S-1 Common Lisp Implementation

Rodney A. Brooks
Massachusetts Institute of Technology
Richard P. Gabriel
Stanford University and Lawrence Livermore National Laboratory
Guy L. Steele Jr.
Carnegie-Mellon University

We are developing a Lisp implementation for the Lawrence Livermore National Laboratory S-1 Mark IIA computer. The dialect of Lisp is an extension of COMMON Lisp[Steele 1982], a descendant of MacLisp [Moon 1974] and Lisp Machine Lisp [Weinreb 1981]).

1. Language Overview

Properties of this particular Lisp dialect which distinguish it from other dialects of Lisp:

— Functions are full-fledged data objects of the language. One may pass around a function (compiled or interpreted) rather than a symbol whose expr or subr property is the function.

— This dialect supports lexically scoped variables as well as dynamically scoped variables. Because it is permissible for a function to return as its value another function, sometimes environment structures must be heap-allocated rather than stack-allocated. See [Sussman 1975], [Steele 1976a], [Steele 1978].

— The language has "tail-recursive" semantics: recursive procedures of a certain form have iterative behavior. See [Burstall 1977], [Hewitt 1977], [Steele 1976a], [Steele 1976b], [Steele 1977]. For example, the following procedure behaves iteratively (it will not produce stack overflow for large N, for example):

```
(defun expt1 (x n a)    ;Compute $ax^n$
   (cond ((zerop n) a)  ;by repeated squaring
         ((oddp n)
          (expt1 (* x x)
                 (floor n 2) (* a x)))
         (t (expt1 (* x x)
            (floor n 2) a) )))
```

— Functions can return more than one value, or zero values. (In most cases the caller will discard all values but the first, or treat zero values as the null value, but special constructs are provided to call a function and retrieve all the values.)

— A rich set of numerical data types is provided, including integers of indefinite size, rational numbers, floating-point numbers of several precisions, and complex numbers.

— The language supports multiprocessing in that it is implemented on a multiprocessor computer. One result is that S-1 Lisp is deep bound.

2. Target Architecture

The S-1 architecture [Correll 1979] has some unusual features (as well as some ordinary ones):

— Words are 36 bits, quarter-word addressable (bytes are 9 bits, not 8).

— Virtual addresses are 31 bits plus a five-bit tag. Nine of the 32 possible tags have special meaning to the architecture (to implement MULTICS-like ring protection, among other things); the others may be used freely as user data-type tags.

— Most arithmetic instructions for binary operations are “2-1/2 address”. The three operands to ADD may be in three distinct places, provided that one of them is one of the two special registers named RTA and RTB. If the destination and one source are identical, then both addresses may be general memory locations (as in the PDP-11). As an example, these patterns are permissible for the “subtract” instruction (M1 and M2 are arbitrary memory or register addresses):

```
SUB M1,M2              ;M1 := M1 - M2
SUB RTA,M1,M2          ;RTA := M1 - M2
SUB RTB,M1,M2          ;RTB := M1 - M2
SUB M1,RTA,M2          ;M1 := RTA - M2
SUBV M1,M2             ;M1 := M2 - M1
SUBV M1,RTA,M2         ;M1 := M2 - RTA
```

— A variant of IEEE proposed standard floating-point is provided, including special “overflow”, “underflow”, and “undefined” values.

— There are sixteen rounding modes for floating-point operations and for integer division (thus FLOOR, CEIL, TRUNC, ROUND, MOD, and REMAINDER are all primitive instructions).

— There are single instructions for complex arithmetic, SIN, COS, EXP, LOG, SQRT, ATAN, and so on.

— There are vector processing instructions to perform component- wise arithmetic, vector dot product, matrix transposition, convolution, Fast Fourier Transform, and string processing.

— The standard configuration is a multiprocessor; synchronization instructions are available to the user. (This are in turn made available to the Lisp user. Moreover, the run-time system, and especially the garbage collector, has been written with multiprocessing in mind.)

3. Data Types

In S-1 Lisp the type of a data object is encoded primarily in the pointer to that data object, and secondarily in the storage associated with the object, if any. It is important that all pointers to a data object have a type field consistent with that object.

The format of a pointer in S-1 Lisp is a single-word with a type in bits ⟨0:4⟩ and an address or immediate datum in bits ⟨5:35⟩. This is consistent with the S-1 Mark IIA pointer format.

