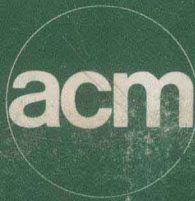


**PROCEEDINGS
of the
Fifth Symposium
on
SMALL SYSTEMS**



**Special Interest Group on Small Computers
Association for Computing Machinery**

**2, 3, 4 August 1982
Colorado Springs, Colorado**

**Editor: G. W. Gorsline
Virginia Polytechnic
Institute and State
University**

ACM ORDER NO. 609820

PROCEEDINGS

Fifth SIGSMALL Symposium

2,3,4 August, 1982

Colorado Springs, Colorado

Sponsored by the
Association for Computing Machinery
Special Interest Group on Small Computers

Edited by:
G. W. Gorsline

Virginia Polytechnic Institute and State University

The Association for Computing Machinery, Inc.
11 West 42nd Street
New York, New York 10036

Copyright © 1982 by the Association for Computing Machinery, Inc.. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage and credit to the source is given. Abstracting with credit is permitted. For other copying of articles that carry a code at the bottom of the first page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, P.O. Box 765, Schenectady, NY 12301. For permission to republish write to: Director of Publications, Association for Computing Machinery. To copy otherwise, or republish, requires a fee and/or specific permission.

ISBN 0-89791-083-4

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

Price:
Members: \$ 8.00
All Others \$12.00

ACM Order Number: 609820

Fifth SIGSMALL Symposium

General Chairman:	Dan Lewis
Program Co-Chairmen:	
Panels:	Larry Walker
Papers:	G. W. Gorsline
Distinguished Speakers:	
Arrangements:	Fred Maryanski
Local Arrangements:	S. Ron Oliver
Treasurer:	H. K. Hodge
Publicity:	Annie Claybrook
SIGSMALL Chairperson:	Fred Maryanski

1982 ACM SICSMALL Symposium
on
Small Systems

Colorado College
Colorado Springs, Colorado

Sunday 1 August 1982

1900 - 2100 REGISTRATION

Monday 2 August 1982

0800 - 0900 REGISTRATION

0900 - 1030 OPENING SESSION

Welcome by SIGSMALL Chairperson, Dr. Fred Maryanski, Digital Equipment Corporation, Hudson, Massachusetts.

Conference Opening by Symposium Chairperson, Dr. Daniel W. Lewis, University of Santa Clara.

Keynote Address: Dr. George Champine, "Workstation Technology and Applications", Digital Equipment Corporation, Marlboro, Massachusetts

1100 - 1230 SOFTWARE TOOLS - Current Work

"An Optimizing Compiler for LISP for the Z80",
Jed Marti, University of Oregon. p.3

"SAUSTALL - A New Software Environment",
Robert K. Bell, Uttenreuth, West Germany. p.9

"A Practical Application of PARNAS Modular Approach",
Joyce R. Calabro and Maria M. Puzzo, the MITRE Corporation,
Bedford, Massachusetts. p.14

1345 - 1515 ADA Systems - A Panel Discussion

Moderator: Dr. Billy Claybrook, the MITRE Corporation,
Bedford, Massachusetts.

1545 - 1745 APPLICATIONS - Reports

"Multimicrocomputer System for Building with Full Service Facility Automation", J.F. Tirado, J.J. Ruz, and M. Mellado, Universidad Complutense, Madrid, Spain. p.21

"Proposing Resident User-Cycles in Personal Computers",
Peter Schulthess and Jiri Hoppe, Swiss Federal Institute of Technology, Zurich, Switzerland. p.26

"CMOS New Development for Health Care Industry: Procedures and Pitfalls", Hal Berghel, University of Nebraska-Lincoln. p.33

Tuesday 3 August 1982

0800 - 0900

REGISTRATION

0900 - 1030

DISTRIBUTED SYSTEMS - Current Work

"A Data Highway for Realtime Distributed Systems: Architecture and Formalization", A. Faro and O. Mirabella, University di Catania, Italy. p.38

"Semantics of a Communication Mechanism in a Distributed System", Robert Zicari, Politecnico di Milano, Italy. p.43

"Distributed MIMD Multiprocessor System with MICROADA/SUPERMICROTM for Asynchronous Concurrent Newton's Algorithms", Shohel Fujita, Tokyo Institute of Technology, Tokyo, Japan. p.49

1100 - 1200

32-bit Microprocessors - A Panel Discussion

Moderator: Dr. G. W. Gorsline, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

1315 - 1500

DATA MANAGEMENT - Current Reports

"Data Base Systems for Large and Small Systems", Ivan Flores, Baruch College, City University of New York. p.60

