because we believe benchmarks are very coarse measurements; and, consequently, we encourage readers to take our results with a large grain of salt.

Benchmarks are not without value as long as their results are not misused. Misuse can result from misrepresenting the relevance of the test or by misinterpreting the results.[17] We propose a modest objective: we try to gain some general un-

From the collection of the Computer History Museum (www.computerhistory.org)

derstanding of the performance of the PROLOG products by running rather typical sorts of procedures, in fact, those procedures we use most often. As a result, our findings are biased toward our own interests in computational linguistics[18,19] and approximate string matching.[20,21] However, because the routines we used are mainstream, our results may be of interest to others.

In the interest of completeness, we refer the reader to the work of Wilk at Edinburgh.[22,23] Wilk's approach is completely different. His intention is to develop standard benchmark techniques for PROLOG environments, carefully selecting benchmarks so that the entire breadth of PROLOG functionality is measured. This is an ambitious project and worthy of continued attention, although we suspect no general agreement will be reached regarding the confidence level to assign to his tests.

We also call your attention to other PROLOG benchmark results appearing in the trade press,[15,24] which occasionally are at odds with our own.

BENCHMARK RESULTS

We begin the results discussion with an analysis of recursion limits. Because PROLOG is an ideal environment for recursion, it is natural to determine the cost of implementation. In microcomputer environments, memory consumption usually is more critical than processing speed. The part of memory most affected is the stack. Unless some optimization takes place, recursion may fill the stack with unnecessary latent calls. To reduce this problem, software developers implement such functions as structure sharing, garbage collection, and last call optimization. Because users typically have no way of knowing whether and to what extent optimization is present, empirical tests are useful.

We performed two separate recursion tests on each product. The tests were adapted from Covington.[25] The number of full recursions before failure (stack space exceeded) is presented in Figure 1. When possible, we defined the largest stack space possible in the environment file. Otherwise, default values were used.

Tail recursion (see Figure 2) should be more efficient because the recursion is invoked at the end of a goal set. For

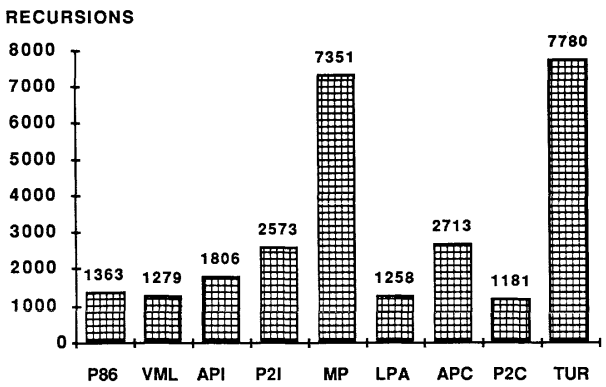

Figure 1—Recursions before failure (normal recursions)

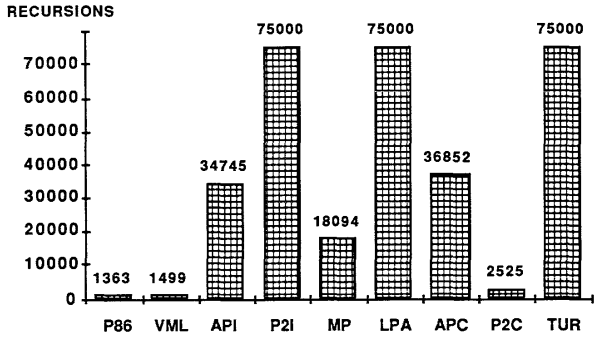

Figure 2—Recursions before failure (tail recursion)

interpreted PROLOG 2, micro-PROLOG, and Turbo, the tests were terminated at 75,000 recursions. Since these products claim optimized tail recursion, additional testing seemed unnecessary. In the case of Arity PROLOG, the failure was not a result of non-optimization; it was a result of the way that the counter represents large integers. Because there is no way of determining the upper bound on recursions without a counter, Figure 2 provides the actual results without adjustment.

