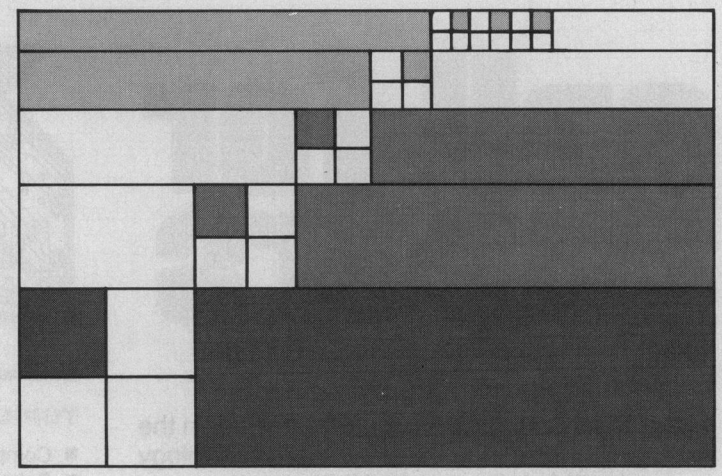
The Scheme-79 single-chip microcomputer implements an automatic storage allocation system and an interrupt facility to support direct interpretation of a variant of the Scheme dialect of Lisp.

Scheme-79 — Lisp on a Chip

Gerald Jay Sussman, Jack Holloway, Guy Lewis Steel, Jr., and Alan Bell

MIT Artificial Intelligence Laboratory



We have designed and implemented Scheme-79, a single-chip microcomputer that directly interprets a typed-pointer variant of Scheme,[1] a dialect of Lisp.[2] To support this interpreter, the chip implements an automatic storage allocation system for heap-allocated data and an interrupt facility for user interrupt routines implemented in Scheme. This article describes why Scheme is particularly well suited to direct implementation of a Lisp-like language in hardware, how the machine architecture is tailored to support the language, the design methodology by which the hardware was synthesized, the performance of the current design, and possible improvements to the design. We developed an interpreter for Scheme written in Lisp. This interpreter can be viewed as a microcode specification that is converted into actual hardware structures on the chip by successive compilation passes. To effect this we developed a language, embedded in Lisp, for describing layout artwork. This language allows procedural definition of generators for architectural elements, which are generalized macro components. The generators accept parameters to produce the specialized instances used in a particular design. Our design methodology made it possible to design and lay out this chip in five weeks.

Why Lisp?

Lisp is a natural choice among high-level languages for implementation on a single chip (or in any hardware,[3,4] for that matter). It is a very simple language, in which a powerful system can be built with only a few primitive operators and data types. In Lisp, as in traditional machine languages, there is a uniform representation of programs as data. Thus the same primitive operators used by a user's program to manipulate his data are used by the system interpreter to effect control. We have added to the traditional Lisp data types to allow representation of programs in a form that can be efficiently interpreted by our hardware.

Lisp is an object-oriented, as opposed to value-oriented, language. The Lisp programmer does not think of variables as objects of interest—bins in which values can be held. Instead, each data item is itself an object to be examined and modified, with an identity independent of the variable(s) used to name it.

Lisp programs manipulate primitive data such as numbers, symbols, and character strings. What makes Lisp unique is that it provides a construction material, called list structure, for gluing pieces of data together to make compound data objects. Thus a modest set of primitives provides the ability to manufacture complex data abstractions. For example, a programmer can make his own record structures out of list structure without having the language designer install special features explicitly for him. The same economy applies to procedural abstractions. Compound procedures defined by the user have the same status as the initial system primitives. The Lisp user can also manufacture linguistic abstractions, because programs can be manipulated as data and data can be interpreted as programs.[5]

We chose the Scheme dialect because it offers further economies in the implementation of our machine. It is tail recursive and lexically scoped. Tail recursion is a call-save-return discipline in which a called procedure is not expected to return to its immediate caller unless the immediate caller wants to do something with it after the called procedure finishes. Tail recursion allows convenient definition of all common control abstractions in terms of just two primitive notions, procedure call and conditional,[6] without significant loss of efficiency.[7-9]

0018-9162/81/0700-0010$00.75

We also adopted lexical scoping of free variables, as in Algol-based languages. This is simpler to implement on the chip than the shallow dynamic binding schemes used in traditional Lisps, and it is more efficient than the alternative of deep dynamic binding.[10] In cases such as the rapidly changing environments in multiprocessing applications, lexical binding is more efficient than any dynamic binding strategy.

How the machine supports Scheme

All compound data in the system is built from list nodes, which consist of two pointers (called the CAR and the CDR, for historical reasons). A pointer is a 32-bit object with three fields: a 24-bit data field, a seven-bit type field, and one bit used only by the storage allocator. The type identifies the object referred to by the data field. Sometimes the datum referred to is an immediate quantity; otherwise, the datum points to another list node.[3,11,12] Figure 1 shows the format of a list node.

Lisp is an expression-oriented language. A Lisp program is represented as list structure, which notates what would be the parse tree in a more conventional language. Lisp expressions are executed by an evaluation process in which the evaluator performs a recursive tree walk on the expression, executes side effects, and develops a value. At each node of the expression, the evaluator dispatches on the type of that node to determine what to do. It may decide that the node is an immediate datum and is to be returned as a value, a conditional expression that requires the evaluation of one of two alternative subtrees based on the value of a predicate expression, or that it is an application of a procedure to a set of arguments. In the latter case, the evaluator recursively evaluates the arguments and passes them to the indicated procedure.

Lisp surface expressions are converted into machine programs, which are represented as list structure made of typed pointers. The type fields in our machine are analogous to the opcodes of a traditional instruction set. The dispatch to be made by the evaluator is encoded in the type field of the pointer to the translated expression. The transformation from Lisp to machine language preserves the expression structure of the original Lisp program. In the Scheme-79 architecture, the evaluator's recursive tree walk over expressions takes the place of the linear sequencing of instructions found in traditional computers. (The IBM 650[13] had an instruction set in which each instruction pointed at the next one. However, it was used to implement a traditional linear programming style, not to implement parse trees.) We call the machine language for our chip "S-code."

The S-code representation differs from the original Lisp expression in several ways. Particularly, it distinguishes between local and global variable references. A local variable reference is an instruction containing the lexical address of its value relative to the current environment structure. A global variable reference is a pointer that points directly at the global value of the symbol. A constant is a literal piece of data. When interpreted, a constant is an instruction that moves the appropriate data into the accumulated value. Certain primitive procedures (CAR, CDR, CONS, EQ, etc.) are realized directly as machine opcodes. Procedure calls are chains of instructions that accumulate the arguments and then apply the procedure. Control sequences (PROGN) are chains of instructions that direct the interpreter's tree walk. Conditionals are forks in a sequence chain.

In Figure 2, a simple Lisp program for appending two lists is represented in the S-code. The details of the S-code language are explained elsewhere;[14] this example is only to give an idea of what the S-code is like. A user of the Scheme-79 chip should never see the S-code.

