Institute for Mathematical Studies in the Social Sciences
Stanford University

MEMORANDUM

Date: July 26, 1983
To: Peter Hirsch
From: Tryg Ager, James McDonald
Subject: Yorktown LISP

This memorandum conveys the results of our experimentation with Yorktown LISP during one week in July, 1983.

We became familiar with the Yorktown LISP system, ran timing tests to compare its performance to PSL, and considered it both from a technical and a user point of view. The most important of our findings is the comparative timing data, where PSL is faster in execution of both interpreted and compiled code. The second important way PSL differs from Yorktown LISP is in strategies employed to implement and maintain the system. PSL is written in itself and uses an implementation technology based on compilation to abstract machines; Yorktown LISP uses conventional methodology where a handcoded assembly language kernel is supplemented by derived functions written in LISP.

1. Performance Data

The performance data presented here is from a series of "spectral tests" designed to isolate and evaluate performance on individual components of a LISP system, such as consing, evaluation, different kinds of function calls, arithmetic, and pointer manipulation. These tests are standardly used to compare LISPs on different machines and are written in straightforward code, so they were easily converted to Yorktown LISP.

Both PSL and Yorktown LISP have special fast arithmetic functions. We were careful to use fast versions of functions in Yorktown LISP whenever PSL did so. Although PSL does not have fast CAR and CDR operations, we used Yorktown's QCAR and QCDR. We set Yorktown compiler switches for non-interruptible code and to the maximum optimization level to get the fastest code the Yorktown compiler could deliver.

The results are given in Table 1 at the end of this report. They show that compiled PSL and Yorktown LISP are comparable on list traversal and list creation. PSL is four to five times faster than compiled Yorktown LISP on the arithmetic and recursion tests.

PSL is thirty times faster on function calls done via EVAL and APPLY; four to five times faster on normal function calls.
as in APL, long LISP programs tend to have many calls to small functions.

The PSL system used for these tests is a prototype whose code is entirely machine-independent. No optimizations of the code generated by the machine-independent compiler has been done for the IBM implementation. This prototype PSL, when run on a 3081, is already 10% faster than PSL on the CRAY-1. In situations where speed is critical and a bottleneck can be identified, PSL makes it easy to hand-optimize selected functions. Therefore the final implementation of PSL should be

somewhat faster than the times reported here.

Performance of Interpreter.

We also compared the performance of the PSL and Yorktown interpreters. When we tried to run timing tests of interpreted code we found execution of code by the Yorktown interpreter to be an order of magnitude slower than interpreted PSL code. Interpreted times are in Table 2 at the end of the report.

The poor performance on interpreted code is a serious weakness of Yorktown LISP, since most LISP program development is done with interpreted code.

2. Portability.

PSL stands for "Portable Standard LISP." With respect to portability, there are fundamental differences between Yorktown LISP and PSL. While it is true that a relatively small handcoded interpreter can be recoded more easily than a large interpreter, and therefore is "more portable;" this is a strategy which was tried for years by the developers of PSL and abandoned in favor of PSL techniques.

PSL is written in PSL, and was designed with the DEC-20 and VAX architectures in mind. Nevertheless the porting to 370 architecture resulted in performance superior to handcoded LISP systems. This result is contrary to popular wisdom which says that writing in assembly code is the way to achieve efficiency.

The porting of PSL to other machines by Utah and HP has also indicated that excellent performance along with extremely rapid conversion to new hardware can occur with PSL. It is important to remember that the compiler is identical for all implementations. Porting to new hardware is a matter of mapping instruction sets, not writing a language translator or solving algorithmic
PSL design changes and maintenance flow down to all implementations. Since the language ports so thoroughly, applications can be absolutely identical on different machines.

We do not see how a handcoded interpreter can contend with the rapid changes in hardware or operating systems likely to occur in the future.

PSL is a powerful systems development tool in its own right. It would be possible to alter PSL code generators so a half-bootstrap implementation of a Common LISP could be done. PSL has already been used to half-bootstrap PSL itself onto half a dozen different machine architectures and/or operating systems.