The next benchmarks deal with string operations, and are, for the most part, standard procedures defined in Clocksin and Mellish.[7] The values are presented in terms of run times. The results of all tests appear in Figures 3 and 4. Naive reverse is a variation on the *de facto* standard benchmark for PROLOG. Despite its frequent use, it has shortcomings: it can overstate the efficiency of the implementation.

The last test (see Figure 5) is a general benchmark which is supposed to have its origins in ICOT. The code fragment is as follows:

```
tak(X,Y,Z,Z):-X = <Y,!.
tak(X,Y,Z,R):-
  tak1(X,Y,Z,R1),!,
  tak1(Y,Z,X,R2),!,
```

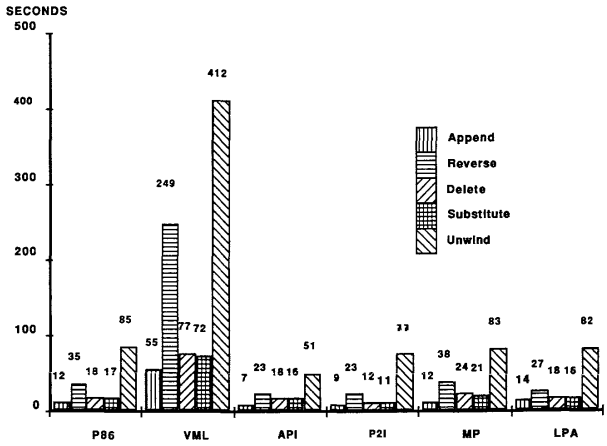

Figure 3—String functions (interpreters)

From the collection of the Computer History Museum (www.computerhistory.org)

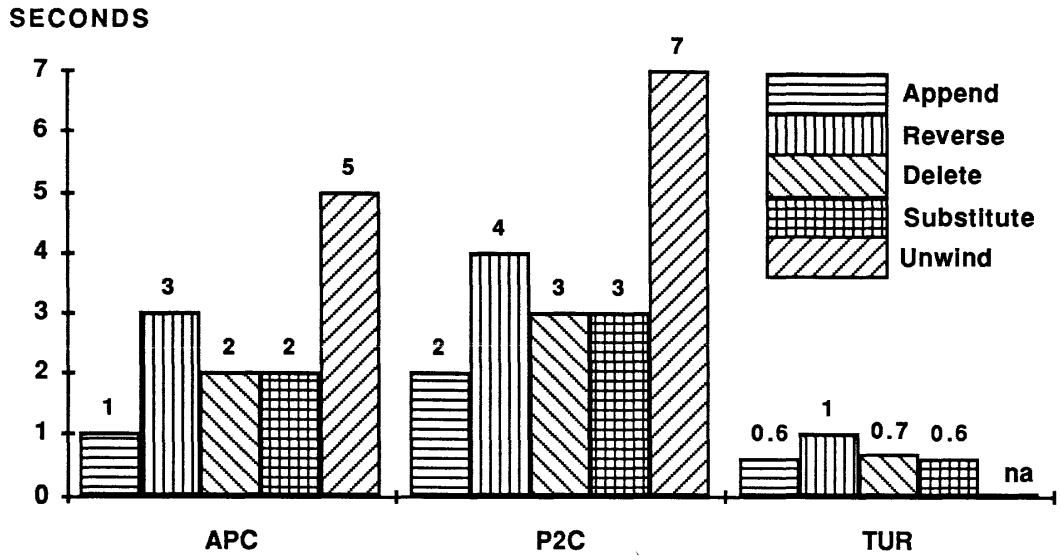

Figure 4—String functions (compilers)

