A LISP Compiler for the DADO Parallel Computer[1]

Mark D. Lerner

Michael van Biema

Gerald Q. Maguire Jr.

CUCS-146-85

Columbia University
Computer Science Department

7 January 1985

[1]This research was conducted as part of the DADO project. It was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165 and the New York State Science and Technology Foundation NYSSTFCAT(84)-15, as well as grants from Intel Corporation, Hewlett Packard, AT&T Bell Technologies, and Nuvatec Corp.

Table of Contents

List of Figures

1. A LISP Implementation

This describes a LISP implementation for the Intel MCS-51 family of microprocessors, the processor used by the DADO parallel computer [Stolfo 83]. This LISP is the implementation language for the high-level parallel LISP (PPSL) [van Biema 84]. It is based upon the Portable Standard Lisp (PSL) [Griss 79] of the University of Utah. By writing PPSL in PSL we have achieved an significant degree of portability.

To execute PPSL on the DADO machine we developed a LISP compiler for the Intel MCS-51. Thusfar, the compiler has been used to write small parallel programs that execute on the DADO prototype hardware. The appendices include the *UNIX* shellscripts used to do this, as well as sample programs. The compiler itself runs on any system that is supported by PSL, although other aspects the DADO software environment make extensive use of UNIX.

The DADO machine is a complete binary tree. The computational component of each processing element (PE) is an Intel MCS-51 8 bit processor. Thus it was necessary to develop techniques for the execution of LISP on this hardware.

The technique is to provide an intermediate semantic level through pseudoregisters, two stacks, and extended macro commands. In addition, we generate two distinct segments. One includes *only* executable instructions, and the other includes all constants and tables.

The cross compiler has been implemented through the use of Portable Standard LISP (PSL) [Griss 79, Griss 82] developed at the University of Utah. New compiler patterns and assembler formatting routines were written. These are loaded into a running PSL, and the result is a functioning cross compiler. These changes were quite extensive: the source of the new CMACROs is 85K bytes, and the source of assember modifications is 30K bytes.

Space optimizations are presently being implemented and are described later in this paper. We hope at some time to add peephole optimization [Kessler 84].

The Intel MCS-51 processor was selected for this project because it has 4 parallel IO ports, yet the chip is not designed to execute LISP programs. Nevertheless, the processor offers considerable flexibility.

The following summarizes the processor. The reader is referred to figures 1-1 and 1-2 for diagrams.

1. Single-chip 8 bit microprocessor.
2. 128 bytes RAM on the chip (includes registers and stack) called direct RAM. Direct RAM is used to simulate additional registers, which are each 24 bits wide, to maintain a pointer to an

external stack, and for temporary locations. The direct RAM is also used for the subroutine stack and by the kernel communication primitives.

3. 4K bytes EPROM or 8K bytes ROM on the chip. This stores object code for communication primitives, as well as selected elements of the LISP runtime system (such as garbage collection and storage allocation).
4. External RAM up to 64K bytes of code, plus 64K bytes of data (DADO is currently configured with 16K bytes at each PE.) Code and data share memory.
5. Access to data in the external RAM is primarily by the 16 bit memory access register named DPTR.

Access to the onchip RAM is through a rich assortment of instructions. Program fetch is automatic from both the onchip ROM and the external program RAM.

Access to the external RAM, however, is rather difficult. The solitary data register must first be loaded with a 16 bit value. The MOVX instruction is subsequently used to transfer one byte of data. The consequent code generation problems are significant.

These architectural factors manifest themselves in three primary ways. First, at least 3 bytes are required to represent a LISP item (16 bits of data, several bits for a tag, and a bit for garbage collection). However, the underlying architecture is 8 bits. This has ramifications in the usage of registers and data manipulation.

Second, the hardware stack is allocated within the 128 bytes of onchip RAM. This memory segment must be shared with the registers and onchip memory locations. Because of space limitations, the stack cannot be used for the allocation of data (such as partially evaluated forms). The hardware stack is reserved for storage of the return addresses. Indeed, this stack is excessively small. Consequently we have written code to migrate data between direct RAM and external RAM.

Third, there is a limited amount of external RAM and hence efficient memory space usage is important.

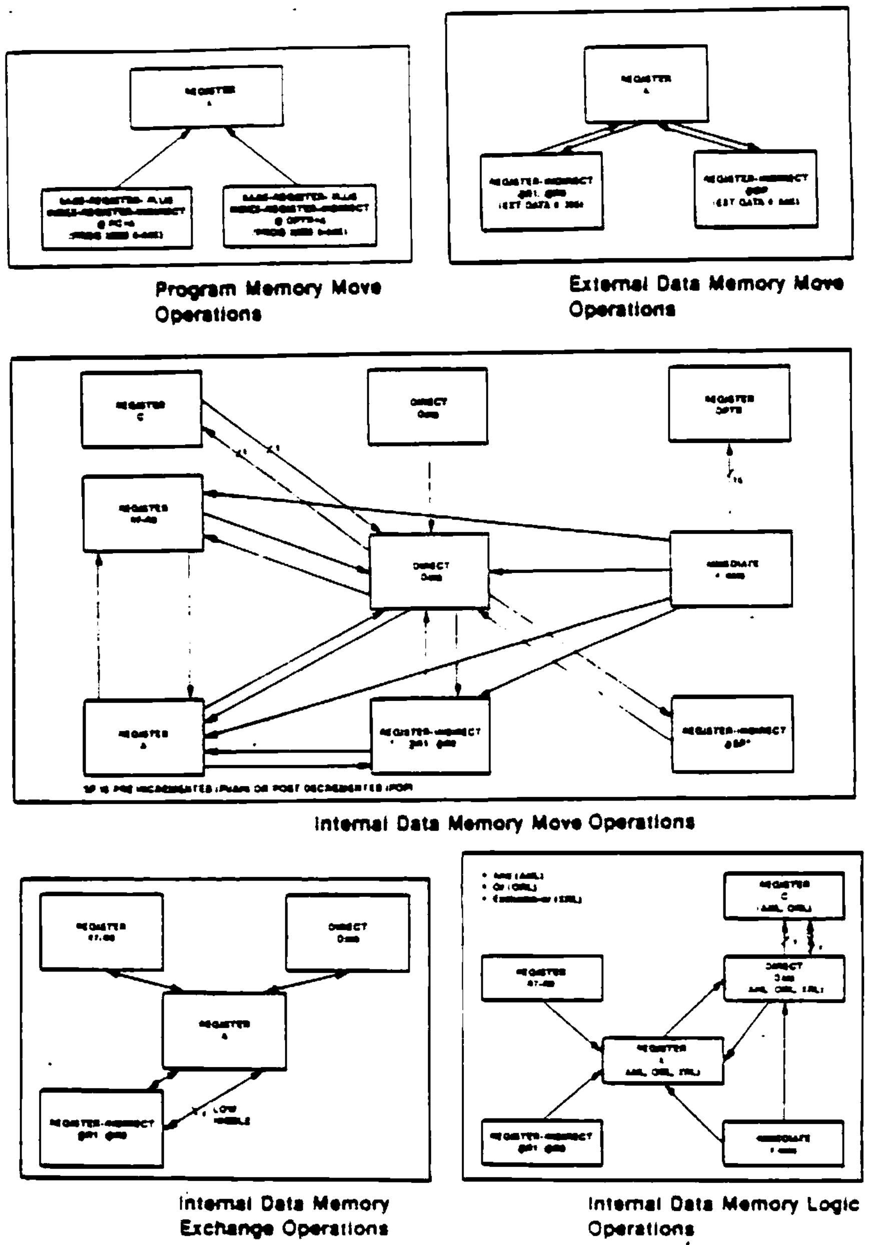
Program Memory Move Operations
External Data Memory Move Operations
Internal Data Memory Move Operations
Internal Data Memory Exchange Operations
Internal Data Memory Logic Operations

Figure 1-1: MCS-51 Memory Operations (from Intel MCS-51 Manual)

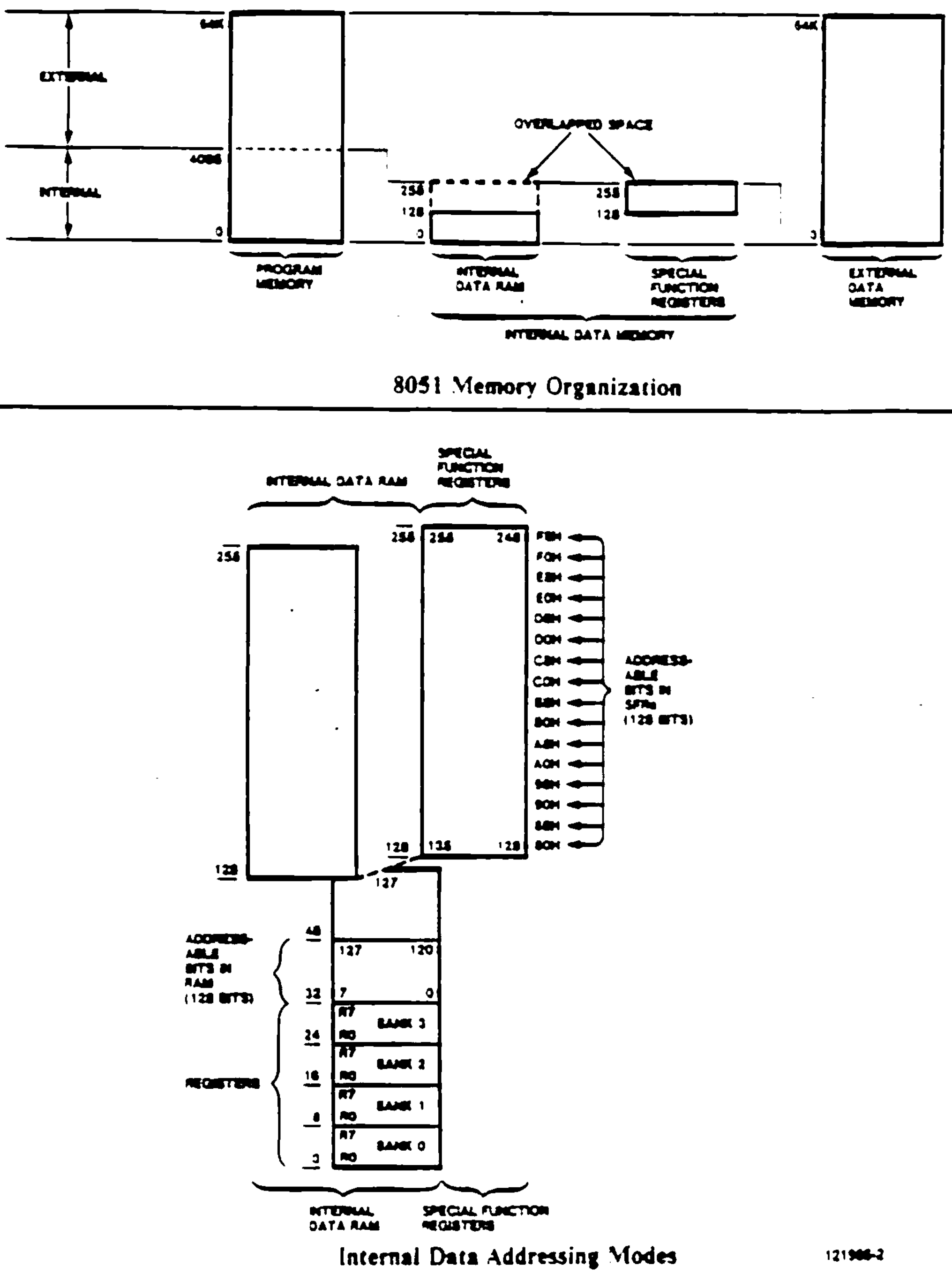

Figure 1-2: MCS-51 Memory Organization (from Intel MCS-51 Manual)

2. Implementation Techniques

The cross compiler is implemented by writing patterns for the PSL compiler. In general, the PSL compiler translates user programs (written in LISP) into a list of macros. These are composed of machine independent addressing forms (ANYREGS) and compiler macros (CMACROS). The ANYREGs represent common addressing hardware: typically this includes CAR, CDR, and stack reference. The CMACROs are used for longer instruction sequences.

The emitted macros are then recursively expanded, under control of the CMACRO definitions. Additional properties give the compiler-writer more control. For more information see [Griss 82].

To determine which address space is to be used, and consequently when the DPTR must be used, the compiler represents *compile-time* objects as tagged pairs. The CAR of such items identifies (at compile time) the type of object or the memory partition where it is assigned. The CMACROs generate appropriate instruction sequences based upon this tag. For example, (ExternalLoc X) indicates the variable X is stored in external RAM, whereas (OnChip Byte '25) indicates a byte in the processor chip at location 25. Onchip is synonymous with Intel Direct RAM.