At each step, the hardware evaluator dispatches on the state of the expression under consideration. The state of

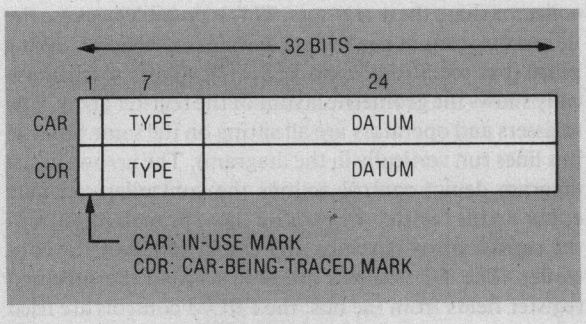


Figure 1. Format of a list node.

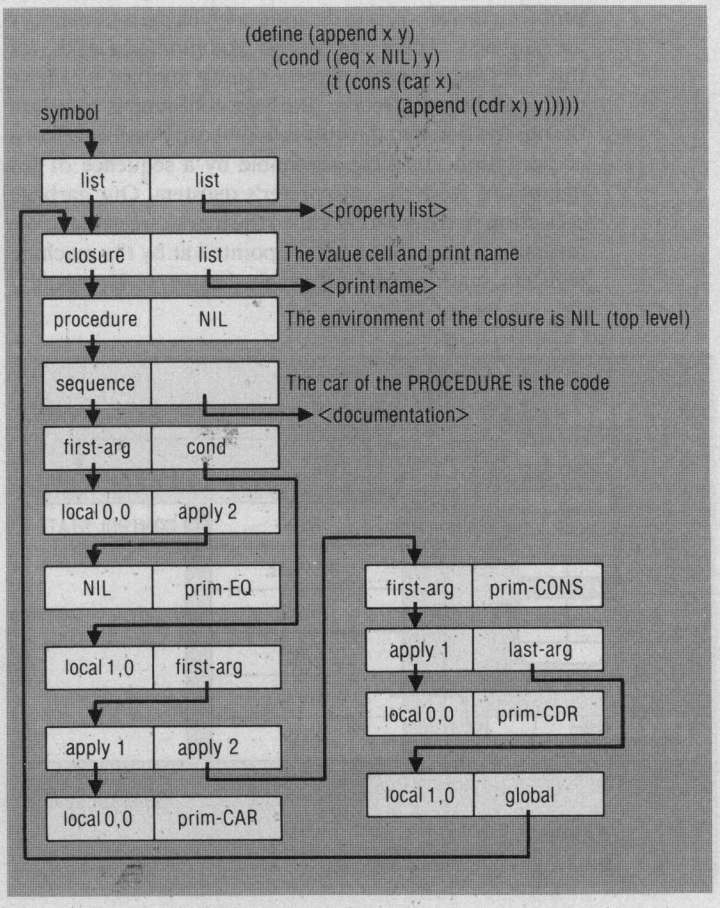


Figure 2. The S-code for APPEND.

the evaluator is kept in the set of registers shown in Figure 3. One register, VAL, holds the value of the last expression. EXP is a register for the current expression. If the current expression is a procedure call, the arguments are evaluated and built into a list kept in the ARGS register. When the arguments are all evaluated, the procedure is invoked. This requires the formal parameters of the procedure to be bound to the actual computed parameters. This binding is effected by changing the lexical environment register, DISPLAY, to point at the CONS of the ARGS register and the environment pointer of the closed procedure being applied. When evaluating the arguments to a procedure, the evaluator might have to recurse to obtain the value of a subexpression. The state of the evaluator must be restored when the subexpression returns with a value; this requires the state to be saved before recursion. The evaluator maintains a pointer to the stack of pending returns and their associated states in the register called STACK.

In our machine, data, programs, and even the stack are represented in list structure allocated from a heap memory. Neither the user nor the system interpreter is interested in how the memory is allocated, but memory is finite—and normal computation leads to the creation of garbage. For example, entries built on the stack during the evaluation of subexpressions are usually useless after the subexpression has been evaluated, but they still consume space in the heap. Thus the storage allocator must have means to reclaim memory that has been allocated but can no longer affect the future of the computation. The problem, therefore, is to determine which parts of memory are garbage. There are several common strategies for this.[15,16] One involves reference counts; another, which we use, is garbage collection. Garbage collection is based on the observation that the only cells that can possibly affect a computation are those reachable by a sequence of list operations from the interpreter's registers. Our garbage collection strategy is thus called the mark-sweep plan. We recursively trace the structure pointed at by the machine registers, marking each cell reached as we go. Eventually, we mark the transitive closure of the list access operations starting with the machine registers; therefore, a cell is marked if and only if it is accessible. We then scan all memory; any location not marked is swept up as garbage and made reusable.

Usually, recursive traversal of a structure requires an auxiliary stack. This is unfortunate for our machine, for two reasons. First, it needs list structure to build stack; second, it is (presumably) garbage collecting because we ran out of room in which to build new list structure. Deutsch, Schorr, and Waite[17] developed a clever method of tracing structure without auxiliary memory; we use it. Our sweep phase has a two-finger compaction algorithm[11] that relocates all useful structure to the bottom of memory.

The chip also supports an interrupt system. The interrupt handlers are written in Scheme. Thus the user can, for example, simulate parallel processing or handle asynchronous I/O. The problem here is that the state of the interrupted process must be saved during the execution of the interrupt routine, so that it can be restored when the interrupt is dismissed. This is accomplished by building a data structure to contain the state of the relevant registers and pass it to the interrupt routine as a continuation argument.[18] The interrupt routine can then do its job and resume the interrupted process, if it wishes, by invoking the continuation as a procedure. The interrupt mechanism is also used to interface the garbage collector to the interpreter.

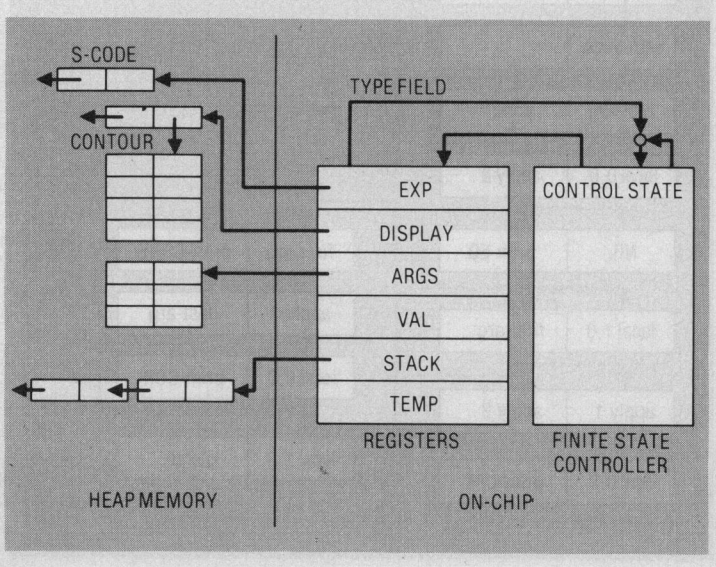


Figure 3. The hardware evaluator.