3. Comparison with Other LISP Dialects.

Yorktown LISP is an eclectic system, having features from both INTERLISP and MACLISP and others all its own. There are some aspects of Yorktown LISP which we perceive as fundamental difficulties, others which are minor flaws, and other features of great interest.

a. As with INTERLISP, functions are edited, maintained, and filed primarily as list structures from within LISP. MACLISP and its derivatives, including the LISP machines, use general-purpose editors and a traditional concept of file.

b. Like INTERLISP it completely implements the theoretical concept of LISP (as defined by the lambda calculus) including the

FUNARG idea. The cost of FUNARG, which complicates function calling, is considerable as is shown in the tables of performance times on the Eval tests and the GTSTA-G0 and GTSTB-G1 tests, each of which tests a different kind of LISP function call. No other popular LISP dialects emphasize these extended control structures.

c. Unlike Franz LISP and ZETA LISP, Yorktown does not have "flavors" or other object-oriented features which promise to become prominent in future LISP applications. Object-oriented programming captures important semantic intuitions, just as structured programming captures key algorithmic intuitions.

d. It does not seem to have a package system orientation of MACLISP where a package implies its own name space. This is a feature advocated by Common LISP and found in PSL. It prevents variable and function name clashes when code from various sources is combined in one complex application.

e. Yorktown LISP is pervaded by slightly different versions of essentially the same function. We think that this makes performance considerations intrude too much into the domain of high-level programming. For example CAR and CDR have fast varieties with different names. We think it is wrong to have to change the name of such key functions to get improved performance. A compiler switch is a better way.

f. The break and trace tools are quite adequate. The Break feature gives useful information in readable form and was very helpful whenever we used it. We did not stress-test the break package, so it is not clear what happens in extreme situations such as stack overflow.

g. The editing environment, LEDIT, answers directly to current interests in programming environments. First of all, we were disappointed to find that LEDIT was not a full-screen editor. Interaction with LEDIT had the feeling of a line editor with extended output-only capabilities. EMACS, XEDIT, and other modern editors allow both command input and direct alteration of the displayed text.

Second, we believe that LEDIT, in emphasizing the editing of LISP structure as such, is both too restrictive and not in accord with the preferences of the LISP community. In point of fact, most LISP programmers use EMACS as their editing tool, even when structure editors are available.

The editing of LISP code raises interesting issues because there are always two perspectives on any LISP structure: as text composed of characters, words, lines, and paragraphs or as structure composed of atoms, lists, and functions. For example,

```
(LIST (FN1 ARG1) (FN2 ARG2) (FN3 ARG1 ARG2))
```

can be viewed as eight words plus punctuation, as a four-element list, or as a three-argument function call.

Because of this duality of representations, it is desirable to be able to manipulate it as structure per se, or as just characters. So the best environment will have both text and structure editing capabilities, freely intermixable. Currently, only the LISP Machines

(LM2 and Symbolics) provide the full power of both approaches. EMACS gives strong text and limited structure capability. Yorktown gives elaborate structure editing but minimal text capability.

h. Documentation. We were able to make sense of the documentation, despite the author's wry disclaimers, but recommend a thorough reorganization and rewriting to make the documentation reasonably accessible to a broad audience. Much attention should be paid to the problem faced by a first-time user of Yorktown LISP. In particular, guidance on the use of files and I/O facilities is obscure. Because Yorktown LISP uses neither standard CMS file concepts nor traditional LISP I/O functions, it is especially important that the documentation contain examples showing how to read and write files, and that all the relevant functions and parameters are illustrated.

In the documentation of individual functions, the organization seems haphazard, in the sense that esoteric and extended-capability functions are intermixed and not distinguished from the essential kernel of LISP routines. For example, the description of CASEGO, a derivative function which emulates the computed goto of FORTRAN and is unique to Yorktown LISP, is inserted between the descriptions of COND and AND.

In general, the documentation is organized as a reference manual. It lacks a users guide and introductory tutorial material normally associated with highly interactive systems. Much theoretical material is at the beginning of the manual, but properly belongs in an accompanying theoretical document.

i. Human factors. We noticed several things about the user interface of Yorktown LISP.

There seems to be no end-of-line concept in the LISP reader. It is necessary to type FOO <space> <enter> to get the value of FOO.

The minimum response time in the editor is very slow. E.g., evaluating NIL takes 5 seconds. Things that simple should happen instantaneously.

Useful constructs such as DO, FOR, REPEAT, and WHILE are not found in the manual.

The Yorktown stringizer (') and quotizer (") are the reverse of usual LISP conventions.

There should be a simple, unparameterized function for reading a file of LISP functions prepared straightforwardly in XEDIT.

The program starts slowly and destructively alters the virtual machine environment.