The tags direct the compilation through a reduction process whereby code is generated and the internal representation is modified. Eventually the representation is NIL and the compilation terminates.

2.1 Assumptions and Remedies

The generic PSL compiler makes several assumptions. First, there must be at least several registers. These registers should be sufficiently large to store a LISP item. The MCS-51 registers, however, are small.

The problems of register size and memory access were overcome through the use of mechanical translation and a two-tiered CMACRO written entirely with the standard PSL tools. In essence, CMACROs were written to emulate the addressing modes of a more flexible abstract machine. This abstraction provides base/index/displacement addressing.

The code of the existing compiler was not changed. About 50 printed pages of changes were needed for the complete implementation.

2.1.1 Representation of LISP Items

The compiler represents LISP registers as abstract registers, and is thereby consistent with the PSL representation of one LISP item per register. The registers are not given a "machine-specific representation until all other processing is complete. At the conclusion of all processing the logical registers are assigned to onchip RAM locations and suitable MCS-51 constructs are emitted.

Logical PSL Register	Tag & GC	High Info.	Low Info.	Memory Space
Register 1 (3 bytes each)	R2	R3	R4	Register
Register 2	R5	R6	R7	Register
Register 3	Reg3ByteA	..	Reg3ByteC	Direct RAM
Register 4	Reg4ByteA	..	Reg4ByteC	Direct RAM
Register 5	Reg5ByteA	..	Reg5ByteC	Direct RAM
Register 6	Reg6ByteA	..	Reg6ByteC	Direct RAM
Register 7	Reg7ByteA	..	Reg7ByteC	Direct RAM
Register 8	Reg8ByteA	..	Reg8ByteC	Direct RAM
Register T1	RegT1ByteA	..	RegT1ByteC	Direct RAM
Register T2	RegT2ByteA	..	RegT2ByteC	Direct RAM

R2 ... R7 are six of the microprocessor's actual registers.

Reg3ByteA ... RegT2ByteC are symbols for 24 bytes of direct RAM.

Figure 2-1: Logical and Physical Registers

The PSL convention is to pass parameters in registers, and hence the low numbered registers are used most frequently. Therefore the PSL registers (REG 1) and (REG 2) are allocated to physical register triplets (R2, R3, R4) and (R5, R6, R7).

Eight additional PSL registers -- (REG 3) through (REG 8), (REG T1) and (REG T2) -- are allocated from onchip RAM. They use a total of 18 bytes. These bytes are accessed as rapidly as registers, yet require longer instruction sequences to specify the operand address.

The physical registers R0 and R1 have special capabilities for indirect addressing. These registers are used to pass parameters to support functions (see page 12).

The mechanism of storing LISP registers in RAM is called PseudoRegisters. The compiler representation of registers is (REG 1) ... (REG n). A translation function, GetPseudoRegister, makes the assignments of these registers to the actual hardware. There are 6 such functions. All are implemented as ANYREGs.

The first three functions are GetPseudoRegisterA, GetPseudoRegisterB, and GetPseudoRegisterC. These return the machine specific locations of the logical register. The suffix "A" retrieves the tag byte, the "B" retrieves the high information byte, and "C" retrieves the least significant byte. By use of three additional functions (named GetPseudoRegisterαLow) the compiler can generate the location as 8 bit addresses rather than 16 bit addresses.

As an example, consider moving an item from (REG 1) to (REG 2). The following CMACRO is emitted.

```
(!*MOVE (REG 1) (REG 2))
```

This is translated to 6 instructions for the MCS-51, and assembles to 6 bytes of storage[2].

```
MOV A, R2
MOV R5, A
MOV A, R3
MOV R6, A
MOV A, R4
MOV R7, A
```

2.1.2 Allocation Types

LISP items are created *and given a type* in several ways, as described below. In addition, memory is allocated by any routine that calls the CONS routine.

Function parameters
These are allocated and passed in registers, with the first parameter in PSL register (REG 1), the second in PSL register (REG 2), etc. The calling routine saves registers before it calls a subroutine. The called routine may only modify registers that are passed to it: other registers must be saved and restored. (There are two temporary registers (REG T1) and (REG T2) that can be modified without restriction.)

Prog Variables These are allocated on the external data stack as anonymous stack locations. Because this is *compiled* code, these values may be passed to functions, but not modified by them. For example, the function F1 (shown in figure 2-1) returns NIL when compiled. It returns T when interpreted, or if Y is declared global.

Global Variables These are declared as:

```
(Global '(var1 var2 var3))
```

The scope of these variables is global. They should not be bound through prog or function parameters. They are allocated in the SYMVAL table and stored in external RAM.

[2]The 8051 processor does not have an instruction to move from one register into another. Although registers can, in a limited way, be addressed by equivalent direct RAM locations, no savings can result from this (because the instruction sequence must include a byte address). As an alternative, the use of a single "move-lisp-register" subroutine would require at least 6 bytes of storage (1 byte to store the source register, 1 byte to store the destination, 3 bytes for the call instruction, and 1 byte for the return). The only way to generate tighter code would be to write 49 separate routines (for all possible pairs). This would save 3 bytes of dynamic RAM per call, but at an expense of 490 bytes if all combinations are used: enough for over 150 such operations. We plan to implement the most common of these instructions with ROM-based assembler routines.

```
(DE F1()
    (PROG (X Y)
       (SETQ Y NIL)
       (SETQ X 'Y)
       (SET X T)  %% Value of Y will not change.
       (RETURN Y)))
```

Figure 2-2: Scoping Issues: Compiled vs Interpreted Code

Fluid Variables These are not currently supported. They are treated as globals, though no error message will be given. A separate program is being developed to verify that the program does not use fluid variables or dynamic scope.

Untyped Allocations
These must be declared and manually allocated. These are *W*ord *Var*iables (WVARs) and are used in untyped code. The declaration format is:

```
(WDECLARE EXTERNAL WVAR (var1 NIL NIL)
                        (var2 NIL NIL)
                           ...     )
```

In addition to the declaration, they must be allocated by the linker/loader through the statement:

```
.DEFINE varname location
```

If the symbols are not defined the linkage editor reports this as an error.

As discussed below, there is also LISP access to direct ram.

2.1.3 Access to Direct Ram

The eight LISP registers [(REG 1) ... (REG 8)] may be directly referenced from SYSLISP with the names REG1ITEM ... REG8ITEM. In addition, the data stack pointer (described later) may be referenced at STACKTOPWORD. These entities are stored in direct RAM.

Declaration of additional direct ram variables is straightforward through three steps. The item must be declared as an exported wvar, it must be tagged is 'directitem or 'directword, and it must manually be given an address.

Support for allocation in direct RAM is provided for two simple objects: items and words. A WVAR may be given the `'DirectType` property of `'DirectItem` to indicate it is a 3 byte lisp item stored in direct ram, or `'DirectWord` for a 2 byte direct ram item. The first is needed to access the registers (needed, for example, during garbage collection). The second is necessary to access untagged external items, such as the data stack pointer.

The EXPORTED WVAR declaration may be written either in the user program, or the file `./global-data.red`. In addition, the address where the item is to be located must be manually determined and subsequently specified to the linkage editor with a `.DEFINE` statement to the linkage editor (see above). Finally, the `'DirectType` property must be set if the entity is in direct RAM.

Support includes the DirectItemP and DirectWordP predicates to recognize the above special cases. The ANYREGS DIRECTBYTE0, DIRECTBYTE1, and DIRECTBYTE2 return the memory location of the first, second and third bytes of an item. The !*MOVE macro utilizes these predicates.

2.1.4 Stack Space

The second problem is one of limited stack space. To solve this the implementation uses two stacks: a ***program stack*** and a ***data stack***. The machine's stack hardware is used to implement the program stack. Its usage is limited to the return addresses for function calls.

Function evaluation generally results in subroutine calls. These use the hardware stack to store the return address. Presently, 60 bytes of stack are reserved for function calls. This is sufficient for a depth 30 calls. More depth would be desirable, and indeed the recently announced Intel 8052 chip provides an additional 128 bytes of stack.

Stack overflow is not checked by the hardware, nor is it presently monitored by software. Instead, there are hightly efficient assembler routines which detect imminent stack overflow and migrate data between the stack and the RAM. The use of these routines is at the discretion of the progammer. They are generally insterted only into recursive routines.

Other stack operations are implemented through software. A data stack discussed on page 11 is stored in external RAM. Two bytes of direct RAM, named HTOS and LTOS, provide a 16 bit data stack pointer. Knowledge about how to access the data stack pointer is encoded into several CMACROs, as well as highly efficient assembler subroutines. The PSL compiler uses the CMACROs !*PUSH and !*POP for direct stack reference. In addition, the FRAME ANYREG is used to access a datum stored on the data stack.

2.1.5 Access to external RAM

The third problem stems from the difficulty in accessing external RAM. This must be done with the data pointer (DPTR). The DPTR is the only 16 bit register other than the PC.

The need to access external RAM is determined from the tags, and the special MOVEX CMACRO manipulates external memory. This corresponds to the underlying hardware restriction that external RAM is accessed by the MOVX instruction.

It is important to note these manipulations depend upon whether the external location is the source or destination. Although PSL does not identify the CMACRO operands as source or destination, CMACROS have been devised to distinguish between source and destination[3]. Such information is necessary on the MCS-51 because the addressing techniques are quite primitive. For example, it is necessary to preserve intermediate results when moving between external memory locations, as in:

```
    (SETQ X (CAR Y)) == (SETQ (ExternalLoc X) (CAR (ExternalLoc Y)))
```

In general, a compiler may be viewed as a mechanized technique to change the semantic level of valid programs in the input language. When the input is LISP and the output is for a 8 bit microprocessor, the disparity in levels causes particular problems. Although LISP is usually executed on a 32 or 36 bit architecture, often with specialized addressing hardware, these features can be implemented with appropriate CMACRO definitions. The code generation phase of the compiler provides a translation from the abstract LISP machine (ALM) level to the target machimne level. In addition, this phase also contains target machine to target machine transformations, which are used to introduce various optimizations.

[3]This is done by using appropriate combinations of predicates.

3. Memory: Stacks and Conses

3.1 Stacks

The implementation uses two stacks. The first is used to save return addreses when a subroutine is directly called; this is directly supported by the hardware (by CALL and RET instructions). The second stack is used to store data frames when subroutines are entered. This is supported by software.

The addressing hardware is used soley for the purpose of subroutine calls. The hardware stack pointer (SP) is initialized during the standard start of program to a value defined during the linkedit process. The linkage command:

```
.DEFINE STACKPOINTER
60H
```

sets the stackpointer to 60.

The second stack is the *data stack*, used for the storage and access of frames during procedure execution. The stack is accessed at the CMACRO level by the !*PUSH and the !*POP CMACROS.

The stack is stored in external RAM. The direct RAM locations HTOS and LTOS maintain the value of the stack pointer, and are copied into the DPTR when the stack is accessed. The initial value of the stack is defined by the linkage editor symbol STACKTOP, and the variable STACKTOPWORD references the value during program execution.

The stack grows down. When a 3 byte item is stored on the stack, the lower addressed bytes are stored at lower absolute stack locations. This scheme is based on architectural considerations. There is an instruction to increment the data pointer, but there is no corresponding decrement instruction.

The following statements retrieve one byte from the stack:

```
MOV DPH,HTOS  ; Move direct ram location 41 to data-pointer high.
MOV DPL,LTOS  ; Move direct ram location 42 to data-pointer low.
MOV A,@DPTR   ; Indirect move from data pointer to accumulator.
```

The assembly code is shown simply to illustrate how the machine actually works: in practice this is done automatically. For example, the CMACRO sequence:

```
(!*PUSH (REG 1))
(!*MOVE (REG 2) (REG 1))
(!*POP (REG 2))
```

swaps the contents of (REG 1) with (REG 2).

Assembly routines provide for efficient stack access. These are used automatically by the CMACROS, and can be accessed via LAP. The seven routines are: TOSTACK, FROMSTACK, MOVESTACK, XPUSH, XPUSH1, XPOP, XPOP1. The arguments to these functions are described below, as are details of the functions.

The calling convention is that all routines communicate with the machine's R0 and R1, which are destroyed. R2...R7 are reserved because they store the PSL registers (REG 1) and (REG 2).

R0 is a logical offset from TOS, i.e. the value is multiplied by 3 in these routines to account for the 3 bytes/item. R1 is the absolute address in onchip ram, except in MOVESTACK where it too is a logical offset from TOS. By storing logical offsets, rather than premultiplying at compile time, the compiler can address a larger offset from the stack. This violates the conventional wisdom of performing such operations at compile time, but it does so because of the larger concern of accessing as much of the stack as possible.

