"... but will RISC run LISP??"
(a feasibility study)

Carl Ponder

University of California
Department of Electrical Engineering & Computer Sciences
Computer Science Division

ABSTRACT

The Berkeley RISC microprocessor, developed under the direction of David Patterson & Carlo Sequin [1], is targeted for efficient execution of C programs. The architecture has competed successfully with existing systems such as the Vax-11/780 and MC68000. A major question about such a reduced, targeted architecture is how well it extends to other languages. An important language in symbolic computation is Lisp. Lisp is a functional language which has little in common with the standard block structured languages, such as C. This has led to the often-asked question – "will RISC run Lisp?".

The purpose of this paper is to explore the feasibility of a LISP system running on RISC. The major parts of this include a look at the behavior of large-scale "typical" Lisp programs, and an examination of current LISP implementations.

May 11, 1983

The work reported herein was supported in part by Defense Advance Research Projects Agency (DoD) ARPA Order No. 3803, Monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251, and the U.S. Department of energy under Contract DE-AT03-76SF00034, Project Agreement DE-AS03-79ER10358.

"... but will RISC run LISP??"
(a feasibility study)

Carl Ponder

1. Introduction

Lisp is the second oldest "high-level" language in popular use. It was designed to perform symbolic manipulation, particularly for problems in artificial intelligence. The main features of Lisp are its simple structure, extensibility, and the equivalence of programs and data. The language has been implemented on many general-purpose machines (IBM 370, PDP-10, VAX), as well as a few special purpose ones (CDC 7600, CADR). Several "high-level language" computers have been designed to run Lisp, interpreting Lisp instructions directly in the microcode; these have received widespread attention in the computer industry.

A new perspective on "high-level language" computers has been popularized with the RISC I design; a simple instruction set for high-speed execution is combined with a radical "register window file" for minimizing procedure call overhead. The exclusory use of high-level languages on the machine allows the compiler to hide from the user any complicated or counter-intuitive properties of the underlying architecture.

These conflicting approaches to high-level language architectures has been characterized as the RISC-CISC controversy, RISC standing for Reduced Instruction Set Computer, and CISC standing for Complex Instruction Set Computer. On the RISC side is the Berkeley RISC I, the IBM 801 [2], and the Stanford MIPS [3] processor; the projected performance of the final Berkeley RISC is competitive with modern general purpose processors such as the VAX 11/780 and Motorola 68000. Two notable CISC processors are the Intel iAPX-432 and the MIT SCHEME chip; the Intel 432 has shown feeble performance, and the SCHEME chip will be examined later on in a Lisp perspective to the RISC-CISC controversy. In the meantime I will study the feasibility of Lisp on the Berkeley RISC processor.

2. Why Lisp is not like C

In the next section, we will look at the RISC architecture and estimate how useful it would be for the properties of Lisp; to do this we must explore the differences between Lisp and C.

Table 1 contrasts the features of the two languages. Lisp is intended to exhibit the full generality and flexibility available to interpreted languages, while C is designed for efficient compilation and fast execution. For example, dynamic scoping, typeless variables, and dynamic storage allocation are relatively easy to implement in an interpreter, whereas static scoping, strong typing, and iterative control constructs are well adapted to compiletime semantics checking and efficient object code generation.

Table 1 -- Lisp vs C

	Lisp	C
1	Recursion - based control structures	Iterative control structures
2	Dynamic scoping	Static scoping
3	call-by-value (lambda), closure or funarg supported parameters	call-by-value
4	typeless variables, explicit checking required to determine types	loose typing, instruction traps
5	runtime support important for space allocation	system primitives available: io's, page allocation
6	executable data, requiring presence of interpreter	strictly compiled
7	operators correspond to function calls in interpreted code, often in compiled code as well. Exceptions are simple control structures and logical operations.	operations closely related to features of machine language
8	lists as fundamental data structure. parameters are always pointers to objects, requiring some degree of fetching.	words as fundamental data type. locality of variable/parameter references.
9	exotic procedure exits cause variable pops of activation records - throw, return, nonlocal goto, etc.	canonical escapes from control structures -- break, return, exit

Lisp can be fairly efficient as a compiled language; some Lisp dialects run competitively with Pascal and C for given benchmarks (Table 6). This efficiency is necessary to obtain tolerable performance from large Lisp systems. Simple extensions to the language make compilation easier, such as optional declarations; furthermore, clever compilers manage to replace inefficient constructs with more efficient ones (such as replacing recursion with iteration).

Many of the differences listed in table 1 have little impact on the performance of compiled Lisp; here I address each of them:

(1) Iterative control structures in Lisp, which are defined using equivalent recursive structures, actually map into iterative forms in compilation. Furthermore, recursion is in many cases transformed into equivalent iteration.

(2) control-flow analysis at compilation can determine whether or not variables must remain dynamically scoped. Some Lisp dialects specify static scoping (NIL, Scheme, and the new Common Lisp), while others consider it the default in compiled modules (UCI and Franz Lisp). Dynamic scoping is then made available through declarations.

(3) Closures and funargs were not used in any of the programs studied here (Franz, GLEAN, Liszt, PHRAN, and Vaxima). An object-oriented style of Lisp programming may use them heavily.

(4) Using typed segments, as in Franz Lisp, typechecking is a simple operation -- shift the address and load from a type table. Other schemes use typed pointers, when only part of an address field is used; this can be done in one operation, although it is only possible on machines with subfield addressing mechanisms.

(5) The memory manager is necessary for Lisp. Making it as efficient as possible is important. Garbage collection is largely a matter of linked list and bit operations.

(6) Calls to "eval" may be faster if the Lisp interpreter is microcoded, but such implementations tend to run much slower than compiled Lisp. The tradeoff involved depends upon the ratio of time spent in the compiled code vs. the amount of time spent in the interpreter. Macsyma, for example, spends most time doing list operations when in the kernel.

(7) This merely suggests that a Lisp program will make more procedure calls than a C program for the same computation. In some cases the extra routines can be expanded in-line for maximal efficiency, but this may cause large object files to be created. The procedure call overhead is often minor in relation to the operations contained within a function.

(8) The implied memory overhead is a very important point. I shall take this up in the next section. It is worth noting that the memory speed of a machine must be fast to guarantee fast list operations.

(9) Table 2 shows the frequency of occurrence of exotic functions exits. Reasonably inefficient implementations should be tolerable.

Table 2 -- frequency of exotic returns in Lisp

contrived examples	calls/returns	exotic returns	ratio
throw-catch (compiled)	812	101	12%
goto (interpreted)	3644	200	5%
real examples			
PHRAN	139484	0	0%
Liszt	392384	0	0%
GLEAN	1649	0	0%

We see, then, that Lisp can be like C in such things as scoping and control structures. In some places where they differ, such as pointer manipulations and typechecking, the operations are simple enough to be performed efficiently on most machines. Use of other features, such as funargs and eval, are a matter of style; systems inefficiently supporting them can competitively execute a wide class of Lisp programs.