"PANDA: A Pascal Network Data Base Management System", Andre Frank, Swiss Federal Institute of Technology, Zurich, Switzerland. p.73

"A Data Management System for a Small Computer", Bill Hays, Gordon Stokes, and Jim Johnson, Brigham Young University, Provo, Utah. p.76

"IRAIN - An Instructional Relational Database System Designed for Small Computer System", A.C. Liu and S.K. Chang, North Dakota State University and University of Illinois at Chicago Circle. p.82

1500 -

TOUR - Semiconductor Production Facility

Wednesday 4 August 1982

0800 - 0900

REGISTRATION

0900 - 1230

TUTORIAL - Local Area Networks

Tutor: Professor Hasan Al Khatib, Department of Electrical Engineering and Computer Science, University of Santa Clara, Santa Clara, California 95053.

MAJOR PANEL

ADA Systems

Moderator: Billy Claybrook (MITRE)

MAJOR PANEL

32-bit
Microprocessors

Moderator: G. W. Gorsline (Virginia Tech)

AN OPTIMIZING COMPILER FOR LISP FOR THE Z80

Jed Marti

University of Oregon

ABSTRACT This paper describes an optimizing compiler for the Z80. Described are the compilation mechanisms, optimization techniques, and performance statistics.

Introduction

UOLISP [1] is a subset of Standard LISP [2] implemented for the Z80 microprocessor. It runs in a minimum of sixteen thousand bytes of storage and most effectively in thirty-two thousand or more. The system is more than just a basic LISP interpreter. The entire facility consists of:

- a program to load precompiled "fast load" files.
- a parser for a subset of RLISP [3].
- a function trace and break facility.
- a LISP structure editor.
- an online help facility and text formatter.
- a pretty print facility.
- the Little MUTA translator writing system [4].
- a compiler and optimizer.
- arbitrary precision integer package.

This paper addresses the mechanisms of the compiler and its optimizer.

The compilation process is divided into three passes: the first translates LISP into pseudo-assembly code called LAP (for Lisp Assembly Program), the second pass performs a peephole optimization on the LAP assembly code, the third pass translates this LAP into absolute machine code and places this in storage for execution or dumps it to a file for later restoration.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Overview

The LISP interpreter contains code for reading functions into the LISP system and executing them interpretively much like other microprocessor based systems. Unfortunately interpreted functions require large amounts of storage and execute very slowly.

A more efficient scheme reads functions in the interpretive form, and then compiles them to machine code to be executed directly by the microprocessor. The interpreted version of the function disappears, its storage becomes available for use at a later time.

For example, the function FACT, which computes the factorial of a number recursively, is defined in UOLISP as follows:

```
(DE FACT (N)
  (COND ((LESSP N 2) 1)
        (T (TIMES? (FACT (SUB1 N)) N))))
```

In UOLISP, dotted-pairs, of which this function is composed, take 4 bytes each. 22 dotted-pairs are used to define FACT for a total of 88 bytes. UOLISP's compiler generates the following code for FACT:

Loc.	Code	LAP	INTEL
0000:		ENTRY FACT,EXPR ;FACT:	
0000:	CD96B3	CALL ALLOC	; CALL ALLOC
0003:	02	DEFB 2	; .BYTE 2
0004:	D7FE	STOX HL,-1	; *STOX HL,-1
0006:	110240	LDI DE,2	; LXI DE,2
0009:	E7	RST LINK	; RST LINK
000A:	1720	DEFW LESSP	; .ADDR LESSP
000C:	2805	JREQ \$1	; JRZ \$1
000E:	210140	LDI HL,1	; LXI HL,1
0011:	180D	JR \$0	; JMPR \$0
0013:		\$1:	; \$1:
0013:	CFBF	LDX HL,-1	; *LDX HL,-1
0015:	E7	RST LINK	; RST LINK
0016:	8320	DEFW SUB1	; .ADDR SUB1
0018:	E7	RST LINK	; RST LINK
0019:	A820	DEFW FACT	; .ADDR FACT
001B:	CF7F	LDX DE,-1	; *LDX DE,-1
001D:	E7	RST LINK	; RST LINK
001E:	1D21	DEFW TIMES2	; .ADDR TIMES2
0020:		\$0:	; \$0:
0020:	CD08B4	CALL RDLLOC	; CALL RDLLOC
0023:	FE	DEFB -2	; .BYTE -2

* means macro form.

A total of 36 bytes are used, less than half the size of the interpreted version. The compiled version runs over 40 times as fast.