TOSTACK	copies 3 bytes from direct RAM to an offset on stack.
FROMSTACK	copies 3 bytes from an offset on stack to direct RAM.
MOVESTACK	copies 3 bytes from the stack to another place on the stack.
XPUSH1	pushes the 3 bytes addressed by R0 onto the stack.
XPOP1	pops the 3 bytes addressed by TOS into memory addressed by R0.
XPUSH	pushes the 2 bytes in R0/R1, then pushes a 16#0 (i.e., zero).
XPOP	discards one byte from the stack, then pops 2 bytes into R0/R1.

Figure 3-1: Stack Access Routines

3.2 Conses

Cons cells are allocated from a reserved area of memory. The program executes at startup the MAKEFREELIST function to initialize this area. Each cell consists of 6 bytes, structured as a linked list, as follows:

```
==========================================================
| <========== car ==========> | <========== cdr ==========> |
| tag1 | byte1high | byte1low | tag2 | byte2high | byte2low |
| <=========== 0 ===========> | <=== address next pair ===> |
==========================================================
```

Figure 3-2: Cons cell layout

The bytes NEXTFREE and NEXTFREE+1 contain the address of the next available CONS cell. The CONS routine returns the first pair and updates the NEXTFREE pointer. A zero in the nextfree pointer indicates that no CONS cells remain.

The garbage collection routine puts cells back onto this chain. The routine is a non-compacting garbage collector. It does a non-recursive search of all accessible cells, and relinks unused cells onto the free chain. It is limited to garbage collection of fixed-sized items. In particular, this allocation scheme requires modification to support vectors of items.

4. Operating Instructions

There are 4 phases to program execution. The first executes on any system that runs PSL, whereas the subsequent steps work only on Unix systems.

Cross compilation	This uses the cross compiler features of PSL. The 8051 cross compiler has been executed on Dec20, Vax 750 (BSD Unix), and HP series 200 (running Pascal Workstation System).
Assembly	This is done with the Nuvatec 8051 cross assembler, and runs only Unix.
Link edit	This is done with the Nuvatec 8051 linkage editor, and runs only under Unix[4].
Execution on hardware	This uses either the DADO 1.75 Prototype, or the Intel Development system.
Alternatively, execution on software simulator	This is done with a C-language based 8051 simulator which is executed under UNIX.

Figure 4-1: Cross Compilation Steps and use of UNIX

The following are several programs that have been executed on the MCS-51 with this compiler. Assembly and linkage editing were performed by the Uniware tools donated by Nuvatec Inc.

The programs have been executed on an Intel ISIS-II Microcomputer Development System with an In Circuit Emulator (ICE). They have also been executed with an 8051 simulator written in C and executed under Unix. The simulator is an indispensable development tool; its flexible features include formatting of s-expressions. In addition, the use of the simulator allows many simultaneous users, thus avoiding some bottlenecks due to the scarcity of DADO machines.

[4]This phase adds various utility functions and the details of DADO internals to the user code. In particular, the files **kernel.s**, **mathlib.s** and **stacks.s** are searched. The first of these contains entry points and memory allocation to interface with the DADO kernel. The **mathlib** contains 16-bit precision arithmetic, and the **stacks** includes routines for efficient stack manipulation.

4.1 Simulator for the MCS-51

A simulator for the MCS-51 simplifies program development. The program simulates all instructions of this Intel processor. The 8051 supports two external memories: one for code and one for data. In the DADO machine code and data share the same memory partition. Likewise the simulator supports either one bank or two. The 8051 has several kinds of onchip memory. Users are most concerned with *direct* RAM, and occasionally with special function registers (sfr). Moreover, the 8052 has additional RAM that may be accessed only indirectly. These are all supported by the simulator.

The simulator will execute until any of the following conditions occur:

- It encounters a breakpoint, either at the beginning or as part of an instruction
- It executes the undefined opcode A5.
- It exceeds the bound on number of instructions. This can be set with the N command (see below) and provides a mechanism to capture control of wayward simulations.

The simulator does not provide complete support of hardware features such as timers, interrupts, or latched ports. These facilities may be accessed at the instruction level, but not the hardware level. For example, the timer register can be set (by a move into the appropriate address) but the timer will not be decremented and will not generate an interrupt to the simulator. These features are not used by the compiler and thus do not require simulation.

The following commands are available:

Break	sets a breakpoint. Control is returned to the console when a breakpoint is encountered in any byte of an instruction.
Clear	clears a break point.
Examine	a memory location of external RAM. The user provides start and end addresses. The memory between these bounds is dumped.
eXamine	first shows values (in hexadecimal and decimal) for the currently selected register bank, the program counter, the data pointer, and the accumulator. Under user control, the program then displays a range of values from direct onchip RAM or special-function RAM.
Debug	toggles debug mode (for internal usage)
Go	begins the simulation of the program
Insert	a value in a location

List	lists all break points
Number	sets number of instructions to execute. The simulator will enter the break loop when this number is exceeded.
Pc	sets the program counter
Step	toggles single stepping mode. When in effect, the program executes one instruction at a time, and preceeds execution with a display of the instruction and operands.
Y	prints out an item. If the item is a pointer it will print it as an S-expression. Accepts either a memory address, the literal 'r1' or 'r2.' The latter indicate the item's address is stored in a PSL-register. Other registers will be added later. The output includes the print-name of atoms. Numerics are printed in hex.
Z	prints out a s-expr of a cons-cell. Accepts a memory address, 'r1,' or 'r2.'
?	prints a list of commands with brief descriptions.

5. Optimization Techniques

There are two kinds of optimization used in the compiler First is optimization of jump instructions. Second is a more general technique for the compation of code

5.1 Jump Optimization

The 8051 instruction includes several jump commands. Short jumps are generally 2 bytes long, whereas long jumps are 3 bytes. The compiler emits short jumps in comparison instructions. A function (fixupjumps) verifies that the destination of the jump instructions is accessible by the instruction which is used. New code is generated when needed to guarantee that operands are within the range of their opcodes.

5.2 Code Compaction

An analysis and optimization tool has been developed which identifies repeated code segments and reduces them to shorter sequences. This involves representing the code sequence as a list where repeated sequences are replaced by references to the original instance of the sequence. This technique is defined recursively, so that each repeated sequence within the program is represented by a reference to a unique instance.

As a tool to analyse the code, this information is used to identify changes to CMACROs. For example, the sequences may indicate that more specific CMACRO cases should be written, or that temporary results should be saved.

As a space optimization tool to be applied after the above optimizations, new sequences which exceed a given length should automatically be revised into a subroutine that is called by the references to the sequence. The tool presently indicates the length of each repeated code sequence, thus it is straightforward to modify the code list by insertion of labels and control intructions.

The computation is quite lengthy, as shown below. Techniques are currently under development to improve the performance of this initial algorithm. The current ideas are to improve the algorithm to avoid recomputing pattern matches at each cycle of the algorithm, and to limit the number of input states. In addition, the optimization can be applied on a subroutine-by-subroutine basis rather than to the complete code.

Experimental results have been obtained, as follows. The number of input states is the number of source lines to be optimized. The number of output states is the number of *distinct states* that result, where each state may consist of one or many statements. The composite output length is the sum of the sizes of all outputs.

Program	Number Input States	Number Output States	Composite Output Length	Percent Reduction	Time hh:mm:ss
Main program in example	457	211	369	19%	3:22
Complete example	2221	789	1870	16%	2:38:31

Figure 5-1: Preliminary Optimization Results

5.2.1 Optimization Details

The main routine is *findloops*. This simply executes the ***preconv*** function to initialize the data, and then repeatedly executes *conv1* and *conv2* as long as they can further compact the code.

The preconv function changes the input program into an initial symbolic form. Each input line is replaced by a symbol. Equal inputs are assigned the same symbol. The conv1 function replaces a pair of equal inputs with a new input *if* the pair occurs again in the file. The conv2 replaces such pairs even when they do not recur.

5.2.2 Optimization Code

Following is the main loop of this optimization:

```
(de findloops
    (input)
    (prog (original result)
          (setq original (preconv input))
          more
          (setq result (conv1 original))
          (setq result (conv2 result))
          (and (equal original result) (Return result))
          (setq original result)
          (go more)))
```

6. Limitations, Unimplemented Features

A number of conventional LISP features have not been provided in this implementation. Two kinds of features were omitted, those which require large amounts of memory, and features which can be more efficiently executed by the host computer. Since DADO functions as an attached processor it is reasonable, for example, to use the host computer to store parts of the symbol table.

1. The EVAL function is not supported. However, the APPLY function is supported and allows indirect function invocation of compiled functions.

2. The symbol table consists of ***name***, ***value***, ***property***, and ***function*** cells. A compile-time option selects which of these cells are allocated in the DADO processing elements (PE). Presently we do not allocate either ***name*** or ***property*** cells in the PEs. Thus, the APPLY function works (since it uses a symbolic identifier to represent a token) but the printname is only available in the host. The GET and PUT functions cannot be used because they would require the property cell. These omitted features will be available by communication with the host processor.

3. Static scoping is used: fluid variables are not supported.

4. Presently certain assembler symbols are reserved and may not be used. This condition is diagnosed by the assembler. The symbols include: A, B, C, DPTR, R0, R1, R2, R3, R4, R5, R6, R7, PC, as well as the MCS-51 mnemonics: ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA, ANL, ORL, XRL, XRL, CLR, CPL, RL, RLC, RR, RRC, and SWAP. There are predefined variables as well: REG1ITEM .. REG8ITEM are LISP aliases for the machine registers, STACKTOPWORD is the alias for the data stack pointer. The following symbols have the special meaning described below.

The following are special symbols. They are assigned values by the linkage editor `.DEFINE variable value` statement. The sample program compilation shows how this is done.

BOTTOM	is the lowest address available in the hardware
TOP	is the highest address available in the hardware. It is assumed that all memory between BOTTOM and TOP is available, i.e. that there are no gaps.
STACKPTR	is the initial value of the hardware stack pointer. On the 8051 and 8751 the stack goes from the value of STACKPTR up to 0xFF. On the 8051 the stack goes up to 0x1FF.
STACKTOP	is the initial location in external ram of the data stack pointer. The stack will grow *down* as it is used.
EXTRAREG	is the first external ram location to store extra registers. When the compiler needs more than 8 registers, it will simulate them beginning at this location.
EXTRAREGTOP	is the last external ram location for extra registers.
NEXTFREE	is the location of the pointer to the next available cons cell.
STARTFREE	is the lowest address of the space reserved for cons cells.
ENDFREE	is the highest address of cons cell space.

Figure 6-1: Special Symbols Defined at Link Edit

7. Tags

The following are internal representations of data objects in the compiler. While, in general, the CMACROs hide the details of the target machine, the tags and predicates emphasize the details of the target machine.

Internal Form	Meaning
(constant xxx)	Constant
(externalloc xxx)	16 bit address constant
(immediate xxx)	Constants or internal compiler item
(indirect register xxx)	Indirect register, as in @r0 or @DPTR.
(label xxx)	A machine-independent label
(lowbyte RAM xxx)	8 bit constant: part of an address
(onchip accumulator a)	The accumulator of the machine
(onchip bit xxx)	A bit address
(onchip dptr)	The data access register
(onchip label xxx)	8 bit constant: part of an address
(onchip RAM xxx)	An address of onchip memory
(onchip register xxx)	A physical register of the MCS-51 processor
(reg xxx)	A machine independent register
(wvar xxx)	A SYSLISP variable: may have the property 'DIRECTTYPE with value 'DIRECTITEM or 'DIRECTWORD if located in onchip RAM.

8. Predicates

The following predicates are used within the CMACROs. They are used as selectors within CMACROs to specify the particular code to generate. Those marked with [+] are unique to the MCS-51 implementation.