4. Comparison of PSL and Yorktown LISP.

We conclude with some additional points of comparison between PSL and Yorktown LISP, with attention to the acceptability of PSL to the academic community.

a. A LISP which will be used in teaching situations should be compact and efficient. On the small 4300 series machines, frequently found in academic settings, anything above 1 megabyte is not feasible. Even on large machines, timesharing constraints usually require restrictions on the size of user programs.

PSL on IBM has been engineered to be a compact, streamlined system. The PSL interpreter and compiler can run in about one megabyte. Yorktown LISP as received would not run in less than three megabytes. There is also evidence that Yorktown LISP uses space less efficiently than PSL. For example, the performance tests were run in PSL using a heap with 40,000 free items and required four garbage collects. The same series in Yorktown LISP with 350,000 free items in the heap required seven garbage collects.

b. There already is a nucleus of a PSL community on IBM systems. At this time the most widely used LISP on IBM is Utah Standard LISP which supports the REDUCE algebra system. REDUCE is an essential tool for many research physicists and applied mathematicians. The Stanford 1980 revision of Utah Standard LISP is in regular university use, both standalone and in a sophisticated CAI application. PSL is a direct descendant of UTAH Standard LISP and is upward compatible with it.

c. There is qualitative evidence of the acceptability of PSL for various academic, research, and commercial purposes. Its use at Utah and Stanford is established. It is the distribution base for REDUCE on VAX and DEC-20. It will be be used by several hundred MIT students this fall. A program in PSL will be used to teach logic to about 300 Stanford undergraduates each year. There is lively interest in PSL at both HP and Apollo.

d. PSL has some untapped teaching advantages. Since it embodies a high-level systems-oriented sub-dialect (SYSLISP) it can be used for instruction about machine architectures, memory management systems, and instruction set design.

e. Because of PSL's portability, new work produced in PSL on IBM systems can migrate to the established AI and LISP community. Migration of existing non-PSL to IBM PSL is more problematic, requiring compatibility packages. Where Yorktown LISP necessarily requires compatibility or emulation for migration in either direction, PSL stands a reasonable chance of becoming a transparent system.

In summary, we have shown that PSL is faster than Yorktown LISP. We have argued that it is better. We attribute the superiority of PSL to its clean, coherent, and modern design principles. The fact of the matter is that a machine-independent design is outperforming handcoded counterparts on the IBM architecture.

Table 1: Performance of Compiled Code for PSL and Yorktown LISP

Spectral Test	PSL	Yorktown

EMPTYTEST-10000	.024	.098
GEMPTYTEST-10000	.371	.878
CDR1TEST-100	.577	.469
CDR2TEST-100	.342	.473
CDDRTEST-100	.205	.272
LISTONLYCDRTEST1	1.607	4.176
LISTONLYCDDRTEST1	2.330	4.873
LISTONLYCDRTEST2	2.985	4.153
LISTONLYCDDRTEST2	3.726	4.666
REVERSETEST-10	.380	.174
MYREVERSE1TEST-10	.372	.329
MYREVERSE2TEST-10	.352	.324
LENGTHTEST-100	.596	.443
ARITHMETICTEST-10000	1.117	5.097
EVALTEST-10000	4.254	120.849
TAK-18-12-6	.772	3.360
GTAK-18-12-6	2.577	9.861
GTSTA-G0	1.938	62.553
GTSTA-G1	2.008	63.177

Table 2: Performance of PSL and Yorktown Interpreters

Since the interepreted tests take so much longer, the parameters of the test functions were reduced by one and sometimes two orders of magnitude. Because of this no attempt should be made to compare compiled times above with the interpreted times below.

Spectral Test	Interpreted PSL	Interpreted Yorktown
EMPTYTEST-1000	1.137	11.233
GEMPTYTEST-1000	1.164	11.305
CDR1TEST-10	1.181	16.195
CDR2TEST-10	1.183	12.940
CDDRTEST-10	.682	8.820
LISTONLYCDRTEST1	.986	12.160
LISTONLYCDDRTEST1	1.030	15.853
LISTONLYCDRTEST2	1.125	15.298
LISTONLYCDDRTEST2	1.039	18.584
REVERSETEST-10	.051	.196
MYREVERSE1TEST-10	2.023	29.186
MYREVERSE2TEST-10	2.050	25.835
LENGTHTEST-100	.261	1.243
ARITHMETICTEST-100	1.184	10.592
EVALTEST-500	.921	12.190
TAK-12-6-3	1.477	6.721
GTAK-12-6-3	1.500	6.734
GTSTA-G0	1.905	21.395
GTSTA-G1	2.212	21.308