The Scheme-79 architecture

The Scheme-79 chip implements a standard von Neumann architecture in which a processor is attached to a memory system. The processor is divided into two parts, the data paths and the controller. The data paths consist of a set of special-purpose registers with built-in operators. The registers are interconnected with a single 32-bit bus. The controller is a finite-state machine that sequences through microcode and implements the interpreter and garbage collector. At each step, it performs an operation on some of the registers. For example, a sequence chain is followed by setting the EXP register to the CDR of the contents of the EXP register. The controller selects its next state on the basis of its current state and the conditions developed within the data paths.

There are 10 registers, each with specialized characteristics. To save space, the interpreter and the garbage collector share these registers. This is possible because the interpreter cannot run while a garbage collection is taking place (but see Steele[19] and Baker[20]). Figure 4 schematically shows the geometric layout of the register array. The registers and operators are all sitting on the same bus (the bus lines run vertically in the diagram). The arrows in the diagram depict control actions the controller can take either on the register array or on those branch conditions the register array develops that can be tested by the controller. The TO controls are used to load the specified register fields from the bus; the FROM controls are used to read the specified register onto the bus.

The division of storage words into mark, type, and data fields is reflected in the physical structure of most of the registers. On each cycle the registers can be controlled to gate one of the registers onto the bus and selected fields of the bus into another register. The bus is extended off the chip through a set of pads. The external world is conceptualized as a set of registers with special capabilities. The external ADDRESS register is used for accessing memory and can be set from the bus; the pseudoregister MEMORY can be read onto the bus or written from the bus. The actual access is performed to the list cell addressed by the ADDRESS register. The CDR bit controls which half of the cell is being accessed. One more external register, INTERRUPT (which can be read onto the bus), contains the address of a global symbol whose value (its CAR) is an appropriate interrupt handler.

The finite-state controller for the Scheme-79 chip is a synchronous system composed of a state register and the control map, which is a large piece of combinational logic. The control map is used to develop, from the current state stored in the state register, the control signals for the register array and pads, the new state, and controls for selection of the sources for the next sequential state. One bit of the next state is computed using the current value of the selected branch condition (if any). The rest of the next sequential state is chosen from either the new state generated by the control map or from the type field of the register array bus (dispatching on the type).

Including both the interpreter and garbage collector on the chip makes it difficult to create a compact realization of the map from old state to new state and control function. If we tried to implement this straightforwardly, using a programmed logic array, or PLA, for the map, this structure would physically dominate the design and make the implementation infeasible. The map logic was made feasible by several decompositions; only a few of the possible combinations of register controls are actually used. For example, only one register can be gated onto the bus at one time. We used this interdependence to compress our register control. Instead of developing all of the register controls from one piece of logic (which would have to be very wide), we developed an encoding within the main map of the operation to be performed and the registers involved. This encoding was then expanded by an auxiliary map (also constructed as a PLA) to produce the actual control signals. This approach was inspired by the similar decomposition in the M68000,[21-23] which uses both vertical and horizontal microcode.

Unfortunately, this does not solve the problem. When examining the microcode, we found many microcode sequences to be nearly identical, differing only in the particular registers being manipulated. To take advantage of this regularity, we extended the previous decomposition to allow common short sequences to be encoded in the second-level PLA. Thus the Micro step consists of a one-(or more) cycle operation (Nano opcode) and a specification of registers to be used as source (from) and destination (to) operands. During the Micro step, the Nano PLA merges the specified source and destination registers into the extracted sequence.

Further savings were realized by taking advantage of the many identical sequences of microinstructions (except for renaming of registers) that contained no conditional branches. For example, to take the CAR of a specified source register and put it in a specified destination register takes two cycles: one to put the source out on the pads and

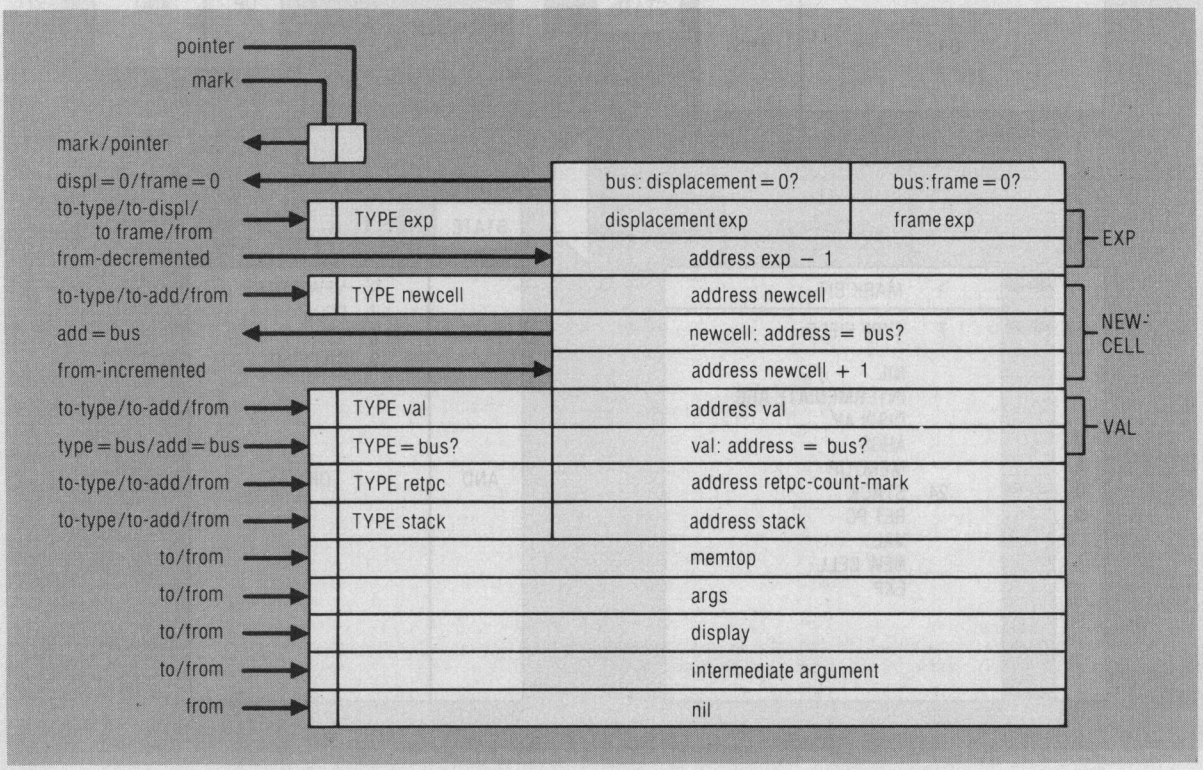


Figure 4. The register array.

tell the memory to latch the address and another to read the data from memory into the destination register. Since sequences like CAR are common, the microcode could be compressed by allowing a limited form of microcode subroutine. We did this rather painlessly by extending the horizontal (Nano) microcode map into a full-state machine (Figure 5). These common subsequences are then represented as single cycles of the Micro engine, which each stimulate several cycles of the Nano engine. To make this work, the vertical (Micro) sequencer must be frozen while the Nano sequencer is running. This mechanism was needed anyway, to make it possible for the chip to wait for memory response.

Still further savings were realized by subroutining common subsequences in the source microcode. Some of the common subsequences could not be captured by the Nano engine because they involved conditional branch operations, which the Nano engine was incapable of performing. They could, however, be subroutined in the source code by the conventional use of a free register to hold the return microcode address (Micro state) in its type field. Many microcode instructions were saved by this analysis.