3. The C machine as a Lisp machine

In this section I will address the problems involved in an efficient Lisp implementation on RISC. Three questions are of importance here:

Is the memory speed of RISC sufficient for list processing?

Is the reduced instruction set capable of supporting Lisp operations?

And will the register window scheme succeed in reducing procedure-call overhead?

The third question is defered until the fifth section.

Table 3 shows the timings on several C-coded benchmarks, executing on different machines. In all but one case, RISC I is favored, however little. The linked-list, bit-test, and Ackermann benchmarks represent cases we would expect to appear in a running Lisp system – linked-list operations as in memory management and structure manipulation, bit manipulation as in typechecking and storage marking, and excessive procedure calls as might occur in interpreting Lisp or making kernel calls from compiled code. Furthermore, the slowest operation (byte manipulation) is not a major part of Lisp, so the RISC architecture appears to support the demands of Lisp.

Table 3.
C Benchmarks: RISC I Execution Time and RISC I Performance Ratio

	RISC I	68000	Z8002	VAX-11/780	11/70	C/70
BENCHMARK	msecs	Number of Times Slower Than RISC I				
E - string search	.46	2.8	1.6	1.3	0.9	2.2
F - bit test	.06	4.8	7.2	4.8	6.2	9.2
H - linked list	.10	1.6	2.4	1.2	1.9	2.5
K - bit matrix	.43	4.0	5.2	3.0	4.0	9.3
I - quicksort	50.4	4.1	5.2	3.0	3.6	5.8
Ackermann(3,6)	3200	—	2.8	1.6	1.6	—
recursive qsort	800	—	5.9	2.3	3.2	1.3
puzzle(subscript)	4700	—	4.2	2.0	1.6	3.4
puzzle(pointer)	3200	4.2	2.3	1.3	2.0	2.1
sed(batch editor)	5100	—	4.4	1.1	1.1	2.6
towers Hanoi(18)	6800	—	4.2	1.8	2.3	1.6
Average±S.D.		3.5 ± 1.8	4.1 ± 1.6	2.1 ± 1.1	2.6 ± 1.5	4.0 ± 2.8

In [4], Fateman asserts that in Lisp programs, memory operations are the dominant factor; the performance of Lisp on a given machine is bounded by its ability to do them quickly. Figure 1 shows a C-coded "pseudo" benchmark to measure the memory speed of a given system; the results for several machines appear in table 4. The memory speed of RISC compares favorably with the others. The Macsyma benchmarks seem to agree, except in the case between the CDC 7600 and the KL-10. Here Fateman suggests three contributing factors:

Figure 1 -- the c-coded PSEUDO benchmark

```
int h[1000], j[1000], k[1000];

main()
{
  register int i;
  register int *hp, *kp;
  int *jp;
  int tv1[6], tv2[6];

  for (i=1; i<=1000; i++) {
        h[i] = 0;
        k[i] = i;
        j[i] = i+1;
  }
  h[1000] = 1;
  times(&tv1);
  hp = h; jp = j; kp = j;
  i = 1;
  while (hp[kp[i]] != 1) {
        hp[kp[i]] = hp[i];
        i = j[i];
  }
  times(&tv2);
  printf("%d0, (tv2[0] - tv[0])*16);
}
```

Table 4 -- comparisons of memory & Lisp speeds

machine	memory access time		pseudo	benchmark A	benchmark B
	memory	cache			
KA 10	1.9 us		43 ms	0.078 sec	1.10 sec
KI 10	1.0 us		22-29 ms		
11/750	2(?) us	0.32 us	13.5-18 ms	0.103 sec	1.4 sec
11/780	2 us	0.2 us	11-14 ms	0.075 sec	0.920 sec
KL 10	940ns	133ns	13 ms	0.011 sec	0.168 sec
RISC I	0.4 us		10.4 ms		
CDC 7600	0.125		1.8-2 ms	0.014 sec	0.205 sec

The CDC & PDP-10 pseudo tests were done in fortran, and the RISC & VAX in C. Benchmark A was macsyma/vaxima performing a symbolic expansion of (x+y)**12; B was the expansion of (x+y+z)**20

(a) the KL-10 data cache, with an access time near the speed of the CDC 7600 memory cycle,

(b) a vastly superior compiler for the PDP-10, and

(c) a better instruction set.

In [5], a high degree of static locality is shown in lists in PDP-10 Interlisp for five benchmarks. 85-90% of the time, successive cars and cdrs occupy the same page. 79-98% of the time, successive cars and cdrs occupy adjacent locations. Dynamic locality was not measured, but sequential accesses of successive list elements would show locality on a fifo basis. This would make a data cache successful only with fast parallel block fetches, a luxury not available to microprocessors.

The small CDC instruction set bears little resemblance to the RISC I; we must satisfy ourselves that it is not an obstacle to Lisp performance. The CDC architecture distinguishes address/index registers from data registers. In simple tasks such as traversing linked lists, an extra operation must be performed at each fetch to move the fetched pointer into an address register for the next fetch. In the 7600, the register-register move takes 25% of the time required to perform the memory fetch [6]. In the macsyma comparisons, the CDC ran ~25% slower than the KL-10; this may explain a large part of the difference, but the KL-10 case was still able to compensate for time lost in cache misses.

The RISC architecture is not crippled by the address/data register distinction. As a further note, it doesn't seem to suffer from lack of double indirect addressing. This mode was used in the Vax-compiled "pseudo" benchmark, but the VAX still lost to RISC.

Table 5 is from [7], a study of macsyma by John Foderaro and Richard Fateman. It shows the dynamic opcode frequencies of vaxima, running on an 11/780 in Franz Lisp. 22% of all movl's were used in stacking. As will be shown later, the RISC must use registers to be competitive -- in which case parameter stacking is replaced by register-register or memory-register operations, a one-for-one exchange of opcodes. For each of the cases, the opcodes have simple analogs in RISC; the problem is the addressing modes.

The static frequencies for Lisp show that 56% of all instructions are nothing but loads and stores (movl, movab, clrl); again, each of the instructions in the list is simple in nature.

Figure 2 shows the frequency of calls to each procedure in the vaxima system; interestingly, 60% of the time was spent in the (C coded) Lisp system and 40% was spent in (Lisp coded) vaxima. This explains in part why the dynamic opcode frequency leans more toward C than Lisp. Another item of interest is that the notable spikes in the graph show that the most popular procedures did nothing but the simplest operations – creating integers and cells, checking inequalities, garbage collecting, and simple list primitives.

The two major spikes on the chart were coded in VAX assembly language, rather than C; the versions coded for RISC were less than twice the size of the original VAX-code, consistent with several of the C-coded benchmarks.

We see that the RISC has the major feature for good Lisp performance -- memory speed. This puts it in the ball park with VAX and pdp-10, aside from data cache considerations. Current microprocessors have no such edge, so the comparison lies with the instruction sets. A simple benchmark is tested in the next section, where RISC shows encouraging performance.