The data types are defined as follows:

Type	Use within S-1 Lisp
0	Positive fixnum
1	Unused
2	Unbound marker
3	Self-relative
4	Program Counter
5	Program Counter
6	Program Counter
7	Program Counter
8	GC Pointer (used only by the garbage collector)
9	reserved
10	Named vector (user data structure)
11	Named array (user data structure)
12	Halfword (immediate) floating-point number
13	Singleword floating-point number
14	Doubleword floating-point number
15	Tetraword floating-point number
16	Halfword complex floating-point number
17	Singleword complex floating-point number
18	Doubleword complex floating-point number
19	Tetraword complex floating-point number
20	Extended number
21	Code pointer
22	Procedure or stack group
23	Array
24	General vector
25	Specialized vector
26	String (vector of string-characters)
27	Character
28	Symbol
29	Pair (cons cell)
30	Empty list
31	Negative fixnum

Subtypes of vectors and arrays are encoded in the header word of the object. The subtype of a vector depends on the types of its components; these subtypes are pointer, bits, and numeric data types Quarterword integers, Halfword integers, Singleword integers, Doubleword integers, Signed byte integers, Unsigned byte integers, Halfword floating-point, Singleword floating-point, Doubleword floating-point, Tetraword floating-point, Halfword complex floating-point, Singleword complex floating-point, Doubleword complex floating-point, Tetraword complex floating-point, Halfword complex integer, and Singleword complex integer

The subtypes of arrays are the same as vectors.

Numbers are represented internally in a variety of ways in order to conserve space and time for common situations. In particular, integers in the range $[-2^{31}, 2^{31}-1]$, as well as half-word floating-point numbers, are represented in an "immediate" format, so that one need not allocate heap storage when such numbers are generated. Numbers are also represented in fixed-precision multiple-word formats, and in indefinite-precision formats. The details are given below.

Numbers may be divided into scalars and complex numbers. Complex numbers are represented as pairs of scalars in one format or another. Scalars may be divided into integers, ratios, and floating-point numbers. These three classes are further subdivided by internal storage types. All classes of numbers have provision for a representation with indefinitely large precision.

Data type 0 is used for positive integers, and data type 31 for negative integers. This implies that an immediate integer is in fact in true two's-complement form and can be utilized directly by arithmetic instructions. (The results of such instructions must, however, be range-checked; they cannot in general be assumed to be in this format.)

Larger integers are represented as extended numbers in a bit-vector format to be determined by multiple-precision instruction set.

Ratios are represented in extended number format.

Floating-point numbers come in five representations: halfword, singleword, doubleword, tetraword, and indefinite precision. The first is an immediate data type; the half-word floating-point value is stored in the low eighteen bits of the pointer. In the singleword, doubleword, and tetraword representations, the pointer simply points to a singleword, doubleword, or tetraword containing the hardware data format. Indefinite-precision floating-point numbers are represented in extended-number format.

Similarly complex numbers come in five formats. Those whose components are halfword floating-point numbers are represented as singlewords, those whose components are singleword floating-point numbers are represented as doublewords, those whose components are doubleword floating-point numbers are represented as four consecutive words, and those whose components are tetraword floating-point numbers are represented as eight consecutive words. In each case the representation consists of the real part followed by the imaginary part in standard floating-point format; the pointer points to the first of the words. A general complex number is a kind of extended number.

Numbers of fixed precision are represented in a vector-like format. There is no subtype field for numbers of fixed precision.

Extended numbers are vector-like objects, with the subtype field distinguishing the kinds of numbers. The vectors contain markable pointers. The following extended numbers are supported:

Bignum: This is an indefinite-precision integer. The data area contains a single pointer to a bits vector.

Bigfloat: Large floating-point numbers are represented as a set of integers; the precise format remains to be worked out

General complex: The data area contains two pointers, the real and imaginary parts of the complex number. Each part may be any kind of scalar number.

Ratio: The data area contains two pointers, the numerator and denominator of the ratio. Each pointer may be any kind of integer; the denominator, however, must be strictly positive.

4. Procedures

Procedures are the only kind of callable object. Any object that can be in the function cell of a symbol must be a procedure. Macros, fexprs, stack groups, and closures are all kinds of procedures.

Procedures are represented as two vectors, one being a (markable) vector of type “procedure” and the other a (non-markable) vector of type “code”. The first vector is the procedure object itself, and contains a pointer to the second vector, as well as the “local environment” (if any), pointer constants used by the code, a name, GC information (relevant to determining the stack frame format), and debugging information. The code itself is in the second vector. The pointer vector must be in S-1 Data space, and the code must be in Instruction space (in fact it is in an Instruction/Data space).