Compilation Mechanisms

Compiled programs move information between registers and call subroutines to perform most operations. In this section we describe how important LISP constructs are implemented in LAP and enumerate the various support functions required.

Parameter Passing

Zero to three parameters may be passed to a function. The first argument of a function (if it has any) will always be in the HL register pair, the second in DE, and the third in BC. Functions with more than three arguments cannot be compiled. This particular mode of execution is called the register model as opposed to the more common stack model. We believe that the register model is inherently more efficient than the stack model though perhaps more difficult to compile for.

Stacks

Function parameters and PROG type variables are kept in a stack frame, sometimes called an activation record, a contiguous block of locations pointed to by the IX index register. When a function is invoked it creates a new frame on the top of the stack by calling the ALLOC support routine. When a function terminates it calls the DALLOC routine which subtracts the number of locations used from IX, freeing the space for use by the next function.

Storing and retrieving values from the stack frame is accomplished by the two support routines LDX and STOX. Since these operations occur frequently in compiled code it is necessary that they use as little storage as possible. Therefore the LDX and STOX routines are called using the 280 RST instruction with the following byte containing which register pair is to be stored (or loaded), and the displacement from the top of the stack frame. The LAP instructions generated by the compiler are also called LDX and STOX and contain the register pair name and what displacement is to be used.

Both LAMBDA expressions and PROG forms generate the ALLOC and DALLOC calls to handle stack frames. One of the optimizations performed is to substitute the appropriate number of increment or decrement IX instructions, or for larger frames, a sequence to add to IX. This has the disadvantage of not checking for stack overflow.

The 280 internal stack is used for saving return addresses and intermediate values during function evaluation. A call to a function FUN3 with three arguments stores the results of evaluation of the first two arguments on the 280 stack while the third is being computed. The values are popped into the appropriate registers just before the function is invoked.

(FUN3 (FUNA ...) (FUNB ...) (FUNC ...))

would generate the following code sequence:

```
... evaluate FUNA ...
  PUSH HL          ;Save result of FUNA.
... evaluate FUNB ...
  PUSH HL          ;Save result of FUNB.
... evaluate FUNC ...
  LDHL BC          ;Move HL to BC.
  POP DE           ;FUNB is second argument.
  POP HL           ;FUNA is first argument.
  RST LINK         ;Call FUN3.
  DEFW FUN3
```

Function Invocation

The compiler will not always know the address of a function being called because it might not be defined yet. Even if the function is defined the compiler does not know whether it will be compiled or interpreted at run time. A special internal subroutine called LINK is used to transfer control at run time. Since both compiled and interpreted functions can exist at the same time, LINK will perform either of two functions. If an interpreted function is being called from compiled code the LISP interpreter will be invoked for that function. If the function being called is compiled or is a system function the call to LINK will be replaced by a direct call to that function. The call to the LINK function must be an RST type link so that the three byte 280 CALL instruction will exactly replace the compiled call. If the system global variable !*FLINK is NIL the substitution will not take place and the slow link form will remain. This is a useful debugging tool as it allows you to compile functions and change their definitions (for tracing) without reloading the system.

<u>Compiled as:</u>	<u>Changed by LINK to:</u>
RST LINK	CALL function-address
DEFW function-name	

The two byte DEFW attached to the LINK contains the symbol table pointer of the function being called. At execution time the LINK routine looks for either a compiled or interpreted function attached to the name and either invokes EVAL, generates the CALL, or if the !*FLINK flag is on, just transfers to the function. If no such function is defined, an error will occur and the name of the function will be displayed.

The LIST Function

The LIST function is compiled in a special way to take advantage of the 280 internal stack. The arguments of the LIST function are compiled and the results of each are pushed onto the stack. When all have been computed the support function CLIST is called.

(LIST (F1 ...) ... (Fn ...))

compiles to:

```

... evaluate F1 ...
  PUSH HL      ;Save result of F1 for CLIST.
  .
  .            ;Evaluate other arguments.
  .
... evaluate Fn ...
  PUSH HL      ;Save result of Fn for CLIST.
  MVI  A,n     ;Number of values on stack.
  CALL CLIST   ;Call to CLIST routine.

```

COND Compilation

The LISP COND function is compiled into a series of tests and conditional jumps. The CMPNIL support routine compares the result of a predicate to NIL and sets the Z80 NZ and Z flag bit which controls the conditional branch instructions generated. If the last predicate of the COND is T, the predicate and jump will not be compiled (the usual case).