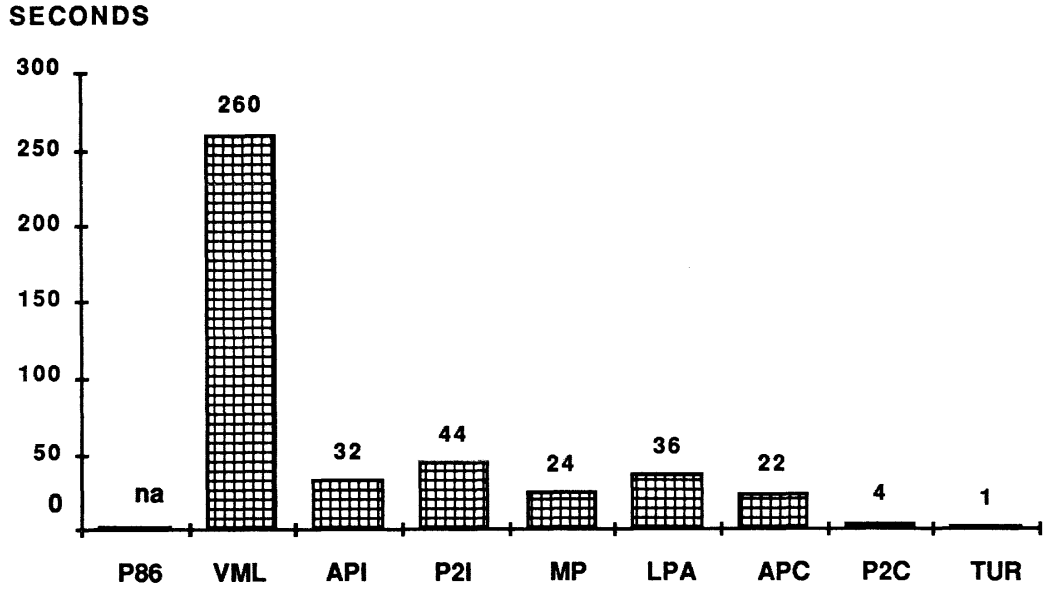

Figure 5—Agglomerative benchmark

```
  tak1(Z,X,Y,R3),!,
  tak(R1,R2,R3,R),!.
tak1(X,Y,Z,R):-X1 is X-1,
  tak(X1,Y,Z,R).
?-tak(12,8,4,N).
```

The test was provided by Robert Morein of Automata Design Associates. It is an agglomerative measure which attempts to assess the overall strengths of the products. We note that PROLOG 86 + would not run the program, presumably due to inefficient memory reclamation.

For additional details on these and other benchmarks, including a listing of the clause sets, see Berghel, Stubbendieck, Traudt.[11]

CONCLUSION

As increasing attention is paid to PROLOG, growing numbers of researchers and system developers wish to avail themselves of PROLOG implementations. For many, microcomputer-based products offer the most cost-effective way to exploit logic programming. This paper is intended as a general overview of these products so that interested parties may select the product most consistent with their needs.

From the collection of the Computer History Museum (www.computerhistory.org)

ACKNOWLEDGEMENTS

The classification scheme used in this paper is a result of collaboration with J. Weeks. The benchmark results are taken from earlier work with G. Stubbendieck and E. Traudt. We wish to thank L. Baxter, P. Gabel, J. Grayson, and R. Morein for many useful comments and criticisms.