Synthesizing the Scheme-79 chip

The major problem introduced by VLSI is that of coping with complexity—logic densities of over a million gates per chip will soon be available. The design of the Scheme-79 chip takes advantage of the techniques and perspectives gained in confronting the complexity problem in the software domain. Hardware design will increasingly become analogous to the programming of a large software system; therefore, we developed two languages embedded in Lisp to specify the behavior and the structure of our chip.

The first of these languages describes the algorithm to be embodied in the hardware. We chose Lisp as a convenient base language for embedding the description of the Scheme interpreter. By adding primitive operations that model the elementary hardware capabilities, we arrived at a form of Lisp (micro-Lisp) suitable for expressing our microcode. Although micro-Lisp has the structure of Lisp, its basic operations are ultimately for side effect rather than for value. They represent actions that can be performed on the registers on the chip. By using this form of Lisp, we can take advantage of such high-level language features as conditionals, compound expressions, and user macro definitions. For example, the following fragment of the on-chip storage allocator is written in micro-Lisp:

```
(defpc mark-node
   (assign *leader* (&car (fetch *node-pointer*)))
   (cond ((and (&pointer? (fetch *leader*))
               (not (&in-use? (fetch *leader*))))
          (&mark-car-being-traced! (fetch *node-pointer*))
          (&rplaca-and-mark! (fetch *node-pointer*)
                             (fetch *stack-top*))
          (go-to down-trace))
```

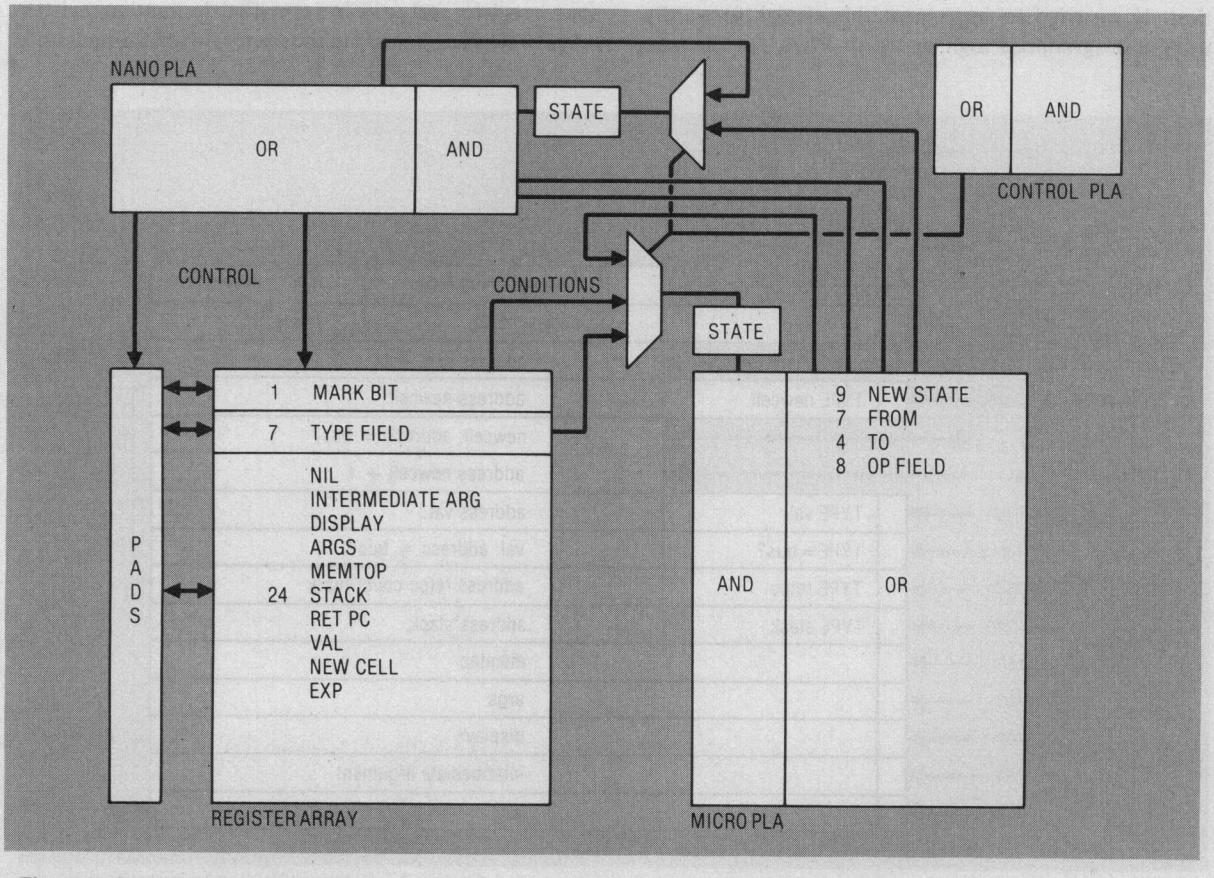


Figure 5. Scheme-79 architecture.

```
        (t
         (&mark-in-use! (fetch *node-pointer*))
         (go-to trace-cdr))))

(defpc down-trace
   (assign *stack-top* (fetch *node-pointer*))
   (assign *node-pointer* (fetch *leader*))
   (go-to mark-node))
```

This example shows two subsequences of the microcode, labeled mark-node and down-trace. Down-trace is a simple sequence that transfers the contents of the machine registers *node-pointer* and *leader* into *stack-top* and *node-pointer*. Although the storage allocator was written as if it uses a set of registers distinct from those of the evaluator, there are micro-Lisp declarations that make these names equivalent to the registers used by the evaluator. The mark-node sequence illustrates the use of a compound expression that refers to the result of performing the CAR operation on the contents of the *node-pointer* register. The CAR operation is itself a sequence of machine steps that access memory. (This is an example of a Nano operation that performs a step to output the address to be accessed. A second step reads the data from the external memory into the *leader* register.) The compound boolean expression that tests the contents of the *leader* register as to whether it is a pointer to an unused cell is compiled into a series of microcode branches, which transfer to separate chains of microinstructions corresponding to the consequent and alternative clauses.

One benefit of embedding the microcode language in Lisp is that by providing definitions for the primitive machine operators that simulate the machine actions on the registers we can simulate the operation of the machine by running our microcode as a Lisp program. Micro-Lisp is also an easy language to compile.

Micro-Lisp is compiled into artwork by a cascade of three compilation steps. The first phase transforms micro-Lisp into a linear, sequential machine language. It removes all embedded structure such as the composition of primitive operators. This involves the allocation of registers to hold the necessary intermediate results. Conditionals are linearized and the compound boolean expressions are transformed into simple branches. The machine language is specialized to the actual target machine architecture by the code generators, which transform the conceptually primitive operators of micro-Lisp into single major-cycle machine operations, which can be encoded as single transitions of Micro (the vertical microcode state machine).