Predicate	When True
Anyp	Always
+ BytePattern000	(And 0 operand) is an identity operand
+ BytePattern001	(And 0xFF operand) is the identity
+ BytePattern010	(And 0xFF00 operand) is the identity
+ BytePattern100	(And 0xFF0000 operand) is the identity
+ DirectItemP	Operand is a 3 byte quantity of onchip RAM
+ DirectWordP	Operand is a 2 byte quantity of onchip RAM
DispInumP	Operand is a displacement
+ DptrP	The real machine data access register (DPTR)
+ ExtraRegP	The extraregister mechanism is to be used
+ FixedSymbolP	Numeric or symbolic constants
InumP	Numbers
+ IndirectP	Indirect addressing
+ SymValLocP	Tests for global and fluid variables
LongInumP	Tests for large numerics
+ MovexCaseOneP	Tests for external access using a general fixup function
+ MovexCaseThreeP	Tests for external access as register/displacement
+ MovexCaseTwoP	Tests for external access for base addressing
+ Neg16p	Equals -16
+ Neg8P	Equals -8
Negp	Any negative number
+ NilRegP	The special register that contains nil
+ NotNilRegP	Any register that is not nil
+ OnchipAccP	Representation of the real machine accumulator
+ OnchipP	Any datum stored on the onchip
+ OnchipRamP	Datum stored in a real machine onchip RAM location
+ OnchipRegP	Datum stored in a real machine register
OneP, TwoP, etc.	Numeric tests
+ Pos255P	Numeric test
PosAmt	Numeric test
QuoteP	A quoted item
RegisterP	A register
SmallInumP	A one-byte numeric
+ StackDisplaceP	A reference to external RAM from the data stack
ZeroP	Numeric test

Many of the predicates form a hierarchical structure, as shown below:

```
                              AnyP
              ------------------------------------------------
                 /               /   |          \                \
             NumberP           QuoteP |       RegisterP         OnchipP
          BytePattern000             |        /    \              |
          BytePattern001             |   NilRegP NotNilRegP       |
          BytePattern010             |                            |
          BytePattern100             |                  ------------------------
            SmallInumP               |                  /           |              \
            DispInumP                |              DptrP  DirectItemP  OnchipAccP
              InumP                  |                     DirectWordP  OnchipRegP
           -----------          -------------------        IndirectP    OnchipRamP
          /     |      \          /                  \
      NegP    ZeroP  PosAmt  MovexCaseOneP        Others:
      Neg16P         Pos255P MovexCaseTwoP           ExtraRegP
      Neg8P          OneP    MovexCaseThreep         FixedSymbolP
                     TwoP                            SymValLocP
                                                     StackDisplaceP
```

9. CMACROs

The following is a description of the 55 CMACROs that have been defined for the MCS-51 cross compiler. Most CMACROs manipulate objects which represent LISP items. They accomplish this by emitting instructions and using other CMACROs for simplification. CMACROs are designed as Abstract Lisp Machine opcodes.

Very knowledgeable users can write code directly in LAP (the LISP Assembly Language) using CMACROs and machine specific opcodes. By using CMACROs the user can avoid many of the details concerning addressing.

The move CMACRO has more general capabilities, and may also be used from LAP. It can manipulate machine specific items such as the registers, the data pointer, and onchip memory locations. It is assisted by the MOVEX CMACRO, which is the repository of information about external memory access.

In addition, there are 9 CMACROs (subb, add, addc, anl, cjne, clr, inc, orl, xrl) which represent single MCS-51 instructions.

Some CMACROs, such as !*MOVE, !*WOR, !*WAND and !*WPLUS2 are optimized for the special case where only one byte of a LISP item is affected by the operation. This test is made by the bytepattern predicates and numeric predicates described above. The CMACROs found only in this cross compiler are designated with [+].

	CMACRO	Purpose
+	*ADD	Provides CMACRO access to the MCS-51 ADD instruction.
+	*ADDC	Provides CMACRO access to the MCS-51 ADDC instruction.
+	*ADDRESSCOMP	Does address computation of index/displacement access to external RAM.
	*ALLOC	Allocates RAM on external data stack.
+	*ANL	Provides CMACRO access to the MCS-51 ANL instruction.
	*CALL	Calls an entrypoint as a subroutine.
+	*CJNE	Provides CMACRO access to the MCS-51 CJNE instruction.
+	*CLR	Provides CMACRO access to the MCS-51 CLR instruction.
	*DEALLOC	Deallocates RAM from external data stack.
	*EXIT	Deallocates RAM and return from a procedure.
	*FIELD	Select and right-justify a subfield of a LISP item.
+	*FROMSOURCE	Moves data from external RAM to direct RAM. See *Todest.
+	*INC	Provides CMACRO access to the MCS-51 INC instruction.
	*JCALL	Jumps to an entry point when return value not needed.
	*JUMP	Unconditional jump.
	*JUMPEQ	Jumps when two arguments are equal.
	*JUMPIF	Generalized conditional jump CMACRO.
	*JUMPNOTEQ	Jump when two arguments are not equal.
	*JUMPNOTINTYPE	Jump when argument is out of a given range of types.
	*JUMPNOTTYPE	Jump when argument is not of a given type.
	*JUMPTYPE	Jump when argument is of a given type.
	*JUMPWGEQ	Inline code to jump when greater or equal.

	*JUMPWGREATERP	Inline code to jump when greater.
	*JUMPWLEQ	Inline code to jump when less than or equal.
	*JUMPWLESSP	Inline code to jump when less than.
	*LINK	Calls intralanguage procedures.
	*LOC	Computes and loads the effective address of the item.
	*MKITEM	Combines a tag and datum to create a LISP item.
	*MOVE	The foundation of the cross compiler. Moves data between any source and destination. See detail below.
+	*MOVE3ITEMSA	Move from SymVal table (named binding) to a register.
+	*MOVE3ITEMSB	Move a register to the SymVal table (named binding).
+	*MOVECONSTANT	Moves a tag and constant into destination.
+	*MOVEX	Perform efficient address computation and access to external memory. See detail below.
+	*ORL	Provides CMACRO access to the MCS-51 ORL instruction.
	*POP	Pop data from the data stack.
	*PUSH	Push data onto the data stack.
	*PUTFIELD	Stores a field of one argument into a field of another.
	*SIGNEDFIELD	Unimplemented (normally a field extract with sign extension).
+	*SUBB	Provides CMACRO access to the MCS-51 SUBB instruction.
+	*TODEST1	Store temp1 result into external RAM defined by temp1. Used by movex. See *fromsource.
+	*TODEST2	Store arbitrary onchip result to any external RAM.
+	*TODEST5A	Copy external constant (wvar) into a register.
+	*TODEST5B	Copy a register into external constant storage (wvar).
	*WAND	Inline code of logical and of arbitrary arguments.
	*WDIFFERENCE	Inline code of numeric difference between tagged items.
	*WMINUS	Unimplemented: Handles unary minus, as in x:=-y.
	*WNOT	Unimplemented: Handles x:=Not(y);
	*WOR	Inline code of logical OR.
	*WPLUS2	Inline code of numeric sum between tagged items.
	*WSHIFT	Shifts one argument specified number of bits. Simple and frequent cases are coded inline.
	*WTIMES2	Unimplemented: use (!*link times2) instead
	*WXOR	Inline code for exclusive or operation.
+	*XRL	Provides CMACRO access to the MCS-51 XRL instruction.

10. The MOVE CMACRO

The MOVE CMACRO is the foundation of the compiler. All other CMACROs can process only some operands, whereas the MOVE CMACRO can handle all operands. Therefore other CMACROs frequently invoke the MOVE CMACRO to coerce their operands into a usable form, then recurse on themselves, and finally use MOVE to restore the form of the operands.

The MOVE explicitly handles the following cases. The order of consideration is important. Specific tests which produce very clean code are placed before more general cases which produce less efficient code. The CMACRO moves two operands, as shown in the following table and subsequently described[5].

The numbers in figure 10-1 correspond to the following operations:

1. Equal operands
2. One onchip operand into another onchip operand
3. A small number into an onchip operand
4. Zero into a PSL-register
5. Small number into a PSL-register
6. Number into a PSL-register
7. Global variable (ie, stored in the SymVal table) into a PSL register
8. PSL register into storage of the SymVal table
9. Quoted item into a register
10. Data pointer into a PSL register
11. PSL register into the data pointer
12. PSL register to PSL register.
13. Indirect machine operand into a onchip register
14. Indirect machine operand into any onchip location
15. Moving any other operand into an onchip location (if new tags are created this may have to be updated)
16. Machine register into an indirect location
17. Onchip memory into an indirect location
18. Between two offsets from the data stack
19. Offset on the data stack into a PSL-register
20. Offset on the data stack into any operand
21. PSL-register into the data stack
22. Any operand to the data stack
23. External static symbol to a PSL-register
24. PSL-register to an external static symbol
25. Any operand to an external static symbol
26. Small number to an extra-register (i.e., when more than 8 registers are needed the machine will use RAM to simulate the additional registers: this is expensive but works)

[5] In addition to the table, support has recently been added for *DirectItemP -> RegP, DirectWordP -> RegP, RegP -> DirectItemP*, and *RegP -> DirectWordP*.

PSL STATE TABLE

Legal Operand Transitions

source ↓ destination->	equal items	PSL register	extra reg	stack offset	SymVal Table	Extrn static	onchip reg	onchip	indirect reg	data pointer	other
equal	1	X1	X1	X1	X1	X1	X1	X1	X1	X1	X1
PSL register	X1	12	29	21	8	24				11	
extra-register	X1	28	30	X22		X24					
stack offset	X1	19	X20	18	X20	X20	X20	X20	X20	X20	20
Symval table	X1	7	X31	X22		X24					
Extrn static	X1	23	X31	X22		X24					
onchip reg+	X1		X31	X22		X24	X2		16		
onchip+	X1		X31	X22				2	17		
indirect reg+	X1		X31	X22		X24	13	14			
data pointer+	X1	11	X31	X22							
quoted item	X1	9	X31	X22		X24					
zero	X1	4	X31	X22		X24					
small number	X1	5	26	X22		X24	X3	3			
number	X1	6	27	X22		X24					
other	X1	X15	31	22	X15	25	X15	15	X15	X15	33

```
+  indicates machine operand.
X# indicates subsumed state,
   as in X1 is subsumbed 1.
```

Figure 10-1: Supported *MOVE operands

27. Number to an extra-register
28. Extra-register to a PSL-register
29. PSL-register to extra-register
30. Between two extra-registers
31. Anything else to an extra-register
32. An immediate operand into anything
33. Anything else (within limits described below)

If the macro cannot processes the data it will pass its operands to the MOVEX CMACRO. In this last case it is presumed that at least one operand is in external RAM, and hence the MOVEX CMACRO must be used. Before invoking MOVEX the operands are modified by a general *fixup* function, and in addition an operand-specific *FixUpFn* is applied if there is one.

The intention of the fixup function is to parse the operand into the index register, base register, and displacement before the MOVEX CMACRO is invoked.

11. The MOVEX CMACRO

The MOVEX CMACRO assumes that at least one of its operands has been processed by a fixup function. It produces code for 13 types of operand pairs. In addition, it generates an error message if it could not produce code. The error

```
!*MOVEX FOUND UNEXPECTED OPERANDS
```

will appear in the output file when either MOVE or MOVEX was unable to process its operands. It may indicate a compiler bug which must be remedied by changing a CMACRO[6].

The fixup functions provide lists to the MOVEX CMACRO. These lists begin with the token *ONE*, *TWO*, or *THREE* to identify the type of fixup which was performed, and indicates further processing requirements.

The *ONE* prefix is only produced when a FixUpFn has been applied. It indicates the use of index/base/displacement addressing. This facilitates address computation as the sum of 3 values: the contents of an index register, plus the contents of a base register, plus a displacement. Any field may be unused thus there are 8 cases. Code is generated for these by the *ADDRESSCOMP CMACRO.

The *TWO* prefix indicates the operand represents a static external memory location. This allows constants or fixed offsets from constant arrays. This case is processed by the AddrFieldExternal ANYREG.

The *THREE* prefix indicates an operand which is either a base address, or a base address plus constant offset.

The following specific cases are processed by the MOVEX CMACRO:

1. Case 3 into a PSL register
2. Register into case 3
3. Small number into case 3
4. Numeric into case 3
5. Case 2 into anything
6. Case 1 into Case 1
7. Register into Case 1
8. Extra register into case 1
9. Small number into case 1
10. Number into case 1
11. Stack offset into case 1
12. Case 1 into a register
13. Case 1 into anything.

[6]Since the *MOVE employs *MOVEX when operands are not resolved the error cannot be diagnosed by the *MOVE macro.

12. LAP Programming

Programming in LAP is necessary when the most precise control over the machine is necessary. An alternative to LAP programming is to use the SYSLISP extensions of PSL.

LAP code is written by writing a CORRECT list of CMACROs. The list must begin with the identifier LAP for this to work. **The LAP programmer must be absolutely certain to write code free from errors.**

All functions and routines have parameters passed in the registers, with the first parameter in PSL register (REG 1), the second in register (REG 2), etc. The function value is returned in PSL register (REG 1).