REFERENCES

1. Mota-Oka, T. (ed.) *Fifth Generation Computer Systems* (Proceedings of the International Conference on Fifth Generation Computer Systems). Amsterdam: North Holland, 1982.
2. Kowalski, R. "Search Strategies for Theorem Proving." In B. Meltzer and D. Michie (eds.) *Machine Intelligence* (Vol. 5). New York: Edinburgh University Press, 1969.
3. Kowalski, R. "And-Or Graphs, Theorem Proving Graphs and Bi-directional Search." In B. Meltzer and D. Michie (eds.) *Machine Intelligence* (Vol. 7). New York: Edinburgh University Press, 1972.
4. Kowalski, R. "Predicate Logic as a Programming Language." *Proceedings of IFIP 74,* 1974, pp. 569–574.
5. Colmerauer, A., H. Kanoui, R. Pasero, and P. Roussel. "Un Systeme de Communication Homme-Machine en Français." Report, Group Intelligence Artificielle, University d'Aix Marseilles, 1973.
6. Colmerauer, A. "Les Systèmes-Q ou un Formalisme pour Analyzer et Synthesiser des Phrases sur Ordinateur." Report #43, Department d'Informatique, Universite de Montreal, 1973.
7. Clocksin, W. and C. Mellish. *Programming in PROLOG.* New York: Springer-Verlag, 1981.
8. Kowalski, R. "Algorithm = Logic + Control." *Communications of the ACM,* 22 (1979), 7, pp. 424–436.
9. Weeks, J. and H. Berghel. "A Comparative Feature-Analysis of Microcomputer PROLOG Implementations." *SIGPLAN Notices,* 21 (1986), 2, pp. 46–61.
10. Weeks, J. and H. Berghel. "Turbo + PROLOG." *Bulletin of the IEEE Computer Society Technical Committee on Personal Computing* (October 1986), pp. 1–7.
11. Berghel, H., G. Stubbendieck, and E. Traudt. "Performance Characteristics of Microcomputer PROLOG Implementations." *Proceedings of the 1986 ACM Sigsmall/PC Symposium on Small Systems,* 1986, pp. 64–71.
12. Pereira, F., Bix Communication: ask.experts, message #17 (Electronic Mail). McGraw Hill/Byte Information Exchange, June 8, 1986.
13. Rubin, D. "Turbo PROLOG: A PROLOG Compiler for the PC Programmer." *AI Expert,* Premier Issue (1986), pp. 87–98.
14. Rubin, D. "Inside Turbo PROLOG." *Computer Language* (July 1986), pp. 23–28.
15. Shammas, N. "Turbo Prolog." *Byte* (September 1986), pp. 293–295.
16. Muntz, R. "Performance Measure and Evaluation." In A. Ralston and E. Reilly, Jr., *Encyclopedia of Computer Science and Engineering.* New York: van Nostrand Reinhold, 1983.
17. Fleming, P. and J. Wallace. "How Not to Lie with Statistics: the Correct Way to Summarize Benchmark Results." *Communications of the ACM,* 29 (1986), 3, pp. 218–221.
18. Berghel, H. and J. Weeks. "On Implementing Elementary Movement Transformations with Definite Clause Grammars." *Proceedings of the Fifth Phoenix Conference on Computers and Communications,* 1986, pp. 366–370.
19. Berghel, H. "Extending the Capabilities of Word Processing Software through Horn Clause Lexical Databases." *AFIPS Proceedings of the National Computer Conference* (Vol. 55) 1986, pp. 251–257.
20. Berghel, H. "Crossword Compilation with Horn Clauses." *The Computer Journal* [in press].
21. Berghel, H. "A Logical Framework for the Correction of Spelling Errors in Electronic Documents." *Information Processing and Management* [in press].
22. Wilk, P. "The Production and Evaluation of a Set of PROLOG Benchmarks," Artificial Intelligence Applications Institute Report #AIAI-PSG51, University of Edinburgh, 1986.
23. Wilk, P. "PROLOG Benchmarking," Artificial Intelligence Applications Institute Report #AIAI-TR-14, University of Edinburgh, 1986.
24. Wong, W. "PROLOG—A Language for Artificial Intelligence." *PC Magazine* (October 14, 1986), pp. 247–263.
25. Covington, M. "Programming in Logic—Part 2." *PC Tech Journal* (January 1986), pp. 145–155.

From the collection of the Computer History Museum (www.computerhistory.org)

From the collection of the Computer History Museum (www.computerhistory.org)