The second phase of the compilation process assembles the linear microcode into programming for the PLAs that control the Micro and Nano sequencers. The Nano PLA is constructed by selecting the needed nanocode from a dictionary of possible nano sequences. Nano instructions can be viewed as macros or subroutines invoked by Micro instructions. The operation field of the Micro instruction selects the Nano instruction; the Nano instruction can then selectively enable decoding of the from and to fields.

The third major phase of the compilation is performed by the PLA architectural element generator, an artwork synthesis procedure written in the layout language described below. This procedure has several parameters: one is the PLA specifications output by the previous phases; others control special details of the clocking, the order of bits in fields in the input and output wiring, the ground mesh spacing, and the option of folding the PLA for adjustment of the aspect ratio. These parameters provide flexibility in accommodating the PLA layout to the other structures on the chip.

The layout language

Current practice in chip design uses the idea of a *cell,* a particular combination of primitive structures that can be repeated or combined with other cells to make a more complex structure. The advantage of this approach is that a cell performing a common function is defined only once. More important, the cell encapsulates and thus suppresses much detail so that the problem of design is simplified. As in programming, we extend this notion (after Johannsen[24]) by *parameterizing* our cells to form compound abstractions that represent whole classes of cells. These classes can vary both in structure and function. We call these abstract parameterized cells *architectural elements.*

For example, a simple nMOS depletion load pullup comes in a variety of sizes. The particular length-to-width ratio of such a transistor is a simple numerical parameter. A parameterized pullup is a simple architectural element. It encapsulates the particular rules for constructing the pullup, including the placement of the contact from the poly to the diffusion layer and the placement of the ion implant. A more interesting architectural element is an n-way selector, which is parameterized by the number of inputs. In this case, such higher-level details as the means of driving the input lines and the particular logic by which the selector is implemented are suppressed. Small selectors may be more effectively implemented with one kind of logic, and large selectors with another.

These are still only simple parameters; we have developed much more powerful architectural elements. For example, a finite-state machine controller can be implemented as a PLA and state register. We have constructed a PLA generator parameterized by the logical contents and physical structure. This generator is used with a compiler that takes a program to be embodied in the state machine to produce the state machine controller, an architectural element parameterized by a program written in a high-level language. Although we have not completely parameterized it, we can think of our register array generator as an architectural element. In fact, an entire high-level language interpreter module can be accurately defined as an architectural element parameterized by the interpreter program augmented with declarations that describe how the interpreter data is represented in registers.

In our system, an architectural element generator is a procedure written in Lisp augmented by a set of primitive data-base and layout operators. This augmented Lisp is the layout language. (The layout language used in the design of the Scheme-79 chip has evolved substantially. It is described in Batali et al.[25])

The layout language primitives create representations of elementary geometric entities such as points and boxes on a particular mask layer. For example:

```
(PT 3 4)   is the point at location (3,4)
(THE-X (PT 3 4))   is 3
(BOX 'POLY 3 4 5 6) is a box on the poly layer from (3,4) to (5,6)
```

Sequences of connected boxes on a single layer can be conveniently constructed by specifying a width, an initial point, and a sequence of directions for proceeding along the desired path. The paths must be rectilinear; they are specified as a sequence of movements in either the x or y coordinate directions. Each movement can be either incremental or to an absolute position.

```
(BOXES (POLY 3) (PT 3 4) (+X 20) (Y 35) (X 70))
  specifies a path in 3 wide poly from (3,4) to (23,4)
  to (23, 35) to (70, 35).
```

The layout language also provides, as primitives, certain common structures used in nMOS layouts, such as contact cuts between the various layers.

```
(CONTACT V POLY (PT 3 4)) makes a metal-to-poly contact which
  is vertically oriented and is centered at point (3, 4).
```

We can combine pieces of layout by instantiating each piece separately. We can then make compound cells by giving a name to a program that instantiates all of its pieces. The cell can be parameterized by arguments to the procedure that generates it. The following layout procedure creates a depletion-mode pullup of a given width and length. It encapsulates knowledge of the design rules by referring to globally declared process parameters.

```
(deflayout general-pullup (length width)
   (boxes (diff width) (pt 0 -1) (+Y (+ length 2)))
   (boxes (poly (+ (* 2 *poly-overhang*) width))
          (pt 0 0) (+Y length))
   (call (butting-contact))
   (boxes (implant (+ (* 2 *implant-overhang*) width))
          (pt 0 (- 0 *implant-overhang*))
          (+Y (+ length (* 2 *implant-overhang*)))))
```

A compound cell can be instantiated as part of a larger structure by invoking its name. The instance created by a call to a compound cell can be translated, rotated, and reflected to place it appropriately within the larger structure.[26] For example, we could put a pullup where we want it by writing

```
(call (general-pullup 8 2) (rot 0 1) (trans (pt 24 35)))
```

Each object can be given a local symbolic name relative to the larger structure of which it is a part. These names can be referred to by means of paths, which describe the sequence of local names from the root of the structure.[27] These symbolic names are attached to the coordinate systems of cells. They are useful for describing operations when creating artwork because they eliminate the need to explicitly write out the numerical values of the operands. For example, it is possible to place a driver cell at the end of a register so that its control outputs align with (and connect to) the control inputs of the first cell of the register column (think of the rows of the register array as bit-slices and of the columns as the individual registers). To effect this we call the drive cell generator to get an instance of the driver and then align the newly called-out instance so that the appropriate points coincide. We also can inherit names from substructure to make new local names.

```
(set-the 'exp-driver (call (regcell-driver)))
(align (the exp-driver)
       (the-pt end ph1-to ph2-driver exp-driver)
       (the-pt to-type-exp array))
(set-the 'to-type-exp (the-pt start sig to-driver exp-driver))
(set-the 'from-exp (the-pt start sig from-driver exp-driver))
```

A form (the-pt . . .) is a path-name: "the point which is the end of the ph1-to of the ph2-driver of the exp-driver (of me)."

Virtual instances of cells can be made. These do not actually create artwork, but they can be interrogated for information about such properties of the virtual artwork as the relative position of a particular bus metal or the horizontal pitch. In the following fragment, pullup-pair is made to be the local name of a virtual instance of the pullup-pair cell. This is used to extract parameters to control the elaboration of pullup-gnd so that it is compatible with the pullup-pair cell and can be abutted to it.

```
(deflayout pullup-gnd (gnd-width)
   (set-the 'v-pitch -10)
   (set-the 'pullup-pair
            (invoke* (pullup-pair gnd-width)))
   (set-the 'vdd
            (wire (metal 4)
                  (pt (the-x end vdd pullup-pair) 0)
                  (Y -10)))
   (set-the 'vdd2
            (wire (metal 4)
                  (pt (the-x end vdd pullup-pair) -5)
                  (X (the-x vdd2 pullup-pair))))
   (set-the 'gnd
            (wire (metal gnd-width)
                  (pt (the-x end gnd pullup-pair) 0)
                  (Y -10)))
   (boxes (metal 4) (pt (the-x end gnd) -5)
          (X (the h-pitch pullup-pair))))
```

The layout language system is intended to be part of an interactive environment in which designs are developed by combining incremental modifications to and adaptations of existing fragments of design. Thus a layout procedure is not just a file transducer that takes an input design and cranks out artwork. Rather, it produces a data base that describes the artwork to be constructed. The output artwork is just one way of printing out some aspects of this data base. Other information contained in the data base embodies the user's conception of the structure of the artifact he is constructing, including mnemonic names for parts of his design. This data base can be interrogated by the user to help him examine his design, to annotate it, and to help produce incremental changes when debugging is necessary.

