<HJOURNAL>10042.NLS;1, 11-APR-72 3:48 XXX ; .HJOURNAL="LPD 9-APR-72 10:07 10042"; Title: .HED="Full tilt into the heffalump trap"; Author(s): L. Peter Deutsch/LPD; Distribution: James G. Mitchell, William H. Paxton, L. Peter Deutsch, Butler W. Lampson, Charles H. Irby/JGM WHP LPD BWL CHI; Sub-Collections: NIC; Clerk: LPD; .IGD=0; .SNF=72; .MCH=65; .TABSTOPS=8,16,24,32,40,48,56,64; .PGN=-1; .SCR=2; .PES;

This note summarizes some serious dissatisfactions with MPS which I feel have gotten shouted down or otherwise ignored in the course of recent discussions.

If they sound like Larry Barnes type gripes, that may well result from my belief that he was right a lot of the time and we were wrong.

I believe that there is nothing we can do about the mess we have gotten ourselves into, and that the only way out is to work like hell and hope we can avoid another SPL debacle by sheer energy and numbers, since I don't believe we are doing much of anything more sensibly.

We always re-invent the wheel "just one more time, then it'll be right."

We essentially went back to the machine level to implement MPS, rather than starting with an established system.

We could have lifted the BLISS compiler, used SIMULA 67 as a semantic base, or written an interpreter in LISP.

Although Tree-Meta and L10 have helped us write our own code, we haven't used any part of them or any part of any other existing software.

If there is a successor to MPS, or even if we start to develop application systems within MPS like xNLS, I am sure we will find ourselves writing the same routines all over again.

For example, there are already routines essentially duplicated between the runtime and the debugger, because the calling conventions (and more importantly, the data environment) aren't quite similar enough to share them.

In xNLS, we are almost certainly not going to be able to implement the marker table or a statement name directory with the "standard" symbol table package, because the resulting tables have to mesh with the structure of the NLS file in a way which we can't arrange.

In contrast, I constantly re-use LISP functions I have written because their interfaces are so simple.

We never seem to learn that user convenience (as in LISP) is worth a lot more than purity of package boundaries during the intensive developmet phase.

We have done the project in entirely the wrong order even though we should have known better.

We wrote a compiler first before an interpreter.

Interpreters are much easier to write than compilers, and they define the semantics of the language much more directly.

We could have brought up a running system in a month if we had been willing to consider implementing it initially as an interpreter in LISP.

We have done almost all of the implementation without anything resembling a coherent description of the semantics.

The system is so large and complex that we should have learned from SPL that the plunge-ahead approach which worked for QSPL wouldn't work here.

Starting with an interpreter would have allowed much easier experimentation with language and semantic changes, and would have given us a cleaner semantics when we were done.

We let efficiency considerations influence the design of the language, at the expense of user convenience.

For example, it seems clear already that the compiler and the debugger will have to be monoliths at the level of 20-40 pages of code because it is so hard to share data between modules.

The INCLUDE philosophy is unconvincing as an efficiency argument, since it is also clear that incremental compilation is a mirage: MPS is no better off than BCC SPL, where people resorted to all sorts of grotesque kludges to avoid changing INCLUDE modules.

The present compiler produces 45 object instructions per second of compute time; this means that a 500-instruction module (3-5 source pages) will compile in 1 to 1 1/2 minutes of real time.

Smaller modules are impractical because of the inconvenience of sharing environment.

The modularized compiler is certain to run significantly slower.

Are YOU willing to wait several times that long after every change in a DATA module?

Remember also that although the design of BCC SPL was heavily influenced by the desire to provide incremental compilation "later" (and some very expensive decisions were based on this), we had so much trouble tracking down the myriad obscure bugs in the compiler, and further consideration indicated that we would have to add so much more information to the permanent data structures, that the idea was essentially abandoned.

Although the more systematic design of the MPL compiler gives us some hope that the bugs won't be so obscure, the complexity of the code generators leads me to believe that we just haven't uncovered many of them yet.

Our desire for fast access via base pointers made us reject the possibility of using something like the LISP A-list as the semantic model, which would have made sharing very easy, and then backing off from that.

We let considerations of getting a working system for other people by a fixed date seriously distort the time organization of the project.

In fact, a good part of the reason for the six-month overrun was that we saw that our initial ideas wouldn't stand up.

Under time pressure we are now going ahead and implementing things we KNOW don't stand up logically.