The following is an example of lap code:

```
(LAP '((!*ENTRY FASTAPPLY EXPR 2)
       (!*PUSH (REG 1))                   %% Save item pointer.
       (!*MOVE (REG 2) (REG 1))           %% Get address ...
       (!*LINK GETFNCENTRYPOINT EXPR 1)   %%    of routine ...
       (!*MOVE (REG 1) (REG 2))           %%       remember it.
       (!*POP (REG 1))                    %% Restore item ...
       (!*MOVE (REG 2) (REG DPTR))        %%    prepare to ...
       (!*CLR (ONCHIP ACCUMULATOR.A))     %%       jump
       ("JMP @A+DPTR")))                  %% Do it.

(LAP '((!*ENTRY GETFNCENTRYPOINT EXPR 1)
       (!*MOVE (WCONST 3) (REG 2))
       (!*LINK TIMES2 EXPR 2)
       (!*MOVE (EXTERNALLOC "#HI SYMFNC") (ONCHIP RAM DPH))
       (!*MOVE (EXTERNALLOC "#LO SYMFNC") (ONCHIP RAM DPL))
       (!*WPLUS2 (REG 1) (REG DPTR))
       (!*EXIT 0)))
```

Figure 12-1: Sample LAP Code

The above routine first saves a PSL register on the stack. It then moves the constant 3 into the register and invokes TIMES2 to multiply register 2 by 3 (since there are 3 bytes per LISP item). It then moves the location of the function-table into the datapointer (since in this particular instance it can be used as a temporary register) and adds the previously computed offset.

The effective address computation continues. The result of the addition is moved into register 2, and this is followed by restoration of the register. Next the accumulator is cleared and register 2 is moved into the data pointer in preparation for an indirect jump. The final line is a quoted string -- it does not involve the expansion of a CMACRO -- and is simply emitted as code.

13. Porting Between Version 3.2 and Version 3.3

The HP PSL implementation made many extensions to the Utah implementation. Hence it required some work to port the compiler to the HP 9836 workstation (series 200). However, this port was accomplished without modifying any CMACROs.

There were two main problems: First, the modifications to the assembler formatting phase of the process had to be moved into a copy of the system-supplied formatting statements. This was apparently required in order to provide proper variable bindings. These modifications format code according to MCS-51 requirements.

Second, numerous properties had to be changed to be compatible with the original implementation. In particular, the 'OPENFN property indicates that an open function mechanism should be used prior to the CMACRO phase, and the 'ANYREG property must be present for an ANYREG to be used. Many 'OPENFN properties had to be removed, and several 'ANYREG properties had to be added.

To guarantee full compatibility, with the original version, the compiler patterns and compiler source of the PSL3.2 compiler [Utah] are also used in the version which runs on the HP workstation.

The modifications made to the compiler at Hewlett Packard (Palo Alto) include significant improvements, yet the above process ignores them. We hope at a future time to be more precise in analysis of the differences between the Utah release and the HP release, so as to exploit the more optimal code sequences that the HP version can produce.

The following code was used to make the cross compiler run on the HP series 200 workstation.

```
(on verboseload) (setq pathin!* loaddirectories!*)
(load compiler rlisp pathin
        syslisp zboot if-system useful)
(dskin  'psl2:compiler.sl')                      %% The UTAH compiler
(reload  hp-comp hp-lap lap-to-asm)
(dskin  'psl2:rem-old-decl.sl')                  %% Listed below
(reload  hp-cmac)
(reload  hp-asm)                 %% Includes 8051-specific assembler
                                 %% which must be included in the
                                 %% primary source file for proper
                                 %% variable bindings.
(dskin  'psl2:rem-old-decl.sl')
(load    8051-cmac)              %% The 8051-specific CMACROs
(setq    lastactualreg!& 5)
(setq    maxnargs!&      15)
(dskin  'setprops.sl')           %% Maintain the property list of PSL 3.2
(flag  '(immediate indirect) 'terminaloperand)
(channelprintf stdout!* 'Ready to dump.%n')
(off     usermode)
(savesystem
 (bldmsg '%w %w' '8051 Cross compiler, version'  (date))
 'MYPSL:8051-CROSS.DUMP' ())
(quit)
```

The system will generate a PASCAL error message after the savesystem returns and execution begins. Ignore the error message.

The following code is psl2:rem-old-decl.sl. It removes the HP9836 specific optimizations, and substitutes general UTAH-PSL definitions.

```
(setq proplist '(ANYREG ARGUMENT-COUNT CMACRO COMPFN CONST DOFN EMITFN
 EMITFN EXTVAR FLIPTST GROUPOPS LONGBRANCH MATCHFN MEMMODFN NEGJMP
 NEGJMP ONE OPENFN OPENTST OPTFN PA1ALGFN PA1FN PA1REFORMFN REG SUBSTFN
 SUBSTFN TERMINAL TRANSFER UNKNOWN UNMEMMOD UNMEMMOD VAR ZERO))
(de propremover(x) (foreach y in proplist do
                           (remprop x y) (remflag1 x y)))
(mapobl 'propremover)
(dskin 'psl2:pattern.sl')                 %% The Utah compiler patterns.
(dskin 'psl2:comp-decls.sl')              %% The Utah compiler declarations.
```

The following code is setprops.sl. It removes additional properties that are not necessary for the cross compiler. It probably could be merged with the above file.

```
(setq openfnlist '(*ADDMEM *MPYMEM ATOM BIGP BYTESP CODEP EQ FIELD
FIXNP FIXP FLOATP HALFWORDSP IDP INTP LOC MKITEMREV NE NEGINTP NOTBIGP
NOTBYTESP NOTCODEP NOTFIXNP NOTFIXP NOTFLOATP NOTHALFWORDSP NOTIDP
NOTINTP NOTNEGINTP NOTNUMBERP NOTPOSINTP NOTSTRINGP NOTVECTORP
NOTWRDSP NUMBERP PAIRP POSINTP PUTFIELDREV RPLACA RPLACD SETQ
SIGNEDFIELD STRINGP VECTORP WAND WDIFFERENCE WGEQ WGREATERP WLEQ
WLESSP WMINUS WNOT WOR WPLUS2 WRDSP WSHIFT WXOR))

(prog (remover)   %% Forget all about opencoded functions for the time being.
  (De propremover (x)
      (and (not (memq x openfnlist)) (remprop x 'openfn)))
  (mapobl 'propremover))
(remob 'Openfnlist)
```

14. Unix Shellscripts for Compilation, Assembly and Linkage

The newlasm script will compile, assemble, linkedit, and format a LISP program. Output is produced in several formats: intel hex format (filetype is i80), for the simulator (no filetype), and in hex dump (filetype is linkdump).

The syntax is:

```
    newlasm [-main] [-clean] [-indir <input dir]
                [-outdir <output dir>] filename>
```

(Note: the square brackets indicate optional items.)

The filename is the filename of the .sl file to be compiled.
The options are:

-main	To compile into a main program.
-clean	To expunge the .global symbol file prior to compilation.
-indir	To specify the directory where the global symbol (8051.sym) file should be copied from (note it is not copied back), and where extra object files should be found.
-outdir	To specify the directory where the listing, object, and other output should be written. The file 8051.sym is always written into . regardless of the value given to -outdir.

The defaults are:

-nomain	Don't generate initialization code.
-noclean	Don't initialize the 8051.sym file.
-indir	Search current directory during linkedit.
-outdir	Send .s, .o, .i, .lst, .linkmap, linkdump, and also executables to the current durectory.

Examples are:

```
newlasm -main -clean myprog
```
: Above compiles myprog.sl as a main program.

```
newlasm -clean -outdir rtna.dir rtna
mv 8051.sym rtna.dir
newlasm -main -indir rtna.dir -outdir bigprog.dir bigprog
```
The above first compiles the routines in rtna.sl and sends the

```
: output to directory rtna.dir. The 8051.sym file is manually
: copied into the directory to prevent errors. Finally a main
: program is compiled with the 8051.sym from rtna.dir, and it
: is linkedited with object files from rtna.dir. The results
: are written to bigprog.dir.
```

It is *critical* that no extraneous .o files be saved either in the working directory or the -indir directory. The only .o files should be from separate compilation, and in this case the 8051.sym file in -indir should correspond to all the .o files in -indir.

Because the 8051.sym file keeps the .o files synchronized it is important that it not be unintentionally overwritten. Therefore this file is not written to -outdir. Moveover, before a new 8051.sym file is written, the old one is renamed to 8051.oldsym for safekeeping.

Figure 14-1: UNIX Shellscript to Compile and Assemble a LISP Program

```
: Cross compiler.  Compile, assemble, and do options. 12/5/84 Lerner.
: Modified to provide complete functionality. 12/14/84 Christensen.
:
set -k
set symclean="/usr1/lerner/8051/symclean"
set compiler="/usr1/lerner/8051/x8051-cross"
: ::::::::::::::::::::::::::::::::::::::::::::::::::::::
: Process the -main and the -clean options.            :
: -main makes it a main file                           :
: -clean expunges the symbol table before compilation. :
:  Default is -noclean                                 :
: -indir the directory where of the symbol table       :
:       and .o files                                   :
: -outdir the directory to place the new symbol table  :
: ::::::::::::::::::::::::::::::::::::::::::::::::::::::

: nomain and clean are defaults

for param in $*
 do
   { case $param in
          -main    )  set optiona="on main;";shift  ;;
          -clean   )  set optionb="(load $symclean) (symclean)";shift  ;;
          -indir   )  shift;set indir=$1;shift;;
          -outdir  )  shift;set outdir=$1;shift;;
          default )
      esac }
done

echo indir is ${indir='.'}
echo outdir is ${outdir='.'}

:
: :::::::::::::::::::::::
: Invoke cross compiler :
: :::::::::::::::::::::::
:

mv 8051.sym 8051.oldsym
cp $indir/8051.sym .
echo Remember to put 8051.SYM into $outdir if appropriate.

$compiler << FLAGWORD
$optionb
(dskin "/usr1/lerner/8051/onelappass1.sl")
(rlisp)
in "/usr1/lerner/8051/mod.red"$
$optiona
asmout "$1";
in "$1.sl"$
asmend;
```

FLAGWORD

```
: :::::::::::::::::
: Assemble results :
: :::::::::::::::::
:
asm $1
if test . != $outdir
   then
       mv $1.i $outdir
       mv $1.o $outdir
       mv $1.s $outdir
       mv $1.lst $outdir
fi
:
:
if test ${optiona+set}
   then asm d$1
        if test . != $outdir
           then
               mv d$1.o $outdir
               mv d$1.s $outdir
               mv d$1.lst $outdir
        fi
   else
               mv d$1.s $outdir
fi
:
: :::::::::::::::::::::::::
: Optionally linkedit and :
: complete processing.    :
: :::::::::::::::::::::::::
:
echo outdir is $outdir
echo indir is $indir
:
:
if test ${optiona+set}
   then linkup $1 $indir $outdir
fi
```

UNIX Shcellscript to Compile and Assemble a LISP Program

Figure 14-2: UNIX Shellscript to Linkedit Cross Compiler Output

```
: Linkedit and further process an assembly.  12/5/84 Lerner.
:
: It is important that kernel.o be the first file in overlay 0.
:
: STACKTOPWORD was changed from 0x42 to 0x4a on Jan 2, 1985.
:
: Jan 2, 1985.
: The definition of REG3ITEM to REG8ITEM  was changed.
: REG3ITEM from 0x24 --> 0x2c
: REG4ITEM from 0x27 --> 0x2f  and so on.
:
link -o $3/$1 -i << FLAGWORD > $3/$1.linkmap
.option +maxdc
.pl 55
.start  STARTCODE
.define *ONCHIP         0x25 0xff
.define *CODEROM        0x26
.define *CODE           0x4000  0x7000
.define *ONCHIPBITS     0       0xff
.define bottom 0
.define top 4096
.define markbit 0x80
.define STACKPTR 0x4d
.define STACKTOP 0x72ff.
.define stackptr 0x4d
.define stacktop 0x72ff
.define extrareg 0x7300
.define extraregtop 0x73ff
.define NEXTFREE 0x7400
.define STARTFREE 0x7400
.define ENDFREE 0x7A00
.define CSBUFFER 0x7B00
.define REG1ITEM 0x2
.define REG2ITEM 0x5
.define REG3ITEM 0x2c
.define REG4ITEM 0x2f
.define REG5ITEM 0x33
.define REG6ITEM 0x36
.define REG7ITEM 0x39
.define REG8ITEM 0x3c
.define STACKTOPWORD 0x4a
.overlay 0
/usr1/lerner/ppsl-utils/kernel.o
$3/$1.o
$3/d$1.o
.search $2
.search /usr1/lerner/ppsl-utils
.end
FLAGWORD
:
: Create result files
```

```
:
echo Creating $3/$1.linkdump
linkdump $3/$1 > $3/$1.linkdump
echo Creating $3/$1.180
amf $3/$1 > $3/$1.180
echo Creating $3/$1.lispsym
:
: Create file of lisp symbols
:
awk '/\(PUT \(QUOTE [a-zA-Z0-9]+\) \(QUOTE IDNUMBER\) \(QUOTE [0-9]+\)\)/ \
  { print substr($7,1,length($7)-2), substr($3,1,length($3)-1) }' \
  8051.sym > $3/$1.lispsym
:
egrep 'STARTCODE|DEBUG' $3/$1.linkmap
```

15. Sample Programs, Compilation, and Execution

The following shows compilation, linkage and execution of several programs It demonstates the primitive *mapcar* with the necessary *fastapply* and *cons* routines, as well as *reverse*, *list*, and a recursive *factorial* function[7].