History

The project started with Guy Steele's painstaking hand layout of a prototype interpreter chip[28,29] as part of the 1978 MIT class chip project. The prototype differed from the current design in several ways. It used separate sets of registers for the storage allocator and evaluator processes, while our new design shares the same set of registers between them. Although this saves space by time multiplexing, it precludes the use of a concurrent garbage collector algorithm. The prototype chip was fabricated

but never tested because a fatal artwork error was discovered (through a microscope!).

The design for the Scheme-79 chip began at the MIT Artificial Intelligence Lab in mid-June 1979. The first task was to construct the microcode we wanted the machine to interpret. We adapted a previous experimental implementation of the Scheme language (written in LISP for the PDP-10) by defining a storage representation and adding a garbage collector. We studied the microcode to determine which architecture would be effective for its implementation. The next step was to create a layout language in order to write procedures that would build the chip artwork. This was followed by the simultaneous construction of a compiler of the microcode to the PLA implementation and the layout of the register array. Complete preliminary implementations of the main structures of the chip were ready in our data base by July 8, 1979.

At this point we went to Xerox Palo Alto Research Center to use their Icarus automated draftsman program[30] to assemble and interconnect these pieces. This was the hardest part of the job; it took almost two weeks. The first completely assembled version of the Scheme-79 chip was completed on July 19, 1979. The implementation was done, except for the discovery of errors.

Some errors were discovered by the sharp eyes of people at Xerox and MIT. We had an 8×10-foot check plot with features of approximately 1/10 inch. Within a few weeks, we had discovered and corrected about 10 design-rule violations or artwork errors and two nonfatal logic bugs. At that point, a program was written by Clark Baker that extracted an electrical description (in terms of nodes and transistors) from our artwork. Baker's program discovered an implausible circuit fragment, which turned out to be an extra power pad superimposed on one of the data pads! This electrical description was then simulated with a program written by Chris Terman, which was based upon an earlier program developed by Randy Bryant.[31] The simulator helped find five additional serious logic errors, a rare PLA compiler bug, an invisible (1/8 minimum feature size) error introduced in the artwork conversion process, and an extraneous piece of polysilicon that had magically appeared during the repair of previous errors. The final simulations checked out a complete garbage collection and the evaluation of a trivial (450 microstep) user program. This experience indicates that efficient tools for checking both the physical and logical design are essential.

The chip went out for fabrication on December 4, 1979, as part of the MPC79 Multi-University Multiproject Chip-Set compiled by the LSI Systems Area of the System Science Laboratory at Xerox PARC. Using a process with a mininum line width of five microns (lambda = 2.5 microns), the Scheme-79 chip was 5926 microns wide and 7548 microns long, an area of 44.73 square millimeters. The masks were made by Micro Mask, Inc., and the wafers were fabricated by Hewlett-Packard's integrated circuit processing laboratory.

On January 9, 1980, we received four chips bonded into packages. By then, Howard Cannon had designed and fabricated a board to interface the Scheme-79 chip to the MIT Lisp Machine. This was a substantial project, which included an interface to allow our chip to access Lisp Machine memory. The interface board contains a map for chip addresses to Lisp Machine memory addresses, a programmable clock to allow the Lisp Machine to vary the durations of the clock phases and the interphase spaces, debugging apparatus to allow the Lisp Machine to set and read the state and internal registers of the chip, circuitry for allowing the Lisp Machine to interrupt the Scheme chip, circuitry to allow the Lisp Machine to single-step the chip, and an interface from chip bus protocols to Lisp Machine bus protocols. This interface project materially contributed to the success of the Scheme-79 chip project, because it allowed testing to begin almost immediately upon receipt of the chip.

The first chip we unpacked had a visible fatal flaw in the metal layer. The second one could load and read state but would not run. The third chip seems to work. It has successfully run programs, garbage collected memory, and accepted interrupt requests. We have found two nonfatal design errors that escaped our previous simulation and testing. One is a subtle bug in the garbage-collector microcode. It could cause rare disasters, but will never be a problem in the actual programs likely to be run by the chip. The other is a race condition in the logic associated with the pad used by the chip to signal the need for an interrupt to collect garbage. Luckily, this function is redundant and can be assumed by the interface.

Performance of the Scheme-79 chip

Three areas independently limit the performance of the machine: the algorithms embodied by the chip are not optimal; there are architectural improvements that can reduce the number of machine cycles per step in the interpreter; and there are electrical performance limits to the design.

The interpreter on the chip uses the heap memory system for all of its data structures, including the interpreter's stack. This becomes the dominant source of garbage produced by execution of a program; reclaiming this garbage is the dominant cost of execution. There are several ways to attack this problem.

We have estimated, by examining fragments of the microcode, that we allocate a heap cell for every 10 cycles of computation. The cost of allocating this cell is in two parts: approximately eight cycles are required to perform the allocation, and (assuming that half of the heap is reclaimed per garbage collection) 35 cycles are required to reclaim each cell. This, of course, implies that the current machine will spend 80 percent of its time in the storage allocator. The garbage collector itself has respectable performance. At a one-MHz clock rate, collection of a one-megabyte heap (128K Lisp nodes) takes less than six seconds. However, a copying collection algorithm[32,33] would generate a third as many memory accesses.

Of course, if the evaluator used a more traditional stack structure it would generate considerably less garbage. We chose not to do this because it would increase the complexity of many parts of the system. For example, there would have to be a mechanism for allocating nonlist structures in memory. Furthermore, the garbage collector would need to treat these structures specially and would

Now available in 18″ × 24″ poster size: our April 1981 cover, "Brutralized Bruegel." Printed in full color on heavy glossy stock (without this promotional copy, naturally) and suitable for framing, it costs $5.95 for members and $7.95 for nonmembers including postage and handling.

Prepaid orders only, please. Use the order form on the inside back cover. Ask for catalog number P1C.

have to mark from pointers stored in the stack. Also, a linear stack regime makes retention of control environments very complicated.[34] This impacts such user multiprocessing features as interrupts and nonlocal catch points.

Richard Stallman has observed that if we sequentially allocate the list nodes for our stack in a separate area, we can normally deallocate when popping, after making a simple check for retained control environments. Assuming that such retained control environments are rare in comparison to the usual uses of the stack, it will be almost as efficient as a normal linear stack on a conventional machine.