There are various ways to call a procedure. Every procedure has a “primary” calling mode, which the compiler of callers will choose as the result of declarations. However, any procedure can support all modes of entry; a patch-up routine is called if necessary.

The calling modes are elements of the Cartesian product of these sets:

$$\{\text{Normal, Numeric}\} \times \{\text{Single-value, Multiple-value}\}$$

A normal call is expected to return its result(s) as pointers; a numeric call is expected to return machine-format numbers for its numerical results. A single-valued procedure returns one value; a multiple-valued procedure may return any number of values.

When a procedure is called the caller will have loaded into a specific register a descriptor, or signature, for the kind of procedure it expected to call. This contains the number of arguments provided in the address part, and in the tag field some bits stating whether the calling mode is normal or numeric and whether a single or multiple value result is expected.

A normal, single-valued procedure call returns its value on the stack in the slot pushed for it by the caller.

A normal, multiple-valued procedure returns its values on the stack in order, last one on top, the first being in the slot pushed by the caller. A specific register will contain the negative of the number of returned values.

A multiple-valued numeric procedure returns its numeric values in one of two ways. There are 8 registers set aside for the purpose, and if the number of numeric values returned will fit there that is where they are put, with non-numeric values on the stack, and a result signature in a specific register. If there are too many numeric values to fit in provided registers, they are returned as pointers.

The return-value signature is a word encoded in such a way as to describe the types of the returned values in a precise manner. For every return-value word there are three bits in the signature word.

5. Garbage Collection

S-1 Lisp uses a modified copying garbage collection strategy [Baker 1978]. There are three types of storage for Lisp objects: dynamic, static, and read-only. Read-only storage contains objects which, once allocated and initialized, are never relocated, reclaimed, or altered. Static storage contains objects which, once allocated, are never relocated or reclaimed; however, the contents of a static object may be altered. Dynamic storage contains objects which may be relocated, reclaimed, and altered at will. Dynamic and read-only storage are divided into two independent areas for data and code objects. Only code vectors are stored in the code areas, and everything else in the data areas.

Each type of storage is a linked list of (variable sized) segments provided by the operating system. A segment consists of some power of two of segmentitos consecutive in the address space. A segmentito is 2^{16} bytes long, and thus the address space is spanned by 32,768 segmentitos. A single table in static space with one word per segmentito provides all the information necessary to quickly identify the space into which a pointer is directed, and the status of that space, along with the actual linked list.

All three kinds of storage may be expanded as necessary. However, dynamic storage may also be shrunk, by reclaiming and discarding unreachable objects. This is done by a transitive copying process. During garbage collection there are two dynamic storage areas: the old one and a newly created one. All reachable objects are traced, and objects in the old dynamic space which are found to be reachable are copied to the new dynamic space. All pointers to such copied objects are updated to reflect the new locations. When the copying is complete, the entire old dynamic area is discarded.

Read-only objects may not contain pointers to dynamic objects, because it might be necessary to alter the read-only object to reflect the new location of the dynamic object when copied. When the read-only area is periodically "frozen", a garbage-collection-like process is performed wherein any dynamic objects pointed to by read-only objects are copied to the static area. (That part of the read-only area not yet frozen is treated as if it were static.)

The choice of a copying, compacting garbage collector puts a number of constraints on the design of storage layout of data objects and has implications for the amount of run-time error checking necessary to ensure correctness of the garbage collection process.

Since copied data objects must be scanned in a sweep through the new dynamic space it must be possible to identify the data type of an object not only from a pointer to it, but also by examining it sequentially in storage. The approach used is to assume every word of storage is a pointer unless proved otherwise. The decision strategy must be completely reliable to ensure correct behavior of the garbage collector. CONS cells are thus simply stored as two consecutive pointers. By convention the first is the CAR and the second the CDR. A vector of pointers is simply stored as a header word which is a fixnum length followed by the pointers. The garbage collector does not need to distinguish such an object from consecutive CONS cells as the fixnum can be identified as such from its own tag field, and the remaining pointers must be fowarded anyway. Unmarkable objects (objects composed of other objects that are not pointers and which, therefore, can display arbitrary bit patterns) must be headed by a word with a distinguishable tag: the GC tag. The address part of this word can perhaps be used for other header information, such as vector type for unmarkable vectors. For floating point numbers, for instance, there is no other direct use for the header word, and its is simply a single word of overhead for every floating point number.