The shellscript called "lasm" performs the entire process The steps of lasm are shown below in boldface, though in practice the user can ignore this. *Comments* are shown in italics, and commands in **boldface**.

The following has been compiled with the cross compiler and correctly executed. The complete source includes additional support routines.

```
(DE FUNCD(X) (TIMES2 X X))
(DE FUNCE(X) (PLUS2 X X))
(DE FACTORIAL(X)
    (COND ((EQ X 1) 1)
          (T (TIMES X (FACTORIAL (SUB1 X))))))

(DE DADO_MAIN()
    (PROG (W Y)
          (MAKEFREELIST)
          (SETQ Y '(FUNCD FUNCE FACTORIAL))
          (SETQ Y (FOREACH Z IN Y COLLECT  (MAPCAR '(1 2 3 4 5) Z)))
          (DEBUG Y)  %Probe to see result
          (SETQ Y (CONS (NCONC Y '(A B C D)) '(E F G H)))
          (DEBUG Y)
          (SETQ Y (REVERSE Y))
          (DEBUG Y)
          (SETQ W 1)
     MORE (SETQ Y (CONS (FACTORIAL W) Y))
          (SETQ W (ADD1 W))
          (AND (NEQ W 6) (GO MORE))
          (DEBUG Y)
          (SETQ Y (REVERSE (CONS Y Y)))
          (DEBUG Y)
          (RETURN Y)))
```

Figure 15-1: Sample Program

[7] The factorial is executed recursively.

Figure 15-2: Sample Program Execution

```
% lasm checker >& checker.log          Execute lisp cross compiler
          Find entry point (startcode) and test probe (debug)
% grep 'STARTCODE|DEBUG' checker.linkmap
                                       0160            STARTCODE
                                       0689            DEBUG
% ../sym51                             Execute the 8051 instruction simulator
filename>checker                       Name of linked object code
Loading memory  100 for   5b             System prints memory utilization
Loading memory  15d for  400
Loading memory  55d for  400
Loading memory  95d for  400
Loading memory  d5d for  25d
Loading memory   1e for   63
sim51> pc                              Give initial program counter ...
Value of pc: 160                          as 160.
sim51> b                               Give a breakpoint ...
break>689                                 to see what it does.
sim51> go                              Off and running!
breakpoint at  689
24355 instructions executed
sim51> x                               Examine registers
Registers: 02 04 09 24 24 09 24 6c
pc: 0689 (1673) acc: 24 (36) dptr: 1ff6 psw: 40
Examine int_ram or sfr_ram?
sim51> z                               Print result
Print S-expr (2 cons cells) from>      r1
((0x0001 0x0004 0x0009 0x0010 0x0019 )
 (0x0002 0x0004 0x0006 0x0008 0x000a )
 (0x0001 0x0002 0x0006 0x0018 0x0078 ) )
sim51> go
breakpoint at  689
711 instructions executed
sim51> z
Print S-expr (2 cons cells) from>      r1
(((0x0001 0x0004 0x0009 0x0010 0x0019 )
  (0x0002 0x0004 0x0006 0x0008 0x000a )
  (0x0001 0x0002 0x0006 0x0018 0x0078 ) A B C D ) E F G H )
sim51> go
breakpoint at  689
3361 instructions executed
sim51> z
Print S-expr (2 cons cells) from>      r1
(H G F E ((0x0001 0x0004 0x0009 0x0010 0x0019 )
         (0x0002 0x0004 0x0006 0x0008 0x000a )
         (0x0001 0x0002 0x0006 0x0018 0x0078 ) A B C D ) )
sim51> x
Registers: 02 04 09 24 ffffff90 1e 00 ffffff80
pc: 0689 (1673) acc: 90 (-112) dptr: 1ff6 psw: 40
Examine int_ram or sfr_ram?
sim51> go
breakpoint at  689
```

```
4176 instructions executed
sim51>  z
Print S-expr (2 cons cells) from>       r1
(0x0078 0x0018 0x0006 0x0002 0x0001 H G F E
  ((0x0001 0x0004 0x0009 0x0010 0x0019 )
   (0x0002 0x0004 0x0006 0x0008 0x000a )
   (0x0001 0x0002 0x0006 0x0018 0x0078 ) A B C D ) )
sim51>  go
breakpoint at  689
7620 instructions executed
sim51>  z
Print S-expr (2 cons cells) from>       r1

(((0x0001 0x0004 0x0009 0x0010 0x0019 )
  (0x0002 0x0004 0x0006 0x0008 0x000a )
  (0x0001 0x0002 0x0006 0x0018 0x0078 )
  A B C D) E F G H 0x0001 0x0002 0x0006 0x0018 0x0078
 (0x0078 0x0018 0x0006 0x0002 0x0001 H G F E
         ((0x0001 0x0004 0x0009 0x0010 0x0019 )
          (0x0002 0x0004 0x0006 0x0008 0x000a )
          (0x0001 0x0002 0x0006 0x0018 0x0078 )
          A B C D)))

sim51>  n
Number instructions to execute before break? 1000
sim51>  go
4096 instructions executed
sim51>  s
0180: 80 fe SJMP code add       sim51-step>     Execution complete.
0180: 80 fe SJMP code add       sim51-step>
0180: 80 fe SJMP code add       sim51-step>
0180: 80 fe SJMP code add       sim51-step>     q
sim51>  q
```

Sample Program Execution

Figure 15-3: Complete Source Code of Sample

```
(ON SYSLISP PLAP PCMAC)
(WDECLARE EXTERNAL WVAR (NEXTFREE NIL NIL)
                        (STARTFREE NIL NIL)
                        (ENDFREE NIL NIL))

%% Initially link memory as follows:
%%
%% ============================================================
%% | <========== car ==========> | <========== cdr ==========> |
%% | tag1 | byte1high | byte1low | tag2 | byte2high | byte2low |
%% | <=========== 0 ===========> | <=== address next pair ===> |
%% ============================================================
%%
%%
%% Later, use a first fit blocking compactor as in Madnick.
%%

(COMPILETIME (SETQ SAVEDCOMPFN (REMPROP (QUOTE CONS) (QUOTE COMPFN))))
(DE NCONS (U) (CONS U NIL))
(DE XCONS (U V) (CONS V U))
(COMPILETIME (PUT (QUOTE CONS) (QUOTE COMPFN) SAVEDCOMPFN))
(DE LIST5 (U V W X Y) (CONS U (LIST4 V W X Y)))
(DE LIST4 (U V W X) (CONS U (LIST3 V W X)))
(DE LIST3 (U V W) (CONS U (LIST2 V W)))
(DE LIST2 (U V) (CONS U (NCONS V)))

(DE ADD1 (U) (WPLUS2 U 1))
(DE PLUS2 (U V) (WPLUS2 U V))

%% Markbit presumed defined at #0x80 (ie,  bit 8)
%% mark marks an item

(lap '((!*entry markitem expr 1)  %%Machine specific mark routine
       (mov (onchip accumulator a) (onchip register r2))
       ("ORL A, #markbit;  turn on the mark bit.")
       (mov (onchip register r2) (onchip accumulator a))
       (!*exit 0)))

%% Unmark unmarks an item

(lap '((!*entry unmark expr 1)  %%Machine specific unmark routine
       (mov (onchip accumulator a) (onchip register r2))
       ("ANL A, # lo ~ markbit;  turn off the mark bit.")
       (mov (onchip register r2) (onchip accumulator a))
       (!*exit 0)))

%% Markp checks if an item is marked

(lap '((!*entry markp expr 1)
       (mov (onchip accumulator a) (onchip register r2))
       ("ANL A, #markbit;  turn off everything except the mark bit.")
```

```
        ("jz markp1")
        (!*move (quote t) (reg 1))
        ("sjmp markp2")
        ("markp1: ")
        (!*move (quote nil) (reg 1))
        ("markp2: ")
        (!*exit 0)))


%% consitemp returns t if p is in conspace nil otherwise

(De consitemp (p)
  (and (leq p (loc endfree))
       (geq p (loc startfree))))

%% gc marks the items of p that are in conspace.
%% Items outside conspace get marked temporarily but are left
%% unmarked when gc finishes.

%% (getmem p) returns the car of p
%% (getmem (plus p 3)) returns the cdr of p


(DE GC (P)
    (PROG (TEMP BACKPT)
          (setq backpt nil)
 CARWORD
        (setq p (MarkItem P))
        (COND
         ((OR (ATOM P) (MARKP (getmem P)))
               (GO CDRWORD))
         (T (SETQ TEMP (getmem P))
            (SETF (getmem P) BACKPT)
            (SETQ BACKPT P)
            (SETQ P TEMP)
            (GO CARWORD)))

  CDRWORD
        (COND
         ((OR (atom p) (ATOM (getmem (plus P 3)))
              (MARKP (getmem (getmem  (plus P 3)))))
                (cond ((and (not (atom p))
                            (consitemp (plus p 3)))
                       (setf (getmem (plus p 3)) (MarkItem (getmem (plus P 3))))))
                (GO BACKWORD))
         (T (SETQ TEMP (getmem  (plus P 3)))
            (SETf (getmem (plus P 3)) BACKPT)
            (setf (getmem (plus p 3)) (MarkItem (getmem (plus P 3))))
            (SETQ BACKPT P)
            (SETQ P TEMP)
            (GO CARWORD)))

BACKWORD
        (COND ((NULL (unmark BACKPT)))
              (T (SETQ TEMP P)
                 (SETQ P BACKPT)
                 (COND ((MARKP (getmem (plus P 3)))
                        (SETQ BACKPT (getmem (plus P 3)))
                        (SETf (getmem (plus P 3)) TEMP)
                          (cond ((not (consitemp (plus p 3)))
                                 (setf (getmem (plus p 3))
                                       (unmark (getmem (plus p 3))))))
                        (GO BACKWORD))
                       (T (SETQ BACKPT (getmem P))
                          (SETf (getmem P) TEMP)
                          (cond ((not (consitemp p))
                                 (setf (getmem p) (unmark (getmem p)))))
                          (GO CDRWORD))))))


(de markregister(r)
    (cond ((not (numberp (getmem r)))
                            (GC (getmem r)))))


%% Markallocated calls gc to mark all the items that we have
%% to save.

(DE MARKALLOCATED ()
    (prog (stw)
%% .mark registers
     (markregister reg1item)
     (markregister reg2item)
     (markregister reg3item)
     (markregister reg4item)
     (markregister reg5item)
     (markregister reg6item)
     (markregister reg7item)
     (markregister reg8item)
     (setq stw stacktopword)
%% mark stuff on data stack
     (FOR (FROM MEMLOC stw (difference (loc stacktop) 3) 3)
              (DO (PROGN
                     (cond ((not (numberp (getmem memloc)))
                            (GC (getmem MEMLOC)))))))))

%% Makefreelist makes a new freelist of the unmarked items within
%% conspace.

(DE MAKEFREELIST ()
    (PROG (MEMLOC OLDMEMLOC)
          (SETF STARTFREE (PLUS (LOC STARTFREE) 6))
          (SETF ENDFREE (DIFFERENCE (LOC ENDFREE) 12))
          (SETQ MEMLOC STARTFREE)
%% Find first free item
          (WHILE (AND (MARKP (getmem MEMLOC)) (LESSP MEMLOC ENDFREE))
                 (setf (getmem memloc) (UNMARK (getmem memloc)))
                 (SETQ MEMLOC (PLUS MEMLOC 6)))
```

```
%% If there is one then start free list at that point
      (COND ((LESSP MEMLOC ENDFREE)
             (progn
             (SETQ NEXTFREE MEMLOC)
             (SETF (GETMEM MEMLOC) 0)
             (setf (getmem (plus memloc 3)) 0)
             (SETQ OLDMEMLOC MEMLOC)
             (FOR (FROM MEMLOC (PLUS NEXTFREE 6) ENDFREE 6)
                  (DO (COND ((NOT (MARKP (getmem MEMLOC)))
                             (PROGN (SETF (GETMEM (PLUS OLDMEMLOC 3)) MEMLOC)
                                    (SETF (GETMEM MEMLOC) 0)
                                    (setf (getmem (plus memloc 3)) 0)
                                    (SETQ OLDMEMLOC MEMLOC)))
                        (T (progn (setf (getmem memloc)
                                        (unmark (getmem MEMLOC)))
                                  (setf (getmem (plus memloc 3))
                                        (unmark (getmem (plus memloc 3))))))))))
            (t (debug '(garbage collection failed no free memory))))))

%% garbagecollect is the function that gets called when garbage
%% collection has to be done.

(DE GARBAGECOLLECT ()
    (MARKALLOCATED)
    (MAKEFREELIST))



(DE CONS (U V)
  (PROG (NF RV)
        (COND ((EQUAL NEXTFREE 0) (GARBAGECOLLECT)))
        (SETQ  RV NEXTFREE)
        (SETQ  NF NEXTFREE)
        (PUTFIELD NEXTFREE 8 16
                  %% Update the available list (AVL)
                  (GETMEM (FIELD (PLUS NEXTFREE 3) 8 16)))
        (SETF (GETMEM NF) U)                                  %% Store car
        (SETQ NF (PLUS NF 3))                                 %% Point to cdr
        (SETF (GETMEM NF) V)                                  %% Store cdr
        (RETURN (MKPAIR RV))))                                %% Return pair


(DE MAPCAR (L FN)   %% TAIL RECURSIVE MAPCAR DEFINITION. USES FASTAPPLY
    (COND ((NOT (PAIRP L)) NIL)
          (T (CONS (APPLY FN (LIST (CAR L))) (MAPCAR (CDR L) FN)))))

(LAP '((!*ENTRY FASTAPPLY EXPR 2)
       (!*PUSH (REG 1))                   %% SAVE POINTER TO ITEM.
       (!*MOVE (REG 2) (REG 1))           %% GET ADDRESS ...
       (!*LINK GETFNCENTRYPOINT EXPR 1)   %%    ... OF ROUTINE ...
       (!*MOVE (REG 1) (REG 2))           %%      ... REMEMBER IT.
       (!*POP (REG 1))                    %%  RESTORE POINTER ...
       (!*MOVE (REG 2) (REG DPTR))        %%    ... PREPARE TO ...
       (!*CLR (ONCHIP ACCUMULATOR A))     %%      ... JUMP TO ITEM.
```

```
        ('JMP @A+DPTR')))                        DO IT.

(LAP '((!*ENTRY GETFNCENTRYPOINT EXPR 1)
        (!*MOVE (WCONST 3) (REG 2))
        (!*LINK TIMES2 EXPR 2)
        (!*MOVE (EXTERNALLOC '#HI SYMFNC') (ONCHIP RAM DPH))
        (!*MOVE (EXTERNALLOC '#LO SYMFNC') (ONCHIP RAM DPL))
        (!*WPLUS2 (REG 1) (REG DPTR))
        (!*EXIT 0)))

(OFF SYSLISP)

(DE DEBUG(X)X)  %% Named entry point for debugging.

(DE REVERSE (U)
    (PROG (V)
          (WHILE (PAIRP U)
                 (PROGN (SETQ V
                              (CONS (CAR U) V))
                        (SETQ U (CDR U))))
          (RETURN V)))

(DE GARBAGECOLLECT() (PROG (X) ZZ (SETQ X X) (GO ZZ)))

(DE FUNCD(X) (TIMES2 X X))
(DE FUNCE(X) (PLUS2 X X))

(DE FACTORIAL(X)
    (COND ((EQ X 1) 1)
          (T (TIMES X (FACTORIAL (SUB1 X))))))

(DE APPEND (U V)
    (COND ((NOT (PAIRP U)) V)
          (T (PROG (U1 U2)  (SETQ U1 (SETQ U2 (CONS (CAR U) NIL)))
                   (SETQ U (CDR U))
                   (WHILE (PAIRP U)
                          (PROGN (RPLACD U2 (CONS (CAR U) NIL))
                                 (SETQ U (CDR U)) (SETQ U2 (CDR U2))))
                   (RPLACD U2 V)
                   (RETURN U1)))))

(DE NCONC (U V)
    (PROG (W)
          (COND ((NOT (PAIRP U)) (RETURN V)))
          (SETQ W U)
          (WHILE (PAIRP (CDR W))
                 (SETQ W (CDR W))) (RPLACD W V)
                                   (RETURN U)))

(DE DADO_MAIN()
    (PROG (W Y)
          (MAKEFREELIST)
          (SETQ Y '(FUNCD FUNCE FACTORIAL))
          (SETQ Y
        (FOREACH Z IN Y COLLECT
                 (MAPCAR '(1 2 3 4 5) Z)))
       (DEBUG Y)
       (SETQ Y (CONS (NCONC Y '(A B C D)) '(E F G H)))
       (DEBUG Y)
       (SETQ Y (REVERSE Y))
       (DEBUG Y)
       (SETQ W 1)
  MORE (SETQ Y (CONS (FACTORIAL W) Y))
       (SETQ W (ADD1 W))
       (AND (NEQ W 6) (GO MORE))
       (DEBUG Y)
       (SETQ Y (REVERSE (CONS Y Y)))
       (DEBUG Y)
       (RETURN Y)))
```