Interpreter performance can also be improved by optimizing use of the stack. This can be effected by exploiting the regularities of the stack discipline, which make many of the stack operations redundant. In the *caller-saves* convention (which the Scheme-79 chip implements), the only reason to push a register onto the stack is to protect its contents from being destroyed by unpredictable uses of the register during the recursive evaluation of a subexpression. Therefore, one source of redundant stack operations lies in the fact that a register is saved even if the evaluation of the subexpression does not affect the contents of that register. If we could look ahead in time, we could determine whether or not the register will retain its contents through the unknown evaluation. This is one standard kind of optimization done by compilers, but even a compiler cannot optimize all cases because the execution path of a program generally depends on the data being processed. Instead of looking ahead, we can try to make the stack mechanism lazy, in that it postpones pushing a register until its contents are about to be destroyed. The key idea is that each register has a state that indicates whether its contents are valuable. If a valuable register is about to be assigned, it is at that moment pushed. To make this system work, each register that can be pushed has its own stack, to allow decoupling of the stack disciplines for each of the registers. Each register-stack combination can be thought of as having a state that encodes some of the history of previous operations. The combination is organized by a finite-state automaton that mediates between operation requests and the internal registers and stack. This automaton serves as an on-the-fly peephole optimizer, which recognizes certain patterns of operations within a small window in time and transforms them so as to reduce the actual number of stack operations performed. We have investigated this strategy[35] and believe it can be implemented in hardware easily; it should substantially improve the performance of the algorithm. Pilot studies indicate that this technique can save three out of every four stack allocations in the operation of the interpreter.

Other trade-offs were made in the architecture of the chip. For example, our register array has a single bus. This forced the microcode to serialize many register transfer operations that logically could have been done in parallel. (A more powerful bus structure would result in a significant speedup.) We also decided to use simple register cells rather than buffered registers, which means that a register cannot be read and written at the same time. This is not usually a problem, because on a single-bus machine it is not useful to read out and load the same register. It does, however, cause increment and decrement operations to take two microcycles rather than one. This is significant because increment and decrement operations are in the innermost loop of the local variable lookup routine, and decrementing the frame and displacement field takes twice as long as is really necessary. Finally, we could have used more registers. In several cases, an extra intermediate would result in fewer register shuffles and CONSs to get something done. For example, special argument registers for holding the arguments for primitive one- and two-argument functions could make a serious dent in the storage allocated in argument evaluation and, ultimately, in the time taken to collect garbage. This optimization can be combined with our stack optimizer strategy (mentioned above). In fact, the entire argument about stack optimization can be thought of as an architectural issue because of the simple implementation of our peephole-optimizing automata in hardware.

We also made significant tradeoffs in the electrical characteristics of the Scheme-79 chip: the bus is not precharged; the PLAs are ratio logic, not precharged; there is a no on-chip clock generator; many long runs are made on poly and diffusion that should be made on metal; and some buffers are not sized to drive the long lines to which they connect. We also used a selector design with implanted pass transistors, an exceptionally slow circuit. Careful redesign of some of the circuitry on the chip would greatly improve performance.

The actual measured performance of our chip on a sample program is shown below. We calculated the values of Fibonacci numbers by the doubly recursive method. (This explodes exponentially; computing each Fibonacci number takes $(1+\sqrt{5})/2$ times the time it takes to compute the previous one.) This is an excellent test program because it thoroughly exercises most of the mechanisms of the interpreter. Sums were computed using Peano arithmetic, which is the only arithmetic available on the chip. The program is as follows:

```
(define (+ x y)
   (cond ((zerop x) y)
         (t (+ (1- x) (1+ y)))))
(define (fib x)
   (cond ((zerop x) 0)                    ;If 0. result is 0.
         ((zerop (1- x)) 1)               ;If 1. result is 1.
         (t (+ (fib (1- x))
               (fib (1- (1- x)))))))
```

We computed (fib 20.) = 6765. with two different memory loadings, with a clock period of 1595 nanoseconds (not the top speed for the chip) and a memory of 32K Lisp cells. When the memory was substantially empty (so that garbage collection was maximally efficient), the Scheme-79 chip took about one minute to execute the program. With memory half full of live structure (a typical load for a Lisp system), the chip took about three minutes to execute the program.

We also compared this performance with that of our standard MacLisp interpreter on the same program, running on the DEC KA10 with a Scheme interpreter written in MacLisp. Lisp did not garbage collect during this operation, but it took about 3.6 minutes. The MacLisp Scheme interpreter (with unknown memory loading)

took about nine minutes, with about 10 percent of the time spent in the garbage collector. ■

Acknowledgments

We would like to thank Lynn Conway and Bert Sutherland of the Xerox Palo Alto Research Center System Science Laboratory and Jon Allen and Paul Penfield of MIT, for help, support, and encouragement. We would also like to thank all those people who gave us ideas and technical assistance, particularly Neil Mayle, Thomas F. Knight, Howie Shrobe, and Richard Zippel at MIT, and Lynn Conway, Jim Cherry, Dick Lyon, and Jerry Roylance at Xerox PARC. These people read and commented on designs, found bugs, and proposed patches. The resulting chip still had many bugs, which were found by sharp-eyed friends and by the simulation support provided by Clark Baker and Chris Terman of the MIT Laboratory for Computer Science. We would like to thank Howard Cannon for designing and constructing the apparatus for testing Scheme-79 chips on our MIT Lisp machines. Recently, we have had excellent suggestions for improving the next design from Richard Stallman and John Rees. Paul Penfield, Ron Rivest, Neil Mayle, John Batali, and Howie Shrobe have had good ideas about better design tools. We are sure that it would have been impossible to undertake so large a design without the extensive help provided by so many friends.

This article describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the following sources: Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-75-C-0643, National Science Foundation Grant MCS77-04828, and Air Force Office of Scientific Research Grant AFOSR-78-3593.