Figure 2

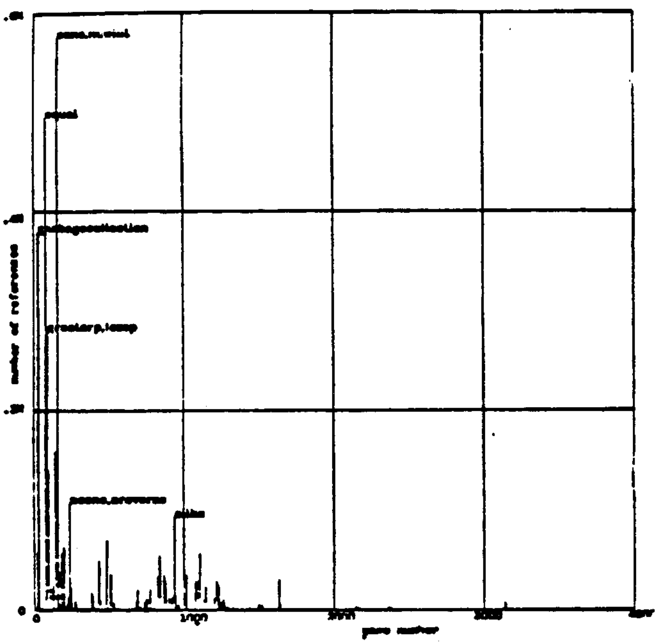


Table 5: Instruction usage

Static				Dynamic	
C coded Lisp functions		Lisp coded Lisp functions		Begin demo	
Instruction	pct	Instruction	pct	Instruction	pct
movl	20	**movl**	43	**movl**	27
pushl	12	**movab**	9	**cmpl**	7
calls	10	**calls**	7	**bnequ**	6
pushal	7	**brb**	4	**beqlu**	5
cmpl	4	**clrl**	4	**ashl**	5
beqlu	4	**jsb**	4	**movab**	4
bnequ	3	**beqlu**	3	**tstl**	4
brb	3	**bnequ**	3	**cvtbl**	3
ret	3	**tstl**	3	**brb**	3
clrl	2	**brw**	2	**calls**	2
other	32	other	18	other	34
100 Unique instr.		32 Unique instr.		109 Unique instr.	

4. aTAKing a current benchmark

As the Franz Lisp system could not be made operational on RISC, hand coding was used to compare performance. A valuable result of this was the realization that Franz Lisp (which makes minimal use of registers) [8] was less suitable for the RISC architecture than the approach used for PSL (Portable Standard Lisp, which uses registers to pass parameters) [9].

Figure 3 shows the TAK benchmark, a heavily recursive function of unquestionable uselessness. It shows the efficiency of procedure call, as well as the difference in speed between fixnum and bignum arithmetic. Fixnum arithmetic refers to integers of bounded length, where operations are tuned to run faster than the unbounded bignum arithmetic on some Lisps. Table 6 shows the execution times for a wide range of machines running a wide range of Lisps. An interesting item to note is the case where 11/750 PSL INUM outruns C; this is probably due to the Lisp compiler removing tail-recursion, while the C compiler is not so sophisticated.

Figure 3 - the TAK benchmark

```
(tak 18 12 6)

(defun tak (x y z)
     (cond ((not (lessp y x)) z)
           (t (tak (tak (sub1 x) y z)
                   (tak (sub1 y) z x)
                   (tak (sub1 z) x y)))))
```

Four entries for RISC are on the list. A C-coded version for RISC performed outrageously well. The Franz benchmark was prepared as follows:

The function was compiled on the Vax using LISZT to produce symbolic assembly. This was converted into RISC code on an instruction-by-instruction basis -- no special models of compilation or RISC-based optimizations were assumed. Two stack pointers, called np and lbot, were passed as parameters. Normally they occupy reserved global registers, but the RISC C compiler does not allow this. The kernel function "lessp" had to be modified to work without the rest of the kernel. This was done in such a way as to force it to use the same set of operations, so we get a valid timing, although an optimizer will affect the final performance. The process is shown step-by-step in appendix I. The kernel funcions and the assembly code were compiled and run on the RISC simulator.

The projection for Franz running on RISC is mediocre compared to C performance. I don't feel safe in "tuning" the code as a real RISC-Lisp compiler might, because the performance may be unrealistically fast. The result is a valid lower bound on performance; it was sufficient to beat the 11/750 in C, and the MC68000 in both Franz Lisp and Pascal. The problem is the excessive amount of memory traffic due to stacking and unstacking Lisp parameters, which are not passed in registers in Franz.

The next stab was to do the same thing in PSL. The PSL kernel is written in a more obscure "SYSLISP", so the two support routines were instead taken from Franz. This benchmark gives a valid lower bound on performance if RISC-Lisp passed parameters in registers, and is like Franz in all other ways. The only difference comes in memory management, where pointers in registers also reference active data.

Table 6 -- executions of the TAK benchmark [12]
Results on tak function, including projection of RISC-compiled Lisp.
Takeuchi function of various types

On 11/750 in Franz ordinary arith	19.9	seconds
On 11/780 in Franz with (nfc)(TAKF)	15.8	seconds
On Dolphin in InterLisp Nov 1981 (tr)	11.195	seconds
On 11/780 in Franz (nfc)	8.4	seconds
On 11/780 in Franz (nfc)	8.35	seconds
On 11/780 in Franz with (ffc)(TAKF)	7.5	seconds
On 11/750 in PSL, generic arith	7.1	seconds
On MC (KL) in MacLisp (TAKF)	5.9	seconds
On Dolphin in InterLisp Jan 1982 (tr)	5.71	seconds
On Dual (MC68000) in Franz(lfc)	5.38	seconds
On Vax 11/780 in InterLisp (load = 0)	4.24	seconds
On Foonly F2 in MacLisp	4.1	seconds
On Apollo (MC68000) Pascal	3.8	seconds (extra waits?)
On 11/750 in Franz, Fixnum arith	3.6	seconds
(Projected) On RISC in Franz (ffc, tr)	3.52	seconds
(Projected) On RISC in Franz (lfc, tr)	3.51	seconds
On MIT CADR in ZetaLisp	3.16	seconds
On MIT CADR in ZetaLisp	3.1	seconds
On MIT CADR in ZetaLisp (TAKF)	3.1	seconds
On Apollo (MC68000) PSL SYSLISP	2.93	seconds
On 11/780 in NIL (TAKF)	2.8	seconds
On 11/780 in NIL	2.7	seconds
On 11/750 in C	2.4	seconds
(Projected) RISC PSL/Franz (lfc, tr)	2.23	seconds
On 11/780 in Franz (ffc)	2.13	seconds
On 11/780 (Diablo) in Franz (ffc)	2.1	seconds
On 11/780 in Franz (ffc)	2.1	seconds
(Projected) RISC PSL/Franz (lfc, tr)	2.04	seconds (NOPs removed)
On 68000 in C	1.9	seconds
On Utah-20 in PSL Generic arith	1.672	seconds
On 11/750 in PSL INUM arith	1.4	seconds
On 11/780 (Diablo) in C	1.35	seconds
On 11/780 in Franz (lfc)	1.13	seconds (open coded?)
On UTAH-20 in Lisp 1.6	1.1	seconds
On UTAH-20 in PSL Inum arith	1.077	seconds
On MC (KL) in MacLisp	.93	seconds
On SAIL (KL) in MacLisp	.83	seconds
On SAIL in bummed MacLisp	.79	seconds
On 68000 in machine language	.7	seconds
On RISC in C	.66	seconds
On Dorado in InterLisp Jan 1982 (tr)	.53	seconds
On UTAH-20 in SYSLISP arith	.526	seconds
On SAIL in machine language	.255	seconds
On SAIL in machine language	.184	seconds
On SCORE (2060) in machine language	.162	seconds
On S-1 Mark I in machine language	.114	seconds

47707 function calls
max recursion depth is 18
average recursion depth is 15.4

notes:
All cases running compiled
(tr) means Tail Recursion Removal
(nfc) means 'normal function call' in Franz
(ffc) means 'fast function call' in Franz
(lfc) means 'local function call' in Franz (function call directly to an entry point using knowledge of the internals of the function by the compiler).
On the 68000 Franz, np & lbot are in registers rather than the standard memory locations.