Figure 15-4: Log file of program compilation, assembly and linkage

```
% cat checker.log
||PSL, version 29-Oct-84
1 lisp> 1 lisp> PSL Rlisp
2 rlisp>> NIL
3 rlisp>> ASMOUT: IN files; or type in expressions
When all done execute ASMEND;
NIL
4 rlisp>> (!*ENTRY MAKEFREELIST EXPR 0)
(!*ALLOC 0)
(!*LOC (REG 1) (WVAR STARTFREE))
(!*WPLUS2 (REG 1) (WCONST 6))
(!*PUTFIELD (REG 1) (WVAR STARTFREE) (WCONST 0) (WCONST 24))
(!*LOC (REG 2) (WVAR ENDFREE))
(!*WPLUS2 (REG 2) (WCONST -12))
(!*PUTFIELD (REG 2) (WVAR ENDFREE) (WCONST 0) (WCONST 24))
(!*MOVE (WVAR STARTFREE) (REG 3))
(!*LBL (LABEL G0005))
(!*JUMPWLEQ (LABEL G0006) (REG 3) (WVAR ENDFREE))
(!*MOVE (WCONST 0) (REG 1))
(!*EXIT 0)
(!*LBL (LABEL G0006))
(!*MOVE (WCONST 0) (MEMORY (REG 3) (WCONST 0)))
(!*MOVE (REG 3) (REG 1))
(!*WPLUS2 (REG 1) (WCONST 6))
(!*MOVE (REG 1) (MEMORY (REG 3) (WCONST 3)))
(!*WPLUS2 (REG 3) (WCONST 6))
(!*JUMP (LABEL G0005))

(!*ENTRY NCONS EXPR 1)
(!*ALLOC 0)
(!*MOVE (QUOTE NIL) (REG 2))
(!*LINKE 0 CONS EXPR 2)

(!*ENTRY XCONS EXPR 2)
(!*ALLOC 0)
(!*MOVE (REG 2) (REG 3))
(!*MOVE (REG 1) (REG 2))
(!*MOVE (REG 3) (REG 1))
(!*LINKE 0 CONS EXPR 2)

(!*ENTRY LIST5 EXPR 5)
(!*ALLOC 4)
(!*MOVE (REG 1) (FRAME 1))
(!*MOVE (REG 2) (FRAME 2))
(!*MOVE (REG 3) (FRAME 3))
(!*MOVE (REG 4) (FRAME 4))
(!*MOVE (REG 5) (REG 4))
(!*MOVE (FRAME 4) (REG 3))
(!*MOVE (FRAME 3) (REG 2))
(!*MOVE (FRAME 2) (REG 1))
```

```
(!*LINK LIST4 EXPR 4)
(!*MOVE (FRAME 1) (REG 2))
(!*LINKE 4 XCONS EXPR 2)

(!*ENTRY LIST4 EXPR 4)
(!*ALLOC 3)
(!*MOVE (REG 1) (FRAME 1))
(!*MOVE (REG 2) (FRAME 2))
(!*MOVE (REG 3) (FRAME 3))
(!*MOVE (REG 4) (REG 3))
(!*MOVE (FRAME 3) (REG 2))
(!*MOVE (FRAME 2) (REG 1))
(!*LINK LIST3 EXPR 3)
(!*MOVE (FRAME 1) (REG 2))
(!*LINKE 3 XCONS EXPR 2)

(!*ENTRY LIST3 EXPR 3)
(!*PUSH (REG 2))
(!*PUSH (REG 1))
(!*MOVE (REG 3) (REG 2))
(!*MOVE (FRAME 2) (REG 1))
(!*LINK LIST2 EXPR 2)
(!*MOVE (FRAME 1) (REG 2))
(!*LINKE 2 XCONS EXPR 2)

(!*ENTRY LIST2 EXPR 2)
(!*PUSH (REG 1))
(!*MOVE (REG 2) (REG 1))
(!*LINK NCONS EXPR 1)
(!*MOVE (FRAME 1) (REG 2))
(!*LINKE 1 XCONS EXPR 2)

(!*ENTRY ADD1 EXPR 1)
(!*ALLOC 0)
(!*WPLUS2 (REG 1) (WCONST 1))
(!*EXIT 0)

(!*ENTRY PLUS2 EXPR 2)
(!*ALLOC 0)
(!*WPLUS2 (REG 1) (REG 2))
(!*EXIT 0)

(!*ENTRY CONS EXPR 2)
(!*ALLOC 4)
(!*MOVE (REG 1) (FRAME 1))
(!*MOVE (REG 2) (FRAME 2))
(!*JUMPNOTEQ (LABEL G0005) (WCONST 0) (WVAR NEXTFREE))
(!*LINK GARBAGECOLLECT EXPR 0)
(!*LBL (LABEL G0005))
(!*MOVE (WVAR NEXTFREE) (FRAME 4))
(!*MOVE (WVAR NEXTFREE) (FRAME 3))
(!*MOVE (WVAR NEXTFREE) (REG 2))
(!*WPLUS2 (REG 2) (WCONST 3))
(!*FIELD (REG 2) (REG 2) (WCONST 8) (WCONST 16))
```

```
(!*PUTFIELD (MEMORY (REG 2) (WCONST 0)) (WVAR NEXTFREE) (WCONST 8) (WCONST
16))
(!*MOVE (FRAME 1) (MEMORY (FRAME 3) (WCONST 0)))
(!*WPLUS2 (FRAME 3) (WCONST 3))
(!*MOVE (FRAME 2) (MEMORY (FRAME 3) (WCONST 0)))
(!*MOVE (FRAME 4) (REG 1))
(!*MKITEM (REG 1) (WCONST 9))
(!*EXIT 4)

(!*ENTRY MAPCAR EXPR 2)
(!*ALLOC 3)
(!*MOVE (REG 1) (FRAME 1))
(!*MOVE (REG 2) (FRAME 2))
(!*JUMPTYPE (LABEL G0005) (REG 1) PAIR)
(!*MOVE (QUOTE NIL) (REG 1))
(!*JUMP (LABEL G0001))
(!*LBL (LABEL G0005))
(!*MOVE (CAR (REG 1)) (REG 1))
(!*MOVE (REG 2) (REG T1))
(!*LINK FASTAPPLY EXPR 1)
(!*MOVE (REG 1) (FRAME 3))
(!*MOVE (FRAME 2) (REG 2))
(!*MOVE (CDR (FRAME 1)) (REG 1))
(!*LINK MAPCAR EXPR 2)
(!*MOVE (FRAME 3) (REG 2))
(!*LINKE 3 XCONS EXPR 2)
(!*LBL (LABEL G0001))
(!*EXIT 3)

(!*ENTRY DEBUG EXPR 1)
(!*ALLOC 0)
(!*EXIT 0)

(!*ENTRY REVERSE EXPR 1)
(!*PUSH (QUOTE NIL))
(!*PUSH (REG 1))
(!*LBL (LABEL G0006))
(!*JUMPNOTTYPE (LABEL G0005) (FRAME 1) PAIR)
(!*MOVE (FRAME 2) (REG 2))
(!*MOVE (CAR (FRAME 1)) (REG 1))
(!*LINK CONS EXPR 2)
(!*MOVE (REG 1) (FRAME 2))
(!*MOVE (CDR (FRAME 1)) (REG 2))
(!*MOVE (REG 2) (FRAME 1))
(!*JUMP (LABEL G0006))
(!*LBL (LABEL G0005))
(!*MOVE (FRAME 2) (REG 1))
(!*EXIT 2)

(!*ENTRY GARBAGECOLLECT EXPR 0)
(!*ALLOC 0)
(!*MOVE (QUOTE NIL) (REG 1))
(!*LBL (LABEL G0005))
(!*MOVE (REG 1) (REG 1))
```

```
(!*JUMP (LABEL G0005))

(!*ENTRY FUNCD EXPR 1)
(!*ALLOC 0)
(!*MOVE (REG 1) (REG 2))
(!*LINKE 0 TIMES2 EXPR 2)

(!*ENTRY FUNCE EXPR 1)
(!*ALLOC 0)
(!*MOVE (REG 1) (REG 2))
(!*LINKE 0 PLUS2 EXPR 2)

(!*ENTRY FACTORIAL EXPR 1)
(!*PUSH (REG 1))
(!*JUMPNOTEQ (LABEL G0005) (REG 1) (QUOTE 1))
(!*MOVE (QUOTE 1) (REG 1))
(!*JUMP (LABEL G0001))
(!*LBL (LABEL G0005))
(!*LINK SUB1 EXPR 1)
(!*LINK FACTORIAL EXPR 1)
(!*MOVE (REG 1) (REG 2))
(!*MOVE (FRAME 1) (REG 1))
(!*LINKE 1 TIMES2 EXPR 2)
(!*LBL (LABEL G0001))
(!*EXIT 1)

(!*ENTRY APPEND EXPR 2)
(!*ALLOC 4)
(!*MOVE (REG 1) (FRAME 1))
(!*MOVE (REG 2) (FRAME 2))
(!*JUMPTYPE (LABEL G0005) (REG 1) PAIR)
(!*MOVE (REG 2) (REG 1))
(!*JUMP (LABEL G0001))
(!*LBL (LABEL G0005))
(!*MOVE (QUOTE NIL) (FRAME 3))
(!*MOVE (QUOTE NIL) (FRAME 4))
(!*MOVE (CAR (REG 1)) (REG 1))
(!*LINK NCONS EXPR 1)
(!*MOVE (REG 1) (REG 3))
(!*MOVE (REG 3) (FRAME 4))
(!*MOVE (REG 3) (FRAME 3))
(!*MOVE (CDR (FRAME 1)) (REG 2))
(!*MOVE (REG 2) (FRAME 1))
(!*LBL (LABEL G0010))
(!*JUMPNOTTYPE (LABEL G0009) (FRAME 1) PAIR)
(!*MOVE (CAR (FRAME 1)) (REG 1))
(!*LINK NCONS EXPR 1)
(!*MOVE (REG 1) (CDR (FRAME 4)))
(!*MOVE (CDR (FRAME 1)) (REG 2))
(!*MOVE (REG 2) (FRAME 1))
(!*MOVE (CDR (FRAME 4)) (REG 3))
(!*MOVE (REG 3) (FRAME 4))
(!*JUMP (LABEL G0010))
(!*LBL (LABEL G0009))
```

```
(!*MOVE (FRAME 2) (CDR (FRAME 4)))
(!*MOVE (FRAME 3) (REG 1))
(!*LBL (LABEL G0001))
(!*EXIT 4)

(!*ENTRY NCONC EXPR 2)
(!*ALLOC 0)
(!*MOVE (REG 1) (REG 5))
(!*MOVE (REG 2) (REG 4))
(!*MOVE (QUOTE NIL) (REG 3))
(!*JUMPTYPE (LABEL G0005) (REG 1) PAIR)
(!*MOVE (REG 2) (REG 1))
(!*EXIT 0)
(!*LBL (LABEL G0005))
(!*MOVE (REG 1) (REG 3))
(!*LBL (LABEL G0009))
(!*JUMPNOTTYPE (LABEL G0008) (CDR (REG 3)) PAIR)
(!*MOVE (CDR (REG 3)) (REG 1))
(!*MOVE (REG 1) (REG 3))
(!*JUMP (LABEL G0009))
(!*LBL (LABEL G0008))
(!*MOVE (REG 4) (CDR (REG 3)))
(!*MOVE (REG 5) (REG 1))
(!*EXIT 0)

(!*ENTRY DADO_MAIN EXPR 0)
(!*ALLOC 5)
(!*MOVE (QUOTE NIL) (FRAME 1))
(!*LINK CONSINIT EXPR 0)
(!*MOVE (QUOTE (FUNCD FUNCE FACTORIAL)) (FRAME 2))
(!*MOVE (QUOTE NIL) (FRAME 4))
(!*MOVE (QUOTE NIL) (FRAME 5))
(!*MOVE (FRAME 2) (FRAME 3))
(!*JUMPTYPE (LABEL G0011) (FRAME 3) PAIR)
(!*MOVE (QUOTE NIL) (REG 1))
(!*JUMP (LABEL G0009))
(!*LBL (LABEL G0011))
(!*MOVE (CAR (FRAME 3)) (REG 1))
(!*MOVE (REG 1) (REG 2))
(!*MOVE (QUOTE (1 2 3 4 5)) (REG 1))
(!*LINK MAPCAR EXPR 2)
(!*LINK NCONS EXPR 1)
(!*MOVE (REG 1) (FRAME 5))
(!*MOVE (REG 1) (FRAME 4))
(!*LBL (LABEL G0010))
(!*MOVE (CDR (FRAME 3)) (REG 1))
(!*MOVE (REG 1) (FRAME 3))
(!*JUMPTYPE (LABEL G0022) (REG 1) PAIR)
(!*MOVE (FRAME 4) (REG 1))
(!*JUMP (LABEL G0009))
(!*LBL (LABEL G0022))
(!*MOVE (CAR (REG 1)) (REG 1))
(!*MOVE (REG 1) (REG 2))
(!*MOVE (QUOTE (1 2 3 4 5)) (REG 1))
```

```
(!*LINK MAPCAR EXPR 2)
(!*LINK NCONS EXPR 1)
(!*MOVE (REG 1) (CDR (FRAME 5)))
(!*MOVE (CDR (FRAME 5)) (REG 2))
(!*MOVE (REG 2) (FRAME 5))
(!*JUMP (LABEL G0010))
(!*LBL (LABEL G0009))
(!*MOVE (REG 1) (FRAME 2))
(!*LINK DEBUG EXPR 1)
(!*MOVE (QUOTE (A B C D)) (REG 2))
(!*MOVE (FRAME 2) (REG 1))
(!*LINK NCONC EXPR 2)
(!*MOVE (QUOTE (E F G H)) (REG 2))
(!*LINK CONS EXPR 2)
(!*MOVE (REG 1) (FRAME 2))
(!*LINK DEBUG EXPR 1)
(!*MOVE (FRAME 2) (REG 1))
(!*LINK REVERSE EXPR 1)
(!*MOVE (REG 1) (FRAME 2))
(!*LINK DEBUG EXPR 1)
(!*MOVE (QUOTE 1) (FRAME 1))
(!*LBL (LABEL G0008))
(!*MOVE (FRAME 1) (REG 1))
(!*LINK FACTORIAL EXPR 1)
(!*MOVE (FRAME 2) (REG 2))
(!*LINK CONS EXPR 2)
(!*MOVE (REG 1) (FRAME 2))
(!*MOVE (FRAME 1) (REG 1))
(!*LINK ADD1 EXPR 1)
(!*MOVE (REG 1) (FRAME 1))
(!*JUMPNOTEQ (LABEL G0035) (REG 1) (QUOTE 6))
(!*MOVE (QUOTE NIL) (REG 1))
(!*JUMP (LABEL G0036))
(!*LBL (LABEL G0035))
(!*MOVE (QUOTE T) (REG 1))
(!*LBL (LABEL G0036))
(!*JUMPNOTEQ (LABEL G0008) (REG 1) (QUOTE NIL))
(!*MOVE (FRAME 2) (REG 1))
(!*LINK DEBUG EXPR 1)
(!*MOVE (FRAME 2) (REG 2))
(!*MOVE (REG 2) (REG 1))
(!*LINK CONS EXPR 2)
(!*LINK REVERSE EXPR 1)
(!*MOVE (REG 1) (FRAME 2))
(!*LINK DEBUG EXPR 1)
(!*MOVE (FRAME 2) (REG 1))
(!*EXIT 5)
NIL
5 rlisp>> (!*ENTRY INITCODE EXPR 0)
(!*ALLOC 0)
(!*MOVE (QUOTE NIL) (REG 1))
(!*EXIT 0)

***** 'SYMVAL' multiply defined
```

```
***** 'SYMPRP' multiply defined
***** 'SYMNAM' multiply defined
***** 'NEXTSYMBOL' multiply defined
NIL
6 rlisp>> Exiting rlisp
NIL
7 lisp> Exiting lisp
1 lines with warnings in this assembly
.pl 66
.start STARTCODE
.define *DATA 1e ff
.define *CODE 100 ffff
.define bottom 0            Low memory to copy when ICE is used
.define top 4096            High memory to copy when ICE used
.define markbit 0x10        Garbage collection bit
.define stackptr 44         Initial value of hardware stack
.define stacktop 2000       Initial value of data (software) stack
.define extrareg 2001       Start of extraregister area
.define extraregtop 2100    End of extraregister area
.define NEXTFREE 2400       Addr of pointer to next free cons cell
.define STARTFREE 2400      Start of cons cell space
.define ENDFREE 2700        End of cons cell space
.overlay 1
kernel.o                    Definitions of kernel entrypoints
.search /usr1/lerner/ppsl-utils
4 Library entries to search
.library /usr1/lerner/ppsl-utils/stacks.o  Efficient stack routines
.library /usr1/lerner/ppsl-utils/sub1.o    Arithmetic
.library /usr1/lerner/ppsl-utils/times2.o  Arithmetic
.library /usr1/lerner/ppsl-utils/nkrn.o    Kernel entry points
.rm
.overlay 0
dchecker.o                  Data segment of object code
checker.o                   Code segment of object code
.search /usr1/lerner/ppsl-utils

4 Library entries to search
.library /usr1/lerner/ppsl-utils/stacks.o
.library /usr1/lerner/ppsl-utils/sub1.o
.library /usr1/lerner/ppsl-utils/times2.o
.library /usr1/lerner/ppsl-utils/nkrn.o
.rm
;library /usr1/lerner/ppsl-utils/stacks.o
;library /usr1/lerner/ppsl-utils/sub1.o
;library /usr1/lerner/ppsl-utils/times2.o
.library /usr1/lerner/ppsl-utils/nkrn.o
.rm
.end
                                        0160           STARTCODE
                                        0160           STARTCODE
                                        0689           DEBUG
                                        0689           DEBUG
% logout
```