(COND (a₀ c₀) ... (a_n c_n))

generates the following code:

```

... evaluate a0 ...
  RST  CMPNIL  ;Is a0 NIL?
  JPEQ G00001  ;Yes, jump to next antecedent
... evaluate c0 ...
  JP   G00002  ;First consequent done, quit.
G00001:          ;Come here if a0 not T.
  .
  .            ;Evaluate other a - c pairs.
  .
G0000x:          ;Try last predicate.
*... evaluate an ...
*  RST  CMPNIL  ;Is last one NIL?
*  JPEQ G00002  ;Go return NIL if yes.
... evaluate cn ...
G00002:          ;Always come here when done.

```

Lines preceeded by an asterisk are not generated if the last predicate is T.

PROG, GO, and RETURN

The PROG function and the control constructs GO and RETURN are compiled by plugging labels and values into a template.

```

(PROG (X))
  .
  LBL ...
  .   ... (RETURN value)
  .
  .
  (GO LBL)
  .
  ... )

```

compiles to:

```

  CALL ALLOC    ;Space to save X allocated.
  DEFB  2
  LDI  HL,NIL   ;PROG variable set to NIL.
  STOX HL,-1
  .
  .
  LBL:          ;A PROG label generated.
  .
  .

```

```

... evaluate value ...
  JP   G00001   ;Jump to the end of PROG.
  .
  .
  .
  JP   LBL      ;(GO LBL) generates JP.
  .
  .
  .
G00001:          ;RETURN's come here.
  CALL DALLOC   ;Deallocate stack frame for
  DEFB  -2      ; storage of X.

```

AND and OR Compiled.

AND and OR are compiled identically except that the evaluation of the arguments of AND terminates if one is NIL, and the evaluation of OR terminates if one is non-NIL. The compilation of AND generates JPEQ instructions after a comparison to NIL, and the compilation of OR generates JPNEQ instructions.

(AND a₀ ... a_n)

compiles to:

```

... evaluate a0 ...
  RST  CMPNIL  ;Is result of a0 NIL?
  JPEQ G00001  ;Stop evaluation if yes.
  .
  .            ;Evaluate other arguments.
  .
... evaluate an ...
G00001:          ;Always end up here.

```

Constants, Variables, and Quoted Values

These items are loaded directly into the correct register for the function to which they are to be passed. Local and global variables may have values assigned to them with the appropriate store instruction.

Quoted items are saved on a list of compiled quoted values so that the garbage collector will not remove them. The value representing the quoted item is loaded into the appropriate register.

Compiling FEXPR Calls

When compiling calls to user or system defined FEXPR's the argument list is passed as a list to the function for evaluation. This interpreted form interacts poorly with compiled code for the following reason. All local variable names declared in a function are replaced with their stack frame locations by the compiler. When the FEXPR tries to evaluate its argument in the environment of the calling routine, the variable name in the S-expression cannot be found. The solution is to declare any variables to be passed to an FEXPR for evaluation as GLOBAL. This need not be done for COND, PROG, OR, and AND because these forms are compiled into object code rather than as calls to functions.

The Optimizer

The optimizing phase is divided into two passes and features two levels of optimization and a speed or space choice. The first phase is an extended peephole optimization, the second removes function prologs and epilogs from routines which do not need stack frames. The three levels of optimization include a "safe set", a set of speed optimizations which increase code size, and a "dangerous set" which removes some error checking.

The Closing Window

There has been considerable research on peephole optimization for retargetable compilers [5-7]. The version used in the UOLISP optimizer might be more aptly called a "closing window" optimizer. The hole examined by the optimizer initially includes the entire program. Each instruction is removed from the window in turn. The advantage of this mechanism is that the entire program may be scanned for each instruction examined. Most of the optimizations do not scan very far ahead.

Redundant Instruction Removal

This optimization removes several forms of instructions which replicate data already in registers. For example:

```
STOX HL,-1      becomes  STOX HL,-1
LDX  HL,-1
```

The closing window method permits any number of instructions between the STOX and LDX which do not modify the contents of HL (or whatever register is used).

A second optimization removes store instructions whose location is never referenced. This optimization is very important in small sub routines. If all store instructions are removed, the stack frame allocation prolog and epilog may also be removed. Many very small routines can be reduced in size by as much as 85%. Since a great deal of time is spent in small routines, this optimization can be very important.

Jump Instructions

Several optimizations of this type are performed. The simplest removes unreachable code.

```
JP    label_a
.
.
.
label_b: ...
```

All instructions between the JP instruction and the first label (label_b) following it are removed since they cannot be reached from anywhere. The same optimization is performed when a subroutine is called from which no return can be expected. Functions which always generate an error or use the THROW function have this feature.