The PSL compiler generated the VAX assembly code, which was again expanded into the equivalent RISC instructions. Under a naive association of registers, the result was surprisingly good. The current RISC standard (chosen by Jim Miros) is to pass the result of a procedure call back in the same register as the first parameter; the construction of the TAK function saves 3 register moves per call. A second kernel function, "sub1", had to be introduced, since the PSL compiler did not expand it in-line. An in-line expansion would have sped up the benchmark somewhat. This process is shown in appendix II.

The result is the faster PSL/Franz entry for RISC; it outran the 11/750 in C, the 11/780 in NIL (a statically scoped Lisp), and the MC68000 running PSL SYSLISP. The PSL SYSLISP is a Lisp-structured language for systems programming; no other Lisp should run much faster.

The PSL Lisp was still a far cry from the C performance. With INUM arithmetic, it should be much faster. The simplicity of the benchmark leaves little room for optimizations, although the following are possible:

(1) the 4 NOP's can be eliminated; some data flow analysis would be required to keep the operations correct. This appears as the "NOP's removed" entry which outruns the 11/780 Franz.

(2) the branch to the return statement might be replaced by the the return statement itself; this requires simple control flow analysis in the optimizer.

I feel this is an encouraging result. Although not outrageously fast, we can, at worst, expect better performance than the VAX 11/750 or the MC68000. A real Lisp system, with a sophisticated compiler, would gain some edge on the 11/780. With Inum arithmetic, for example, the benchmark should beat the speed of C on the RISC. The Inum arithmetic would use the same hardware arithmetic as C, and removal of tail-recursion would eliminate extra procedure calls.

5. Ups and Downs with the Lisp runtime stack

I was at first worried about the performance of the RISC window file. The window file is an array of eight frames of local registers; procedure calls and returns cause the active frame to shift. In C, most of the stack motion is contained by the window file. Occasionally the file overflows or underflows, and windows have to be moved in or out of memory. The memory traffic due to procedure calls and returns is greatly reduced.

Textbook Lisp programs tend to be highly recursive functions such as factorials or linear list traversals. Such functions would generate long, monotonic rises and falls in the stack height; these would negate the advantages of the window file, as opposed to a standard register saving mechanism. To test the validity of this assumption, a special compiler and interpreter were constructed.

The Lisp compiler, LISZT, was modified to interject a call statement before and after each original call and jsb. These extra calls invoke the tracing procedures "upstack" and "downstack", which put tracing codes into an output file for later analysis.

The Lisp interpreter was rebuilt to trace its own internal calls and jsb's, and the "upstack" and "downstack" procedures are included in the kernel. When Lisp functions were compiled with the tracing compiler, and loaded into the tracing interpreter, all stack movement is monitored, except for system calls and calls to the tracing functions themselves.

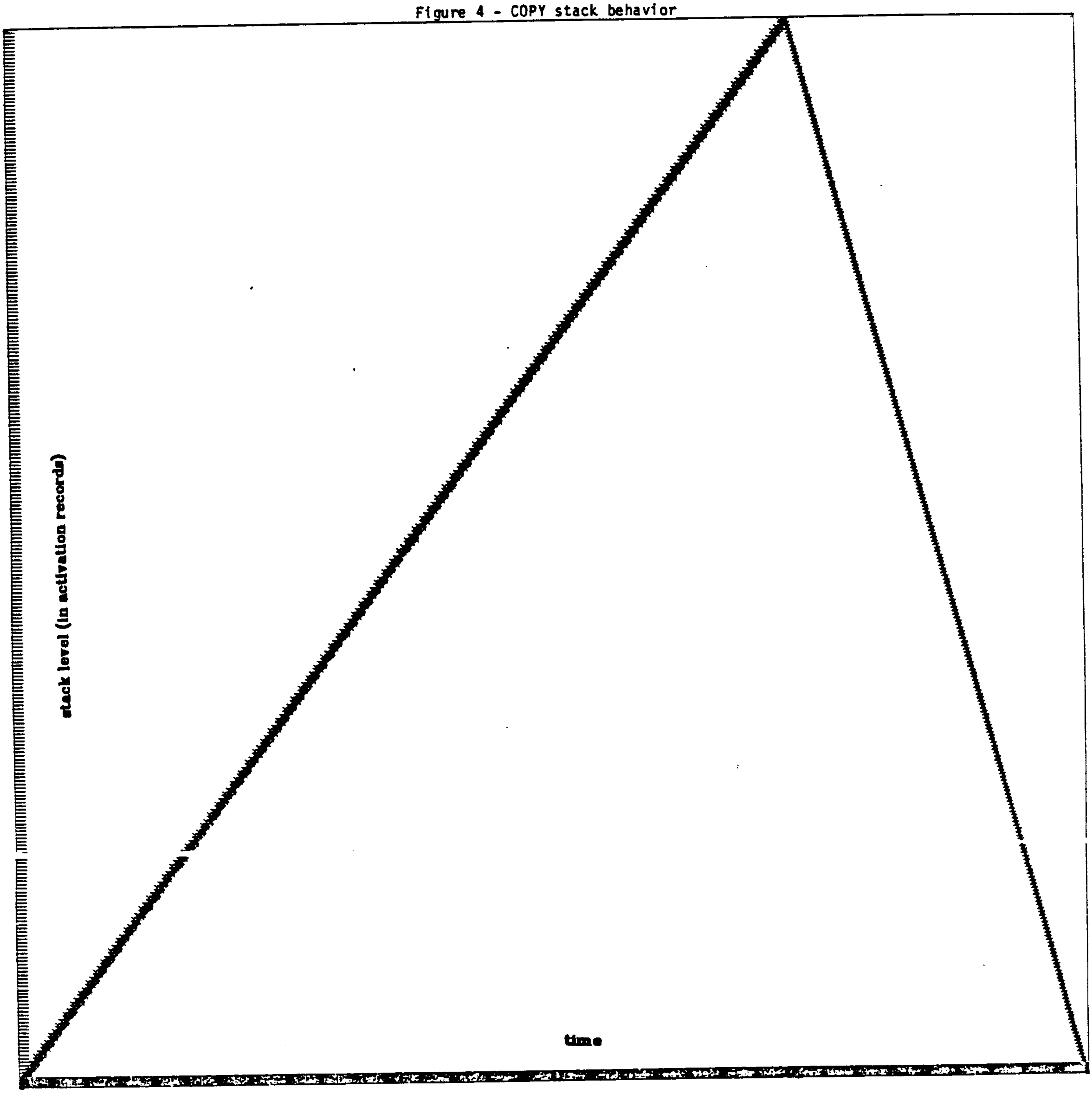
The usage of exotic returns was handled separately. Franz Lisp keeps a linked list threaded through the execution stack, and the links are followed in the event of an abnormal return. Eventually a frame in the execution stack is found, which is capable of catching the exception. I had intended to handle this on RISC by directly writing over the window file, and restoring execution from the correct frame. This requires about the same amount of work as processing a file overflow. To account for the occurrence of such returns in the stack simulation, they were replaced by a sequence of eight stack rises. Rises were used, as opposed to falls, to prevent the stack from falling into negative space; the effect on file performance should be the same. Interestingly, no abnormal returns occurred in any of the test cases.