16. Acknowledgements

Gerald Q. Maguire Jr. commented on an earlier version of this paper.

This work could not have been conceived, let alone completed, without the support of Salvatore Stolfo (Columbia University) and Gerald Q. Maguire Jr. (Columbia University). It is based on the Portable Standard Lisp (PSL) compiler developed at the University of Utah [Griss 81, Griss 82]. Bob Kessler (University of Utah) provided insights and code at critical points in the cross compiler development. Gerald Q. Maguire Jr. provided key insights into the compiler, and with Steve Taylor developed the idea of using LISP to program DADO. They specified the idea of maintaining the symbol table in the host processor of the machine. Michael van Biema made important design decisions and understood the thorniest parts of the code generation issues. Chris Maio is studying the question of parallel debugging, and has suggested the generation of certain code sequences to make automatic debugging possible. Among the technical staff, Douglas Gordin integrated the compiler with the DADO kernel, and improved the 8051 machine simulator. Jens Christensen developed the garbage collecton algorithm specified by Michael van Biema, and patiently tracked down obscure bugs. Philip Yuen worked on the kernel software, and developed techniques for the communication of LISP structures between processors.

A LISP Compiler for the DADO Parallel Computer

17. References

[van Biema 84]
||*PSL: A Parallel Lisp for the DADO Machine* by van Biema M., M. D. Lerner, G. Q. Maguire Jr. Columbia University Technical Report, February 1984.

[Griss 79]
A Portable Lisp Compiler by Martin L. Griss and Anthony C. Hearn, Department of Computer Science, University of Utah. Report #UUCS-79-113.

[Griss 81]
A Portable LISP compiler by Martin L. Griss and Anthony C. Hearn in "Software - Practice and Experience," June 1981, pages 541-605.

[Griss 82]
PSL Implementation Guide by M. L. Griss, E. Benson, R. Kessler, S. Lowder, G. Q. Maguire, Jr. and J. W. Peterson. Department of Computer Science, University of Utah. May 1982.

[Intel 84]
Microcontroller Handbook, 1984 Edition Intel Corporation, Santa Clara, CA 95951.

[Kessler 84]
Peep -- An Architectural Description Driven Peephole Optimizer by Robert R. Kessler. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6, June 1984.

[Stolfo 83]
Architecture and Applications of DADO: A Large Scale Parallel Computer for Artificial Intelligence by Stolfo S. J., D. P. Miranker, and D. E. Shaw in "Proceedings of the Eighth International Joint Conference on Artificial Intelligence".

[Stolfo 84]
DADO: A Parallel Processor for Expoert Systems by Stolfo S. J., and D. P. Miranker in "Proceedings of the 1984 International Parallel Processing Conference."

[Utah]
PSL Manual, Department of Computer Science, University of Utah.