Another jump optimization removes worthless forward jumps. Thus:

```
JP    label_a
label_a:
```

results in the jump instruction being removed completely.

Conditional expressions are examined for multiple inversions. Thus:

```
CALL NOT      becomes  RST  CMPNIL
RST  CMPNIL    JP-not-cond. label
JPcond. label
```

The final jump optimization garners the most savings of all optimizations. It determines the distance jump instructions must travel and if it is less than 127 bytes in either direction the instruction is converted to its short form. Since most LISP functions are very short, most jumps end up in their short forms saving 1 byte. Unfortunately short jumps are usually 20% slower.

Stack Frame Optimizations

Many times the end of a PROG form is also the end of its corresponding LAMBDA expression and two DALLOC calls will occur in a row. In this case the optimizer combines the two calls into one by adding their sizes together. A further optimization occurs if the last CALL DALLOC is immediately followed by a RET instruction. The call to DALLOC is replaced by a call to the special routine RDLLOC which automatically does the extra return. The use of this routine saves 1 byte and about 5 microseconds (for the 4 mhz. Z80A) on each function exit.

Reduction in Strength

This class of optimizations replaces several long form instructions (or sets of instructions) with a simpler Z80 instruction. Thus moving HL to DE has an XCHG instruction substituted, saving a single byte. A 3 byte call to any of the CAAR, CADR, CDAR, and CDDR is replaced with two single byte calls on CAR and CDR saving a single byte. This optimization is disabled on machines which do not have the 1 byte calls on CAR and CDR. Finally, the 4 byte version of the LHLD instruction is replaced with its shorter and faster 3 byte version.

Fast Optimizations

The LDX and STOX stack frame referencing functions take two bytes for each use. The functions themselves take approximately 50 microseconds to execute. Approximately 50% of the execution of compiled code is spent in these two routines. By open coding them as indexed MOV instructions, the time is reduced to less than 10 microseconds at the expense of 4 additional bytes. This particular optimization can be turned on and off by the user so that very important functions are optimized and less important ones, slower but much smaller. In the factorial example, use of this optimization results in a 24% speed improvement at a cost of a 38% increase in size.

Dangerous Optimizations

This set of optimizations removes a number of error checks to increase execution efficiency. With selective use they cause no problems. One such optimization replaces the stack frame allocation routine calls by a string of increment or decrement register IX instructions:

```
CALL  ALLOC      becomes  INX  X
DEFB  4                               INX  X
                                      INX  X
                                      INX  X
```

Larger stack frames use a DADX instruction rather than the increments.

```
CALL  ALLOC      becomes  EXX
DEFB  16                               LXI  HL,16
                                      DADX HL
                                      EXX
```

The corresponding decrement forms are used for the stack frame deallocation calls. The deallocation is done as part of the fast optimization because it is never dangerous.

The second optimization is open coding of the ADDI and SUBI functions. These are replaced by INX HL, and DCX HL instructions. They are not dangerous as long as the sign of the number does not change. A sign change causes overflow into the tag field of a number changing it into a bad identifier or string pointer.

Second Optimization Pass

The second optimization pass removes the function prolog and epilog if no stack frame is used. Thus the function:

```
(DE CAAAAR (X) (CAAR (CAAR X)))
```

is compiled without optimization into:

```
ENTRY CAAAAR,EXPR
CALL  ALLOC
DEFB  2
STOX  HL,-1
LDX   HL,-1
CALL  CAAR
CALL  CAAR
CALL  DALLOC
DEFB  -2
RET
```

This version uses 19 bytes. After the first optimization pass the following code is produced:

```
ENTRY CAAAAR,EXPR
CALL  ALLOC
DEFB  2
RST   CAR
RST   CAR
RST   CAR
CALL  RDLLOC
DEFB  -2
```

This version takes 12 bytes. The second pass notices that the stack frame is never used (there

are no STOX or LDX instructions). The final pass produces:

```
ENTRY CAAAAR,EXPR
RST   CAR
RST   CAR
RST   CAR
RST   CAR
RET
```

The final version takes only 5 bytes, a savings of about 75%.