Six test cases were used, four real and two contrived. The real examples are PHRAN, the PHRasal ANalyser; LISZT, the Franz Lisp compiler; the Lisp-coded portion of LISZT; and GLEAN, a system for performing static analyses of Lisp programs. The two contrived cases are the compiled and interpreted versions of a function which copies a list; it is used to measure the effect of linear stack behavior.

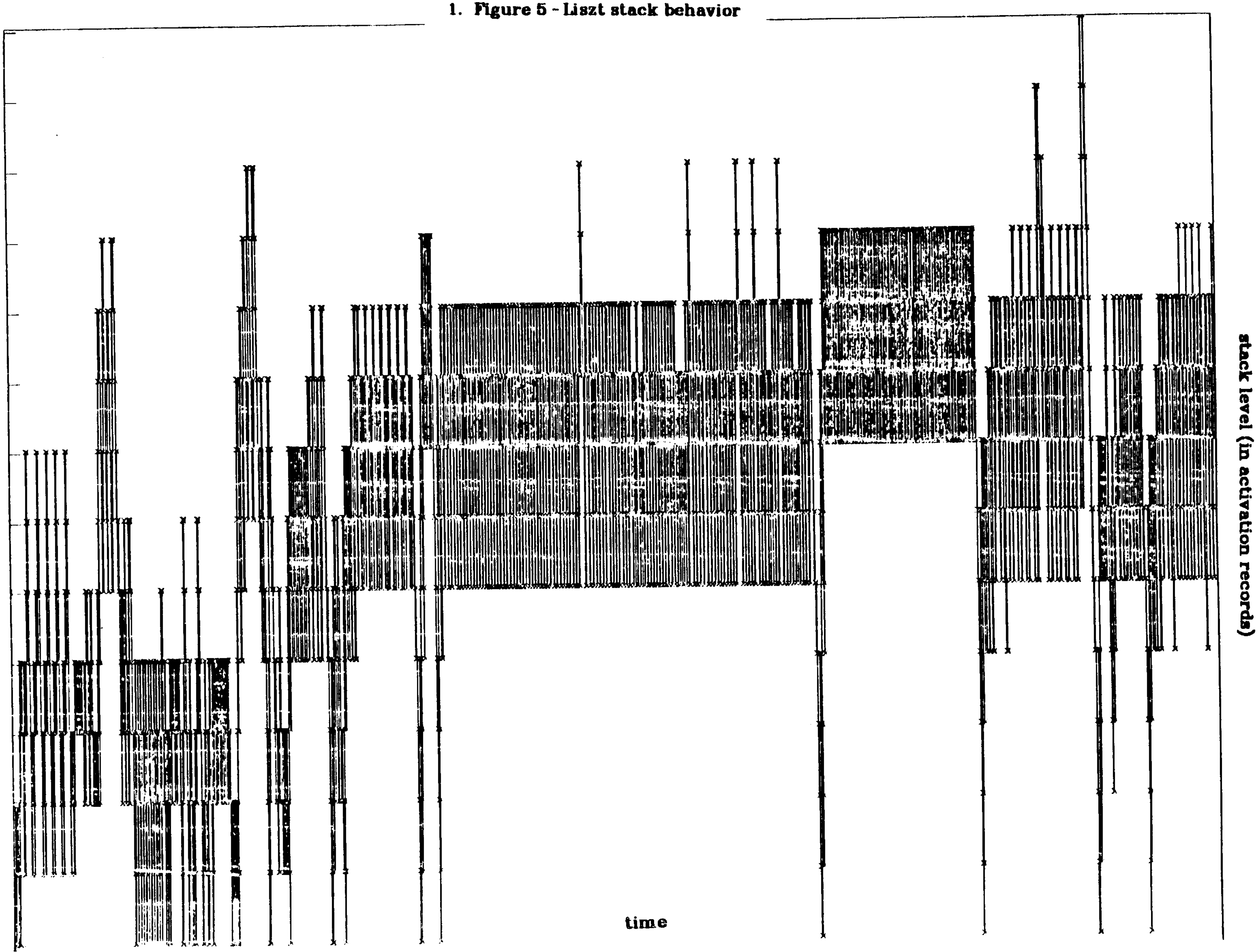
Figure 4 shows the stack behavior of the interpreted copy function copying a list. The intermediate calls in the interpreter obscure the overall rise/fall pattern, so [2,2] replacement is optimal (as in most C programs). Under [2,2] replacement, two windows are copied to memory each time the file overflows, and two are restored from memory each time the file underflows. Figure 5 shows the operation of LISZT as it compiles itself. The monotonic rises and falls tend to be shallow.

Table 7 shows the frequency of the various length rises and falls. Over 80% in each case were of length two or less. Over 60% of all calls & returns were contained in these shallow moves. Table 8 shows the performance of the [2,2] replacement policy. In the case of LISZT, [2,2] replacement came in third place to [2,1] and [1,2] replacement, but was within 3% of the first place policy. In the compiled copy case, [7,6] replacement is in first place. In this contrived example, the stack rises and falls 100 places, with a minor amount of intermediate

Figure 4 - COPY stack behavior



1. Figure 5 - Liszt stack behavior



calls. [7,6] replacement is within 2% of optimal, while [2,2] replacement is consistent with the other measurements. In the other four cases, [2,2] replacement showed the best performance. This is the same best policy as for C.

Table 7 – rise/fall patterns in Lisp

program	% of intervals length=1	% of intervals length=2	% of calls/returns contained by intervals length=1	% of calls/returns contained by intervals length=2	maximum depth
PHRAN	61%	26%	38%	32%	38
LISZT	56%	25%	33%	29%	43
LISZT (Lisp part)	64%	27%	43%	36%	56
GLEAN	61%	22%	37%	27%	26

Table 8 -- behavior of window file, under [2,2] replacement policy

program	calls/returns	% memory traffic over optimal	% file dumps saved
PHRAN	139484	49%	97%
LISZT	392384	63%	98%
LISZT (Lisp part)	141059	46%	98%
GLEAN	1649	40%	98%
COPY (interpreted)	2626	42%	89%
COPY (compiled)	612	51%	68%

Note also from table 8 that the register file was able to contain all but 3% of the procedure calls, saving ~85% of the procedure-call housekeeping. The register file is definitely a success for these cases, something I had not anticipated at the beginning of this project.