Another requirement is that it must be possible to tell when an object has been forwarded from old space. Thus there must be a distinguishable way of writing a forwarding pointer into an object. Again the GC tag is used. For markable objects (such as CONS cells or markable vectors) the first word of storage can be used. For unmarkable objects (e.g. floating point numbers) the first word of storage can legitimately contain any pattern of bits so the forwarding pointer must be stored in a header. In most cases objects have an extra header word (see above) used to identify an object when scanned during the garbage collection which caused its instantiation. This same word can be used during the next garbage collection as a location for the forwarding pointer. To make the scheme work it is necessary that the scanning algorithm overwrite the GC tags used for unmarkable object identification as soon as it scans them, so that the subsequent garbage collection does not think that the object has already been forwarded. This extra step could be saved if two distinguishable tags were allocated for garbage collection purposes, rather than the single one used.

Bounds checking during storage of a vector element is necessary to ensure that a pointer does not clobber valuable header information in another data object. Similarly RPLACD must check on the data type of the object it is operating upon (for instance, RPLACD of a single word floating point number will clobber the following storage location). For most data types a spurious RPLACA will at most lead to an invisible pointer. However procedure headers have their pointers offset by -39 words to maximize short addressing mode accessibility of their literals and so RPLACA must also run with error checks at all times. Similar considerations apply to FSET and SETQ.

Bounds checking during vector element reference is necessary to ensure that some non-pointer object (e.g. a word of a floating point number) does not find its way into a storage location which is markable. This could lead to incorrect fowarding during garbage collection providing an inconsistent environment and subsequent degradation due to confused operation of the garbage collector.

The vector storage layout has been designed so that the necessary bounds checking can be carried out in a single S-1 instruction. This required a slightly increased overhead for strings, but was virtually free in all other vector-like objects.

Of course, if enough declarations are made the compiler should often be able to deduce the correctness of the code it is compiling, and so refrain from generating error checking code.

Stacks are allocated in separate segments, rather than as vectors in dynamic data segments. This is so that the garbage collector can easily recognize stack allocated objects in order to avoid relocating them twice.

Code vectors are copied and compacted in instruction spaces in the same manner as other objects in data spaces. All PC's on stacks must therefore be updated to provide the correct return addresses when processing resumes. The tag fields of the architecture provide an easy mechanism for detecting PC's, but the structure of the stack frames has to be carefully controlled so that it is possible to identify the code vector referred to by the address part of the PC in order to find its forwarding pointer.

6. Synchronization

S-1 Lisp allows multiple processes to run concurrently in a single address space, possibly using multiple S-1 processors. The following conditions and conventions are required to ensure the integrity of the Lisp data structures:

> If a block I/O transfer has been started and an address given to the operating system for it, then no garbage collection may take place until the transfer has been completed. This is because the region of storage involved in the transfer must not be relocated out from under the operating system.
>
> Only one process may allocate storage at a time. This is to ensure that the same storage doesn't get allocated twice.
>
> No process which refers to pointers into dynamic space may run when a garbage collection is in progress, as there will be many invalid pointers around.
>
> At no time may a process introduce an invalid pointer into a markable data area.
>
> At no time should any process which is not in the act of allocating storage expect to use storage which has not had a valid pointer pointing to it since originally allocated.

These requirements are met with the aid of the values of three symbols: *allocate-lock*, *gc-lock*, and *block-io-status*. The quiescent state values of these variables are −1, −1, and 0, respectively. (These symbols are pointed to by the systemic quantities vector. They all reside in static space.) Processes such as storage allocation, garbage collection, and initiators of block I/O transfers coordinate their activities by the values of these three symbols.

A process which wishes to allocate storage must wait till *allocate-lock* has a value of −1 and then indivisibly set it to be non-negative before it can allocate storage. If it finds there is not enough free storage and for some reason it is not permitted to ask the operating system for more (there is no more, or more likely the user has set a limit on working set size), it must initiate a garbage collection. The only garbage collection possible while *allocate-lock* is non-negative is one initiated by the process which set the lock. When a storage allocator has a valid pointer to the new object in a markable place, it releases its lock by setting *allocate-lock* to −1.

If a garbage collection must be initiated for some other purpose (for example, by the purification process for freezing read-only storage), it must likewise first gain control of the *allocate-lock*.