Execution Statistics

We now examine the effect of the optimizer on code size and execution speed. A rough approximation of two different types of programs and their size and execution statistics are given. The first program is the factorial example. 6! was computed 10,000 times on a 4 megahertz, 64k CP/M system. The second test does a complete reversal to all levels of a binary tree. It is also executed 10,000 times and experiences 6 garbage collections.

```
(DE SUPER!-REVERSE (A)
(COND ((ATOM A) A)
(T (CONS (SUPER!-REVERSE (CDR A))
(SUPER!-REVERSE (CAR A)) ))))
```

The tree ((A . B) . (C . D)) was reversed to ((D . C) . (B . A)).

	<u>Size</u>	<u>A/B</u>	<u>Time</u>	<u>A/B</u>
	Bytes		Seconds	
No optimization	42	/ 44	48	/ 43
Safe optimization	37	/ 38	45	/ 37
Safe and fast	51	/ 56	34	/ 26
Fast and dangerous	49	/ 56	27	/ 23

At best the optimizer provides a 47% speed up at the expense of a 20% space increase.

To get a view of the effectiveness of each of the individual optimizations over a class of programs, 8 different programs were compiled and the number of bytes saved by each of the reduction in size optimizations were tallied.

Opt. No.	Program								Total
	A	B	C	D	E	F	G	H	
1	44	16	16	12	0	24	12	12	136
2	52	43	20	5	13	16	7	42	198
3	34	34	38	2	26	28	8	98	268
4	18	28	8	0	2	2	2	36	96
5	56	118	52	10	3	4	0	0	233
6	0	6	0	0	6	0	0	15	27
7	12	54	0	3	0	6	6	33	114
8	47	8	26	4	10	21	18	77	211
9	16	64	16	0	0	7	7	84	194
10	22	0	36	2	8	8	20	2	98
11	66	27	27	9	6	18	24	30	207
12	129	170	75	20	33	55	60	135	677
13	12	1	31	0	4	2	5	8	63
14	33	21	0	9	4	12	0	0	79
%	12	12	13	10	14	12	14	14	12.5

The most important space optimization by far is the short jump conversion, the second, the removal of redundant load register instructions, the third the conversion of 4 byte LHLD instructions to 3 bytes, the fourth, the conversion of 16 bit move HL to DE instruction (actually two instructions) to an XCHG instruction, the fifth, the inversion of conditional jumps, and the sixth the use of the RDLLOC stack deallocation routine. The least important is the removal of dead code after functions which do not return.

The average reduction in size achieved by the optimizer is a little over 12.5%. This compares very favorably with other peephole optimizers which gather about 15 % (one of these has over two hundred separate optimizations).

A final test compares UOLISP generated compiled code with that produced by various compilers for mainframes.

Test	A	B	C	UOLISP
1	132	391	51	1450
2	135	3502	1037	4000
3	117	748	1173	15500
4	562	4692	2312	18500
5	2072	8313	2023	37000
6	1098	9231	12806	109000
7	1062	1972	1360	13700
8	1019	18326	6800	49000

The 8 different programs tested were designed to exercise various features of compiled LISP code. The tests for the first three LISP compilers were taken from [8] and have been subsequently improved. Machine A is a large DEC 2060 running LISP 1.6 with the Portable LISP Compiler [9], machine B is a VAX 11/750 running Franz LISP, machine C is a VAX 11/750 running Portable Standard LISP Version 2 [10], and UOLISP runs on a 64k Z80A system with CP/M 2.2. A few of the time tests reflect the relatively small amount of space available and a large number of garbage collections. The statistics show that compiled UOLISP code is on the average one fiftieth the speed of a DEC 2060 running LISP 1.6, one seventh the speed of Franz LISP, and one tenth the speed of Portable Standard LISP on the VAX 11/750.

Conclusions

The UOLISP compiler runs on almost any Z80 based machine with a minimum storage configuration of 32k bytes and a disk drive. The compiler and optimizer have been tested under both CP/M and the TRS-80 Model I and III with success. Turning on all of the optimizations slows down compilation by approximately 40 percent. The UOLISP compiler occupies 3750 bytes of storage and the optimizer with statistics collection another 3000 bytes. Standard Use has debugging done without the presence of the optimizer and the final run with the optimizer enabled.