This unexpected pattern in stack behavior is probably due to:

(1) a large percentage of the execution occurring in the C-coded kernel, which shows typical C-like behavior,

(2) use of iterative control structures in Lisp, eliminating much need for recursion,

(3) clever compilation frequently removing recursion, and

(4) a large number of intermediate function calls masking out any long, monotonic rises and falls.

The frequency of execution in the Lisp kernel is mentioned in section 2. The shallow maximum depth in table 7 is attributable to the second and third reasons, and the fourth is demonstrated in the graph of figure 4.

6. RISC vs. CISC Lisping

Previously I mentioned execution of data in Lisp; this is commonly done by invoking an interpreter function, even from compiled code. An alternative is to interpret Lisp directly on the hardware or firmware. Some current Lisp machines interpret bytecodes, which are produced by preprocessing Lisp programs; one such is the CADR machine listed in table 6. Its performance is reasonable, although nothing spectacular. An important consideration is whether a machine is single-user or timeshared; the MC68000 shows reasonable performance in comparison with an overloaded pdp-11, so the cost per operation per user may favor the microprocessor.

The SCHEME chip is an attempt to execute Lisp directly in microcode [10]. Scheme [11] is a statically scoped dialect of Lisp. The order of parameter evaluation is unspecified, and all parameters are evaluated before calling. This makes the language well-adapted for compilation.

The projected performance of the scheme chip is good in comparison with interpreted Lisp, but shows poor performance compared with compiled Lisp. A LISP system would have to be largely interpreted to run as slow as the SCHEME chip.

Table 9 shows the comparison of times to compute the 20th Fibonacci number, using Peano arithmetic. The projected performance of SCHEME is twice as fast as the Franz Lisp interpreter, but is twenty times slower than compiled Franz Lisp. This is outrageously bad performance for Lisp applications such as Macsyma.

Figure 6 - The scheme benchmark

```
(defun fib (x)
     (cond ((zerop x) 0)
           ((zerop (sub1 x)) 1)
           (t (plus (fib (sub1 x))
                    (fib (sub1 (sub1 x)))))))

(defun plus (x y)
     (cond ((zerop x) y)
           (t (plus (sub1 x)
                    (add1 y)))))
```

Table 9 - performance of the scheme benchmark	
KA-10 scheme interpreter in MacLisp	3.6 min
VAX 11/780 Franz interpreter	2 min
scheme chip (projected)	1 min
VAX 11/780 Franz, compiled (normal funcall)	8.7 sec
VAX 11/780 Franz, compiled (local funcall)	3 sec

7. Conclusion

We find no reason to anticipate poor performance of Lisp running on RISC architecture. The memory speed is high enough for list processing, the instruction set has the most important features, and the register window file appears surprisingly successful. For good performance, a RISC Lisp system must be modeled after the C system; the language must be compiled and registers must be used as much as possible, resembling more the structure of PSL/SYSLISP.

Other general-purpose and CISC microprocessor systems, such as the MC68000 and scheme chips, are unlikely to deliver superior performance and may indeed perform a great deal worse.

8. References

1. Patterson, D.A. and Sequin, C.H., "A VLSI RISC", Computer, 9/82, p. 8
2. Radin, G., "The 801 Minicomputer", Proc. Symposium on Architectural Support for Programming Languages and Operating Systems, 3/83, p. 39
3. Hennessy, J., Jouppi, N., Baskett, F., Strong, A., Gross, T., Rowen, C., and Gill, J., "The MIPS Machine", Proc. Compcon 2/82
4. Fateman, R.J., "Is a Lisp machine different from a Fortran machine?", ACM SIGSAM bulletin, vol. 12 no 3, August '78 (8-11).
5. Clark, D.W., and Green, C.C., "An empirical study of list structure in Lisp", CACM 20, p. 78, 1977
6. "Control Data 7600 computer system - preliminary reference manual", Control Data corporation, St. Paul, Minnesota, 1968
7. Foderaro, J.K., and Fateman, R.J., "Characterization of VAX Macsyma", Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, SYMSAC 81, p. 14.
8. Foderaro, J.K., and Sklower, K.L., "The Franz Lisp Manual", University of California, Berkeley, 9/81
9. Griss, M.L., and Morrison, B., "The Portable Standard Lisp Users Manual", University of Utah, 6/5/82
10. Sussman, G.J., Holloway, J., Steel, G.L., and Bell, A., "SCHEME-79 -- Lisp on a chip", Computer, 7/81, p. 10.
11. Steele, G.L., and Sussman, G.J., "The revised report on SCHEME – a dialect of Lisp", MIT AI memo #452
12. Gabriel, D., private communication from RPG@SU-AI

Appendix I – coding TAK from Franz to risc

```
(defun tak (x y z)
       (cond ((not (lessp y x)) z)
             (t (tak (tak (sub1 x) y z)
                     (tak (sub1 y) z x)
                     (tak (sub1 z) x y)))))


(defun test ()
    (tak 18 12 6))
```