If any process, while waiting to gain control of the *allocate-lock*, notices a garbage collection in progress, it may decide to not bother to initiate another garbage collection at all.

In any case, when a garbage collection is initiated, it sets *gc-lock* to zero. Notice that there can be only one process with control of the *allocate-lock* and so there is no synchronization problem with setting *gc-lock*. The process then must wait until *block-io-status* is 0 and indivisibly set it to −1. Next it must interrupt or stop all the other processes. Then the copying compacting garbage collection can proceed. On completion it resets *block-io-status* to 0, then resets *gc-lock* to −1, then resumes any stopped processes, then returns to the invoker.

The value of the symbol *block-io-status* is used to count the number of block I/O transfers in progress. When a process wishes to initiate a block transfer it must wait until *gc-lock* is −1. Thus *gc-lock* prevents new transfers from being initiated. The initiating process must then wait until *block-io-status* is non-negative, and indivisibly increment it. On completion of the data transfer *block-io-status* must be indivisibly decremented. (The reason for the second wait on *block-io-status* is that a garbage-collection process may have gotten in after the I/O process checked *gc-lock,* made it non-negative, then gotten to *block-io-status* before the I/O process, and set it to −1 and commenced a garbage collection. Without the second check the I/O process might initiate a new data transfer before getting interrupted or stopped by the garbage collector.)

7. Systemic Quantities Vector

To speed up various operations a vector of commonly referred to constants and procedures is pointed to by a register dedicated to the purpose. This vector contains the quantities T, (), the locks mentioned above, constants defining the sizes some objects, and the addresses of the CONSers, and addresses of routines which allocate de-allocate, and search special lookup blocks on the stack. Calling these routines is inexpensive compared to a normal procedure call.

8. Conclusions

S-1 Lisp supports a number of non-standard programming constructs, such as multiple values and functional data types, is both lexically and dynamically scoped, and is multiprocessed.

The multiprocessing requirements and the choice of a copying compacting garbage collector have put strong constraints on the implementation design. Since the target machine is a time shared machine with no dynamiclly writeable microstore it is not possible to resort to microcoding to overcome many of the problems, as it is for Lisps on personal machines (e.g. [Weinreb 1981]). The tagged address architecture of the S-1 proves to be its saving grace enabling efficient implementation of the necessary runtime checks within the standard macro-machine architecture.

9. References

[**Baker 1978**] Baker, Henry G., *List Processing in Real Time on a Serial Computer*, Communications of the ACM, 21(4):280-293, April 1978

[**Burstall 1977**] Burstall, R.M., and Darlington, John. *A Transformation System for Developing Recursive Programs* in **Journal of the ACM** 24(1):44-67 January, 1977.

[**Correll 1979**] Correll, Steven. *S-1 Uniprocessor Architecture (SMA-4)* in **The S-1 Project 1979 Annual Report**, Chapter 4. Lawrence Livermore National Laboratory, Livermore, California, 1979.

[**Hewitt 1977**] Hewitt, Carl. *Viewing Control Structures as Patterns of Passing Messages* in **Artificial Intelligence** 8(3):323-364, June, 1977.

[**Moon 1974**] Moon, David. **MacLisp Reference Manual, Revision 0**, M.I.T. Project MAC, Cambridge, Massachusetts, April 1974.

[**Steele 1976a**] Steele, Guy Lewis Jr. *LAMBDA: The Ultimate Declarative*, AI Memo 379, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, November, 1976.

[**Steele 1976b**] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *LAMBDA: The Ultimate Imperative*, AI Memo 353, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, March, 1976.

[**Steele 1977**] Steele, Guy Lewis Jr. *Debunking the 'Expensive Procedure Call' Myth; or, Procedure Call Implementations Considered Harmful; or, LAMBDA: The Ultimate GOTO* in **Proceedings of the ACM National Conference**, pages 153-162. Association for Computing Machinery, Seattle, October, 1977.

[**Steele 1978**] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP*, AI Memo 452, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, January, 1978.

[**Steele 1982**] Steele, Guy Lewis Jr. et. al. *An Overview of Common Lisp*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Sussman 1975**] Sussman, Gerald Jay, and Steele, Guy Lewis Jr. *SCHEME: An Interpreter for Extended Lambda Calculus*, Technical Report 349, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, December, 1975.

[**Weinreb 1981**] Weinreb, Daniel, and Moon, David. **LISP Machine Manual**, Fourth Edition. Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, July 1981.