List of References

1. J. Marti, 'UOLISP Users Manual', University of Oregon Department of Computer and Information Science Technical Report CIS-TR-80-18.
2. J. Marti, A. C. Hearn, M. L. Griss, C. Griss, 'The Standard LISP Report', SIGPLAN Notices, Vol. 14, No. 10, (October 1979), pp. 48-68.
3. A. C. Hearn, 'REDUCE 2 Users Manual', Utah Symbolic Computation Group UCP-19, University of Utah, 1973.
4. J. Marti, 'A Session with the Little Meta Translator Writing System', University of Oregon Technical Report CIS-TR-82-01.
5. J. W. Davidson, C. W. Fraser, 'The Design and Application of a Retargetable Peephole Optimizer' ACM TOPLAS, Vol. 2, No. 2, (April 1980), pp. 191-202.
6. A. S. Tannenbaum, H. van Staveren, J. W. Stevenson, 'Using Peephole Optimization on Intermediate Code', ACM TOPLAS, Vol. 4, No. 1, (January 1982), pp. 21-36.
7. D. A. Lamb, 'Construction of a Peephole Optimizer', Software Practice and Experience, Vol. 11, No. 6, (June 1981), pp. 639-647.
8. M. L. Griss, PSL Interest Group Newsletter #4, February 1982.
9. M. L. Griss, 'A Portable LISP Compiler', Software Practice and Experience, Vol. 11, (June 1981), pp. 541-605.
10. 'The Portable Standard LISP Users Manual', The Utah Symbolic Computation Group, University of Utah, January 1982.

SAUSTALL - A New Software Environment

Robert K. Bell
Marloffsteiner Str.14c
8521 Uttenreuth
West Germany

1. Introduction

SAUSTALL (Sequential Algorithmic Universal Set-Theoretical Associative Logical Language) started as an attempt to define a new programming language. I have widened its scope to include the total software environment. It is hoped that SAUSTALL will be accepted as a general purpose language considerably better than conventional programming languages. Further, SAUSTALL attempts to overcome the conventional division of software into compilers, editors, command language, etc. It may be thought presumptuous to place before the public something which is a long way from availability. The reason for doing it is in the hope of getting reactions from the software community which will improve SAUSTALL.

2. The small computer environment

The relative merits of small computers, large computers, distributed computers, etc., have been discussed at great length in the literature. I do not wish to repeat this discussion, but there is one aspect of it which is very relevant to SAUSTALL. Small computers encourage innovation. There are three reasons for this.

1. The complexity of the systems software of large computers and the lack of good documentation makes alterations difficult. Even the correction of trivial errors can take months.

2. The commercial success of the large computer manufacturers depends largely on their success in achieving installed base migration (IBM) within their own

product line. This means that ease of transition from older to newer products must be easier than transition to competitive products. It follows that a facility once provided is seldom withdrawn and software grows more in quantity than in quality.

3. Large computers tend to be under the direct control of a computer department comprising individuals with a direct interest in the continuation of the existing arrangements and see change as a danger to their own relatively comfortable and well paid positions. If we turn our attention to small computers the picture is far less gloomy.

1. The software is simpler and often better documented. It is, in general, much easier to alter or extend.

2. Small computers are relatively new and the requirement that facilities available in current software will be available in the future is less onerous in practice.

3. Small computers tend to be under the control of individuals whose careers are not closely linked to the computer. Thus their resistance to any change which might jeopardize their career does not apply to their computer.

3. The language

The genesis of SAUSTALL lay in a certain personal dissatisfaction with existing programming languages. I do not propose to discuss existing languages in detail; in general I would suggest that existing languages do less than is possible to facilitate the production of good programs.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SAUSTALL will be a large language, in this context closer to PL/1 than to BASIC. An important design aim is genuine modularity. By this I mean that the ordinary user need not understand, or even be aware of those facilities which he does not use. All user identifiers must be declared. If a user identifier is the same as a basic word of SAUSTALL then the compiler will treat it as a user identifier. There will be some exception to this, e.g. "end", but these will be relatively few. Thus there will be a small number of reserved words, which the user must know, and a large number of basic words, which the user need not know unless he makes use of them deliberately. If a psychologist, unaware of complex numbers, introduces a variable called "complex", there is no harm done.

In my opinion a programming language should not compel a programmer to use a particular style. If the programmer does not want to use "goto" then the language does not compel him to use it. If, however, he is recoding in SAUSTALL an existing program in some other language, he may be very glad to be able to do so. Universality is a design aim of SAUSTALL. I mean by this that there are no application areas which are not considered relevant to SAUSTALL. On the other hand compatibility with existing programming languages is not a design aim. Ultimately the motivation for SAUSTALL is personal. I am trying to express in it my own ideas on programming languages, based on my own experience.

4. Variables

The conventional concept of a variable is generalized as follows:

1. Constants can be declared, e.g. constant real pi = 3.14159.
2. Asynchronous variables are allowed. They differ from ordinary variables in that they do not necessarily have a value. They are useful when executing concurrent processes. If a process attempts to read an asynchronous variable which does not have a value, then the process is held up until some other process assigns a value to the variable. If a process attempts to write to an asynchronous variable which has a value, then the process is held up until some other process reads the value. Asynchronous variables can also be tested as to whether they have a value.
3. Exogeneous variables can be assigned a value only by reading their values from a file or from the keyboard.
4. Endogeneous variables can be assigned a value only by executing an assignment statement, not by reading the value.