1.1. TAK in Franz

1.2. Vax output code from Liszt

```
.globl  F00013  #(fcn lambda tak)
F00013:
.word   0x5c0
movab   linker,r8
movl    r7,r10
movab   12(r10),r6
L00014:
movl    4(r10),(r6)+    #(beginning cond)
        #(beginning not)
        #(calling lessp)
        #(from y to stack)
movl    0(r10),(r6)+    #(from x to stack)
movab   -8(r6),r7
calls   $0,*trantb+0
movl    r7,r6
tstl    r0
jneq    L00016
movl    8(r10),r0       #(from z to reg)
jbr     L00015
L00016:
cmpl    0(r10),$1024    #(calling tak)
        #(tail merging)
        #(calling tak)
jleq    L00017
cmpl    0(r10),$9212
jleq    L00018
L00017:
movl    0(r10),r0       #(from x to reg)
jsb     _qoneminus
movl    r0,(r6)+        #(from reg to stack)
jbr     L00019
L00018:
subl3   $4,0(r10),(r6)+
L00019:
movl    4(r10),(r6)+    #(from y to stack)
movl    8(r10),(r6)+    #(from z to stack)
movab   -12(r6),r7
calls   $0,*trantb+8
movl    r7,r6
movl    r0,(r6)+        #(from reg to stack)
cmpl    4(r10),$1024    #(calling tak)
jleq    L00020
cmpl    4(r10),$9212
jleq    L00021
L00020:
movl    4(r10),r0       #(from y to reg)
jsb     _qoneminus
movl    r0,(r6)+        #(from reg to stack)
jbr     L00022
L00021:
subl3   $4,4(r10),(r6)+
```

```
L00022:
movl    8(r10),(r6)+       #(from z to stack)
movl    0(r10),(r6)+       #(from x to stack)
movab   -12(r6),r7
calls   $0,*trantb+8
movl    r7,r6
movl    r0,(r6)+           #(from reg to stack)
cmpl    8(r10),$1024       #(calling tak)
jleq    L00023
cmpl    8(r10),$9212
jleq    L00024
L00023:
movl    8(r10),r0          #(from z to reg)
jsb     _qoneminus
movl    r0,(r6)+           #(from reg to stack)
jbr     L00025
L00024:
subl3   $4,8(r10),(r6)+
L00025:
movl    0(r10),(r6)+       #(from x to stack)
movl    4(r10),(r6)+       #(from y to stack)
movab   -12(r6),r7
calls   $0,*trantb+8
movl    r7,r6
movl    r0,(r6)+           #(from reg to stack)
movl    -12(r6),0(r10)
movl    -8(r6),4(r10)
movl    -4(r6),8(r10)
movab   12(r10),r6
jbr     L00014
L00015:
ret
.globl  F00026  #(fcn lambda test)
F00026:
.word   0x5c0
movab   linker,r8
movl    r7,r10
movab   0(r10),r6
L00027:
movl    $5192,(r6)+        #(calling tak)
        #(from (fixnum 18) to stack)
movl    $5168,(r6)+        #(from (fixnum 12) to stack)
movl    $5144,(r6)+        #(from (fixnum 6) to stack)
movab   -12(r6),r7
calls   $0,*trantb+8
movl    r7,r6
ret
bind_org:
.set linker_size,        0
.set trans_size,         2
```

```
.long   0
.long   0
.long -1
lit_org:
.asciz "lessp"
.asciz "tak"
.asciz "tak"
.asciz "test"
lit_end:
.data  # this is just for documentation
.asciz "@(#)Compiled by Liszt version 8.10 on Tue Sep  7 20:10:14 1982"
.asciz "@(#)decl.l      1.9     3/15/82"
.asciz "@(#)array.l     1.1     9/25/81"
.asciz "@(#)datab.l     1.3     5/27/82"
.asciz "@(#)expr.l      1.3     5/6/82"
.asciz "@(#)io.l        1.1     9/25/81"
.asciz "@(#)funa.l      1.3     2/10/82"
.asciz "@(#)funb.l      1.11    7/21/82"
.asciz "@(#)func.l      1.4     5/7/82"
.asciz "@(#)tlev.l      1.17 3/24/82"
.asciz "@(#)fixnum.l    1.6     10/21/81"
.asciz "@(#)util.l      1.2     10/7/81"
```

1.3. Equivalent code for risc

```
        .globl  F00013              ; (fcn lambda tak)
F00013:
        add     r0, #0, r8                    ; linker stub
        add     r30, #12, r29
L00014:
        ldl     4(r30), r18         ; (beginning cond)
        stl     r18, 0(r29)
        add     r29, #4, r29        ; (beginning not)
                                    ; (calling lessp)
                                    ; (from y to stack)
        ldl     0(r30), r18         ; (from x to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        add     r29, r0, r13        ; pass np & lbot as parameters
        add     r29, #-8, r14
        ldl     trantb+0(r0), r18             ; calling lessp
        call    r15, _lessp(r0)
        add     r0, r1, r16
        add     r29, #-8, r29
        sub     r14, r0, r0, {c}
        jmp     ne, L00016(r0)
        nop

        ldl     8(r30), r30                   ; (returning z)
        jmp     none, L00015(r0)              ; return
        nop

L00016:
        ldl     0(r30), r18                                 ; (calling tak)
        sub     r18,#Fixzero+4596-4096,r0,{c}               ; constant MAY be too big!
                                                            ; (tail merging)
                                                            ; (calling tak)

        jmp     la, L00017(r0)
        nop

        ldl     0(r30), r18
        sub     r18,#Fixzero+4596+4096,r0,{c}    ; constant MAY be too big!
        jmp     le, L00018(r0)
        nop
L00017:
        ldl     0(r30), r14                   ; (from x to reg)
        call    r15, _getout(r0)
        add     r0, r1, r16                   ; exit on overflow

L00018:
        ldl     0(r30), r18
        sub     r18, #4, r18
        stl     r18, 0(r29)
        add     r29, #4, r29
```

```
L00019:
        ldl     4(r30), r18                     ; (from y to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        ldl     8(r30), r18                     ; (from z to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        add     r29, #-12, r14
        ldl     trantb+8(r0), r18
        call    r15, F00013(r0)
        add     r0, r1, r16
        add     r29, #-12, r29
        stl     r14, 0(r29)                     ; (from reg to stack)
        add     r29, #4, r29
        ldl     4(r30), r18
        sub     r18, #Fixzero+4596-4096, r0, {c}        ; (calling tak)
        jmp     le, L00020(r0)
        nop

        ldl     4(r30), r18
        sub     r18, #Fixzero+4596+4096, r0, {c}        ; (calling tak)
        jmp     la, L00021(r0)
        nop

L00020:
        ldl     4(r30), r14                     ; (from y to reg)
        call    r15, _getout(r0)
        add     r0, r1, r16

L00021:
        ldl     4(r30), r18
        sub     r18, #4, r18
        stl     r18, 0(r29)
        add     r29, #4, r29

L00022:
        ldl     8(r30), r18                     ; (from z to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        ldl     0(r30), r18                     ; (from x to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        add     r29, #-12, r14
        ldl     trantb+8(r0), r18
        call    r15, F00013(r0)
        add     r0, r1, r16
        add     r29, #-12, r29
        stl     r14, 0(r29)                     ; (from reg to stack)
        add     r29, #4, r29
        ldl     8(r30), r18                     ; (calling tak)
        sub     r18, #Fixzero+4596-4096, r0, {c}
        jmp     le, L00023(r0)
        nop
```

```
        ldl     8(r30), r18
        sub     r18, #Fixzero+4596+4096, r0, {c}
        jmp     le, L00024(r0)
        nop

L00023:                                         ; (from z to reg)
        ldl     8(r30), r14
        call    r15, _getout(r0)
        add     r0, r1, r16

L00024:
        ldl     8(r30), r18
        sub     r18, #4, r18
        stl     r18, 0(r29)
        add     r29, #4, r29

L00025:                                         ; (from x to stack)
        ldl     0(r30), r18
        stl     r18, 0(r29)
        add     r29, #4, r29                    ; (from y to stack)
        ldl     4(r30), r18
        stl     r18, 0(r29)
        add     r29, #4, r29
        add     r29, #-12, r14
        ldl     trantb+8(r0), r18
        call    r15, F00013(r0)
        add     r0, r1, r16
        add     r29, #-12, r29                  ; (from reg to stack)
        stl     r14, 0(r29)
        add     r29, #4, r29
        ldl     -12(r29), r18
        stl     r18, 0(r30)
        ldl     -8(r29), r18
        stl     r18, 4(r30)
        ldl     -4(r29), r18
        stl     r18, 8(r30)
        add     r30, #12, r29
        jmp     none, L00014(r0)
        nop

L00015:
        ret     r31
        nop
```

```
        .globl  _test                          ; name visible from "C"
        .globl  F00026                         ; (fcn lambda test)
_test:
F00026:
        add     r0, #0, r3                     ; linker stub
        add     r30, r0, r29
L00027:
        add     r0, #Fixzero+4596+72, r18          ; (calling tak)
        stl     r18, 0(r29)                    ; (from (fixnum 18) to stack)
        add     r29, #4, r29
        add     r0, #Fixzero+4596+48, r19          ; (from (fixnum 12) to stack)
        stl     r19, 0(r29)
        add     r29, #4, r29
        add     r0, #Fixzero+4596+24, r18          ; (from (fixnum 6) to stack)
        stl     r18, 0(r29)
        add     r29, #4, r29
        add     r29, #-12, r14
        ldl     trantb+8(r0), r19
        call    r15, F00013(r0)
        add     r0, r1, r16
        add     r29, #-12, r29
        add     r14, r0, r30                       ; pass result upward
        ret     r31
        nop

; linkage stuff
trantb:
        .long   _lessp                         ; location of "lessp"
        .long   F00013                         ;             "tak"
        .long   F00013
        .long   F00026                         ;             "test"
```