References

1. Guy Lewis Steele, Jr., and Gerald Jay Sussman, *The Revised Report on SCHEME: A Dialect of LISP,* MIT AI Memo 452, Cambridge, Mass., Jan. 1978.
2. John McCarthy et al., *LISP 1.5 Programmer's Manual,* MIT Press, Cambridge, Mass., 1962.
3. Alan Bawden, Richard Greenblatt, Jack Holloway, Thomas Knight, David Moon, and Daniel Weinreb, *LISP Machine Progress Report,* MIT AI Memo 444, Cambridge, Mass., Aug. 1977.
4. Eiichi Goto, Tetsuo Ida, Hiraki Kei, Masayuki Suzuki, and Inada Nobuyuki, "FLATS, A Machine for Numerical, Symbolic, and Associative Computing," *Proc. 6th Ann. Symp. Computer Architecture,* Apr. 1979, pp. 102-110.*
5. Guy Lewis Steele, Jr., and Gerald Jay Sussman, *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two),* MIT AI Memo 453, Cambridge, Mass., May 1978.
6. Guy Lewis Steele, Jr., and Gerald Jay Sussman, *LAMBDA: The Ultimate Imperative,* MIT AI Memo 353, Cambridge, Mass., Mar. 1976.
7. Guy Lewis Steel, Jr., "Debunking the 'Expensive Procedure Call' Myth," *Proc. ACM Nat'l Conf.,* Oct. 1977, pp. 153-162. Revised as MIT AI Memo 443, Cambridge, Mass., Oct. 1977.
8. Guy Lewis Steele, Jr., *LAMBDA: The Ultimate Declarative,* MIT AI Memo 379, Cambridge, Mass., Nov. 1976.
9. Guy Lewis Steele, Jr., *Compiler Optimization Based on Viewing LAMBDA As Rename Plus Goto,* MS thesis, MIT, Cambridge, Mass., May 1977; published as *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization),* MIT AI TR 474, May 1978.
10. Henry B. Baker, Jr., "Shallow Binding in LISP 1.5.," *Comm. ACM,* Vol. 21, No. 7, July 1978, pp. 565-569.
11. S. W. Galley and Greg Pfister, *The MDL Language,* Programming Technology Division Document SYS.11.01, Project MAC, MIT, Cambridge, Mass., Nov. 1975.
12. William R. Conrad, *Internal Representations of ECL Data Types,* Technical Report 5-75, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., Mar. 1975.
13. *650 Data Processing System Bulletin G24-5000-0,* International Business Machines Corporation, 1958.
14. Jack Holloway, Guy Lewis Steele, Jr., Gerald Jay Sussman, and Alan Bell, *The SCHEME-79 Chip,* MIT AI Memo 559, Cambridge, Mass., Jan. 1980.
15. Wilfred J. Hansen, "Compact List Representation: Definition, Garbage Collection, and System Implementation," *Comm. ACM,* Vol. 12, No. 9, Sept. 1969, pp. 499-507.
16. Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* Addison-Wesley, Reading, Mass., 1968.
17. H. Schorr and W. M. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," *Comm. ACM,* Vol. 10, No. 8, Aug. 1967, pp. 501-506.
18. John C. Reynolds, "Definitional Interpreters for Higher Order Programming Languages," *Proc. 25th ACM Nat'l Conf.,* Aug. 1972, pp. 717-740.
19. Guy Lewis Steele, Jr., "Multiprocessing Compactifying Garbage Collection," *Comm. ACM,* Vol. 18, No. 9, Sept. 1975, pp. 495-508.
20. Henry B. Baker, Jr., "List Processing in Real Time on a Serial Computer," *Comm. ACM,* Vol. 21, No. 4, Apr. 1978, pp. 280-294.
21. Skip Stritter and Nick Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," *Proc. 11th Ann. Microprogramming Workshop,* Nov. 1978, pp. 8-15.*
22. G. Frieder and J. Miller, "An Analysis of Code Density for Two Level Programmable Control of the Nanodata QM-1," *Proc. IEEE 10th Ann. Microprogramming Workshop,* Oct. 1975, pp. 26-32.*
23. A. Grasselli, "The Design of Program-Modifiable Micro-Programmed Control Units," *IRE Trans. Electronic Computers,* EC-11, No. 6, June 1962.
24. Dave Johannsen, "Bristle Blocks: A Silicon Compiler," *Proc. Caltech Conf. VLSI,* California Institute of Technology, Pasadena, Calif., Jan. 1979.
25. John Batali and Anne Hartheimer, *The Design Procedure Language Manual,* MIT AI Memo 598, Cambridge, Mass., Sept. 1980.
26. Carver A. Mead and Lynn A. Conway, *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass., 1980.
27. R. C. Daley and P. G. Neumann, "A General-Purpose File System for Secondary Storage," *AFIPS Proc. Conf.,* 1965 FJCC, Vol. 27, Part I, Nov. 1965, pp. 213-230.

28. Guy Lewis Steele, Jr., and Gerald Jay Sussman, "Storage Management in a LISP-Based Processor," *Proc. Caltech Conf. VLSI,* California Institute of Technology, Pasadena, Calif., Jan. 1979.

29. Guy Lewis Steele, Jr., and Gerald Jay Sussman, "Design of LISP-Based Processors: or, SCHEME: A Dielectric LISP; or, Finite Memories Considered Harmful; or, LAMBDA: The Ultimate Opcode," *Comm. ACM.,* Vol. 23, No. 11, Nov. 1980, pp. 629-645; also MIT AI Memo 514, Cambridge, Mass., Mar. 1979.

30. Doug Fairbairn and Jim Rowson, "ICARUS: An Interactive Integrated Circuit Layout Program," *Proc. 15th Ann. Design Automation Conf.*, June 1978, pp. 188-192.*

31. Randy Bryant, *The MOSSIM User Manual,* unpublished working paper, MIT Laboratory for Computer Science, Cambridge, Mass., 1979.

32. M. L. Minsky, *A LISP Garbage Collector Using Serial Secondary Storage,* MIT AI Memo 58 (revised), Cambridge, Mass., Dec. 1963.

33. Robert R. Fenichel and Jerome C. Yochelson, "A LISP Garbage Collector for Virtual-Memory Computer Systems," *Comm. ACM,* Vol. 12, No. 11, Nov. 1969, pp. 611-612.

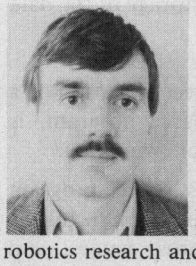
34. Daniel G. Bobrow and Ben Wegbreit, "A Model and Stack Implementation of Multiple Environments," *Comm. ACM,* Vol. 16, No. 10, Oct. 1973, pp. 591-603.

35. Guy Lewis Steele, Jr., and Gerald Jay Sussman, *The Dream of a Lifetime: A Lazy Scoping Mechanism,* MIT AI Memo 527, Cambridge, Mass., Nov. 1979.

*These proceedings available from the Order Desk, IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

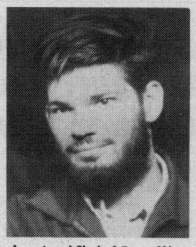
Gerald Jay Sussman is an associate professor of electrical engineering at Massachusetts Institute of Technology. His research interests are within the general area of using computers to model the intellectual processes of engineers. He is currently working on applying artificial intelligence techniques to computer-aided design of integrated systems, designing unusual architecture for the support of artificial intelligence programming, and investigating clean, modular styles in computer programming.

Sussman holds a BS and PhD in mathematics from MIT, obtained in 1968 and 1973, respectively. He is the author of the book *Computational Models of Skill Acquisition* as well as of numerous magazine and journal articles.



John T. Holloway is vice-president of technology at Symbolics, Inc., a company marketing Lisp machines, in Cambridge, Massachusetts. His research interests include special-purpose VLSI architectures for artificial intelligence applications and tools for computer-aided design.

Holloway has been associated with MIT's Artificial Intelligence Laboratory since its inception in 1965, participating in early robotics research and timesharing system development. From 1975 to 1980 he was instrumental in the design of "The Lisp Machine," a network-based scientific workstation developed as an alternative technology to timesharing. Most recently he was a principal research scientist in the Artificial Intelligence Laboratory's VLSI group; during that time he developed the Scheme-79 chip.



Guy Lewis Steele, Jr. is an assistant professor of computer science at Carnegie-Mellon University. His current research interests include programming language design and implementation, VLSI design, and computer architectures. Steele received his AB degree in applied mathematics from Harvard in 1975 and his SM and PhD degrees in computer science from MIT in 1977 and 1980. While at MIT, he worked within the Artificial Intelligence Laboratory. Steele is a member of ACM.

Alan Bell is a member of the VLSI systems research staff at the Xerox Palo Alto Research Center. His initial activities when he joined PARC in 1978 included being the primary architect and designer of the MPC VLSI silicon implementation system. His current activities center around VLSI design and the creation of an integrated wafer-scale framework in which many diverse, independently designed VLSI systems can communicate and interact with each other. Bell earned his BS in computer science at the University of California at Irvine.