Consider a program containing only exogeneous and endogeneous variables. It must be possible to calculate the values of all the endogeneous variables from the exogeneous variables in a finite number of steps. If the user alters the value of an exogeneous variable then the system automatically recalculates all the endogeneous variables. Users of VISICALC will be familiar with this concept. In SAUSTALL I have tried to provide facilities similar to VISICALC for a much greater variety of data types and structures.

5. Data types

SAUSTALL will support the following data types:

1. logical.
A logical variable takes the value true or false.
 2. integer.
An integer variable takes an integer value in some given range.
 3. character.
 4. flag.
A flag takes a value from a range of named alternatives e.g. piece = king, queen, castle, knight, bishop, pawn.
 5. real.
A real variable takes a value in a specified accuracy. It is not necessarily represented in the computer by its mantissa and exponent.
 6. bit.
A bit takes the value 0 or 1.
 7. complex.
A complex variable represents a complex real number. The components have a specified accuracy and range.
 8. interval.
An interval variable represents a real interval. Addition, subtraction, multiplication, and division of intervals is possible.
 9. flag value.
Values of flags which occur in more than one flag with the same meaning must be declared as flag values. It is then possible to compare the values of flags

e.g. flag value = white, black;
flag next move = white, black;
flag piece colour= white, black;
.
.
.
if piece colour = next move then...
- Symbolic variables will also be supported, i.e. SAUSTALL will be able to perform algebra as well as arithmetic

```
e.g. symbolic real x, y, z;
x:='a+b';
y:='c+d';
z:= x+y;
.
.
.
```

6. Data structures

SAUSTALL will support the following data structures:

1. composition.

I regard a composition as a means of grouping together under a single name variables which often occur together

```
e.g.
flag marital status = single, married;
flag sex             = male, female;
composition family status = marital
                        status, sex;
.
.
.
if family status = (single, male)
then ...
```

Thus the components of a composition are identifiers which have previously been individually declared, and which can be accessed individually.

2. array.

An array is a multidimensional structure as in Fortran, except that it is not necessarily rectangular. Index $[i]$ is the value of the index of the i 'th dimension.

```
e.g. a [1:3, 1:index [1]]
```

is a lower triangular array

```
*
* *
* * *
```

The array bounds can be conditional expressions so that highly complicated arrays can be declared.

3. collection.

A collection is an unordered set of values of a given data type,

```
e.g. integer collection i (1 to 100)
```

contains some integer values from 1 to 100. To put it another way, a variable has normally a unique value. If the uniqueness requirement is removed we get a collection. Elements can be added to or taken from a collection, and the number of elements in a collection can be determined.

4. sequence.

A sequence is a set of values of a given type in a definite order. One or more elements can be removed from the beginning of the sequence, i.e. the sequence can be

split. One or more elements can be added to the end of the sequence, i.e. two sequences can be concatenated.

5. singly linked list.

A singly linked list is a set of values in a definite order. Associated with each element is a pointer to its successor. There are also pointers to the head of the list and the tail of the list. Due to the use of a pointer a list is more flexible than a sequence but requires more memory.

6. doubly linked list.

In a doubly linked list the set of values can be examined in either direction of the order. Thus each element has a successor and a predecessor, both of which can be accessed from the element. Thus a doubly linked list is an extension of a singly linked list.

7. professor.

A professor is a logical variable which can only take the value "true". Thus

```
professor smith;
logical x;
.
.
.
if x = smith then ...
```

is equivalent to

```
logical x;
.
.
.
if x then ...
```

8. matrix.

A matrix has the properties described in conventional algebra text books, and the more common matrix operations will be provided. Symmetric matrices, band matrices, and symmetric band matrices will be implemented. Matrices (other than band matrices) can also be declared with the attribute "sparse". Only the nonzero elements of sparse matrices are stored.

A possible declaration is
sparse symmetric real symbolic matrix
m[1:n,1:n];

9. ladder.

A ladder is familiar from "for" loops but has not previously, as far as I know, been suggested as a data structure. A ladder has 3 components: initial value, increment, final value.

```
ladder 1 [1,1,n] ;
.
.
.
for i: = 1 do
```

is equivalent to

```
for i:=1 step 1 until n do
```