Appendix II – coding TAK from PSL to risc

```
(de tak (x y z)
    (cond ((not (lessp y x)) z)
          (t (tak (tak (sub1 x) y z)
                  (tak (sub1 y) z x)
                  (tak (sub1 z) x y)))))


(de test ()
    (tak 18 12 6))
```

2.1. TAK in PSL

```
                (*ENTRY TAK EXPR 3)
                (SUBL2 20 (REG ST))
G0002           (MOVL (REG 1) (DEFERRED (REG ST)))
                (MOVL (REG 2) (DISPLACEMENT (REG ST) 4))
                (MOVL (REG 3) (DISPLACEMENT (REG ST) 8))
                (MOVL (REG 1) (REG 2))
                (MOVL (DISPLACEMENT (REG ST) 4) (REG 1))
                (JSB (ENTRY LESSP))
                (CMPL (REG 1) (REG NIL))
                (JNEQ G0004)
                (MOVL (DISPLACEMENT (REG ST) 8) (REG 1))
                (JBR G0001)
G0004           (MOVL (DEFERRED (REG ST)) (REG 1))
                (JSB (ENTRY SUB1))
                (MOVL (DISPLACEMENT (REG ST) 8) (REG 3))
                (MOVL (DISPLACEMENT (REG ST) 4) (REG 2))
                (BSBW (INTERNALENTRY TAK))
                (MOVL (REG 1) (DISPLACEMENT (REG ST) 12))
                (MOVL (DISPLACEMENT (REG ST) 4) (REG 1))
                (JSB (ENTRY SUB1))
                (MOVL (DEFERRED (REG ST)) (REG 3))
                (MOVL (DISPLACEMENT (REG ST) 8) (REG 2))
                (BSBW (INTERNALENTRY TAK))
                (MOVL (REG 1) (DISPLACEMENT (REG ST) 16))
                (MOVL (DISPLACEMENT (REG ST) 8) (REG 1))
                (JSB (ENTRY SUB1))
                (MOVL (DISPLACEMENT (REG ST) 4) (REG 3))
                (MOVL (DEFERRED (REG ST)) (REG 2))
                (BSBW (INTERNALENTRY TAK))
                (MOVL (REG 1) (REG 3))
                (MOVL (DISPLACEMENT (REG ST) 16) (REG 2))
                (MOVL (DISPLACEMENT (REG ST) 12) (REG 1))
                (JBR G0002)
G0001           (ADDL2 20 (REG ST))
                (RSB)
*** (TAK): base 547566, length 123 bytes
TAK
                (*ENTRY TEST EXPR 0)
                (MOVL 6 (REG 3))
                (MOVL 12 (REG 2))
                (MOVL 18 (REG 1))
                (JMP (ENTRY TAK))
*** (TEST): base 547776, length 15 bytes
```

2.2. Intermediate code from compiler

```
58:     subl2   $14,sp
5b:     movl    r1,(sp)
5e:     movl    r2,4(sp)
62:     movl    r3,8(sp)
65:     movl    r1,r2
69:     movl    4(sp),r1
6d:     jsb     *$c0000801
73:     cmpl    r1,r11
76:     bnequ   7e
79:     movl    8(sp),r1
7c:     brb     cf
7e:     movl    (sp),r1
81:     jsb     *$c0000802
87:     movl    8(sp),r3
8b:     movl    4(sp),r2
8f:     bsbw    58
92:     movl    r1,c(sp)
96:     movl    4(sp),r1
9a:     jsb     *$c0000802
a0:     movl    (sp),r3
a3:     movl    8(sp),r2
a7:     bsbw    58
aa:     movl    r1,10(sp)
ae:     movl    8(sp),r1
b2:     jsb     *$c0000802
b6:     movl    4(sp),r3
bc:     movl    (sp),r2
bf:     bsbw    58
c2:     movl    r1,r3
c5:     movl    10(sp),r2
c9:     movl    c(sp),r1
cd:     brb     5b
cf:     addl2   $14,sp
d2:     rsb
d3:     movl    $6,r3
d6:     movl    $c,r2
d9:     movl    $12,r1
dc:     jmp     *$c0000800
e2:     clrl    r3
e4:     movl    $f2000804,r2
eb:     movl    $f2000800,r1
f2:     jsb     *$c0000805
```

2.3. **VAX output code from compiler**

```
        .globl  tak                     ; (fcn lambda tak)
tak:
        add     r30, r0, r13
        add     r29, r0, r14
        call    r15, _lesso(r0)
        add     r0, r1, r16
        add     r14, r0, r0, {c}
        jmp     ne, stay(r0)
        nop
        add     r28, r0, r30
        jmp     none, out(r0)
        nop
stay:
        add     r30, r0, r14
        call    r15, _sub1(r0)
        add     r0, r1, r16
        add     r28, r0, r12
        add     r29, r0, r13
        call    r15, tak(r0)
        add     r0, r1, r16
        add     r14, r0, r17
        add     r29, r0, r14
        call    r15, _sub1(r0)
        add     r0, r1, r16
        add     r30, r0, r12
        add     r28, r0, r13
        call    r15, tak(r0)
        add     r0, r1, r16
        add     r14, r0, r18
        add     r28, r0, r14
        call    r15, _sub1(r0)
        add     r0, r1, r16
        add     r29, r0, r12
        add     r30, r0, r13
        call    r15, tak(r0)
        add     r0, r1, r16
        add     r14, r0, r28
        add     r18, r0, r29
        add     r17, r0, r30
        jmp     none, tak(r0)
        nop
out:
        ret     r31
        nop

.globl _test
_test:
        add     r0, #Fixzero+24, r28    ; fixnum 6
        add     r0, #Fixzero+48, r29    ;        12
        add     r0, #Fixzero+72, r30    ;        18
        jmp     none, tak(r0)
        nop
```

2.4. **Equivalent code for risc**