CS 113

THE IMPACT OF STORAGE MANAGEMENT ON PLEX PROCESSING LANGUAGE IMPLEMENTATION

BY

WILFRED J. HANSEN

TECHNICAL REPORT NO. CS 113
JULY 1969

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

THE IMPACT OF STORAGE MANAGEMENT ON PLEX PROCESSING LANGUAGE IMPLEMENTATION

Wilfred J. Hansen*

STANFORD GRAPHICS PROJECT
Professor William F. Miller,
Principal Investigator

July 1969

Computer Science Department
Stanford University

'*Present address: Applied Mathematics Division, Argonne National Laboratory

This work was supported in part by National Science Foundation Grant GP-7615.

Abstract

A plex processing system is implemented within a set of environments whose relationships are vital to the system's time/space efficiency:

- Data Environment
  - Stack Structures
  - Data Structures
- Subroutine Environment
  - Routine Linkage
  - Variable Binding
- --. Storage Management Rnvironment
  - Memory Organization for Allocation
  - Storage Control

This paper discusses these environments and their relationships in detail. For each environment there is some discussion of alternative implementation techniques, the dependence of the implementation on the hardware, and the dependence of the environment on the language design. In particular, two language features are shown to affect substantially the environment design: variable length plexes and 'release' of active plexes. Storage management is complicated by the requirement for variable length plexes, but they can substantially reduce memory requirements. If inactive plexes are released, a garbage collector can be avoided; but considerable tedious programming may be required to maintain the status of each plex.

Many plex processing systems store numbers in strange formats and compile arithmetic operations as subroutine calls, thus handicapping the computer on the only operations it does well. Careful coordination of the

system environments can permit direct numeric computation, that is, a single instruction for each arithmetic operation. This paper considers with each environment, the requirements for direct numeric computation.

To explore the techniques discussed, a collection of environments called Swym was implemented. This system permits variable length plexes and compact lists. The latter is a list representation requiring less space than chained lists because pointers to the elements are stored in consecutive words. In Swym, a list can be partly compact and partly chained. The garbage collector converts chained lists into compact lists when possible. Swym has careful provision for direct numeric computation, but no compiler has been built. To illustrate Swym, an interpreter was implemented for a small language similar to LISP 1.5. Details of Swym and the language are in a series of appendices.

TABLE OF CONTENTS

APPENDICES

PREFACE

Plex processing is an effective technique for attacking graphical problems. The Stanford Graphics Project conducted project Swym to examine current techniques and develop new techniques. An important result is that plex processing cannot be viewed as simply another high-level language facility. Instead, it must be viewed as having an impact on the most vital components of a language implementation. Introduction of plex processing into a language has far-reaching repercussions in the design of implementations of that language.

Many graphics projects have based their implementations on plex processing. An early effort was Sutherland's Sketchpad thesis reported in [Suth 63] and [John 63]. More recent are [vDam 67] and the interactive display project at General Motors [Joyc 67]. A review of several systems implementations useful for graphics is [Gray 67].

This paper can be considered as an outline for a course entitled 'Semantics of Plex Processing Languages.' Knowledge of Fortran and assembly language would be prerequisite and the course would cover six languages in detail: ALGOL [R&R 64] - the arithmetic mother, LISP [MCar 62] - the plex father, and their offspring - ALGØLW [BBG 68], GEDANKEN [Reyn 69], PL/I [IBM 68b], and Swym/STUTTER (this paper and appendices). As far as possible, the course should ignore the syntax of the languages since there exists a superabundance of literature on that field. Instead the course should cover the fundamental semantics of data structures and program control.

The author would have preferred to continue making additions to Swym rather than write it up. There came a point, however, where the goals of the project had been met and further effort would not add useful information. This paper, especially the appendices, represents a system in an arrested state of development. This is not because there are conceptual difficulties in

making STUTTER a practical programming system, but rather because there do not appear to be any such difficulties. Swym serves its purpose: it is a framework within which systems can be implemented.

The body of the paper is an abstract discussion of language implementation and storage management. The appendices give complete details of the Swym system, while the bibliography indicates previous work in implementing storage management. Unfortunately, some of the papers referenced, especially in section 111.2, describe programming languages with no description of the implementation details being discussed in this paper. In such cases, the implementation details have been ferreted out in private communication. Bibliographic references are in the form,

[name yr]

where name is four reasonably mnemonic characters from the author's name and yr is the year the work was published. If the information was a private communication, the year is coded 'pc'.

The author is indebted to all those who have taken their time to explain and discuss the intricacies of various plex processing implementations, notably the 'system didlers' at the Stanford Artificial Intelligence Laboratory and the Computer Based Laboratory. Thanks are due to Dr. J. Reynolds, creator of CØGENT, for discussion of that system and language implementation in general. F.L. Morris acted as an invaluable sounding board for descriptions of the evolving Swym system. A special debt is owed my adviser, Dr. William Miller, for his advice and encouragement.

INTRODUCTION

The term "plex" may have been first proposed in [Ross 61]. D.T. Ross invented the term to mean a structure composed of 'n-component elements' just as a binary tree is composed of 2-component elements. It has become more common, though, to use the term plex to mean 'n-component element' and to call a structure of these a 'plex structure.' One main characteristic of plex processing is the pointer - a data item that encodes the location of some other data item. Most commonly, a pointer is the address of a plex in memory. In short:

plex - one or more data fields of computer memory, usually consecutive.

pointer data coding the location of other data (usually a pointer is the address of a plex.)

plex structure - a group of plexes connected in the sense that starting from one or more of the plexes, all other plexes can be reached by means of pointers, either directly or through a sequence of pointers.

plex process - a program using plexes to represent a substantial amount of its data. (An almost equivalent and more determinate definition is: any program that requires storage management beyond a stack.)

A list is an important special case of a plex structure. Basically, it is an ordered set of plexes. Normally a list is realized with 2-plexes in this way: the first component of each 2-plex points at an element of the ordered set of plexes; the second component points at the next 2-plex in the list. Usually, the second component of the last 2-plex points at some standard list terminator. Lists were treated mathematically by John McCarthy [MCar 60] and implemented in the plex processing language LISP 1.5 [MCar 62]. [Knth 68] includes a complete discussion of plex data structure implementation.

Other good reviews of the literature on plex implementations are in [Schr 67] and [Lang 68]. The most promising work is reported in [Ross 67], [Hawk 67], and [Styg 67]. The last two are part of the ambitious SDC LISP 2 for the 360, described in the SDC TM - 3417 series.

When plexes are created and destroyed during execution of the program, some storage management technique must keep track of the occupied and unoccupied memory. Some storage management schemes require a garbage collector. This is a routine that processes all memory, identifies the occupied and unoccupied areas of memory, and makes the latter available for reallocation. Although this is a time consuming process, other storage management techniques may involve extensive bookkeeping.

Satisfactory computer languages must also provide numerical computation. In plex systems numbers must be distinguished from pointers. Often this means that numerical operators must retrieve their arguments from plex structure; and this sometimes requires several memory accesses and one or more shifts. Since plex languages usually permit more than one type of number, the operators must also test the types of the arguments. But lengthy access sequences and type-testing can seriously slow down a numeric calculation. Solving this problem requires some form of compilation process and a declaration structure in the language. The compiler can then determine at compile time the types of operators and compile the appropriate machine instructions. The problem of directly accessing numbers that is, direct numeric computation - requires that the stack and memory be permitted to contain arbitrary bit pattern numbers. This means, for example, that a garbage collector cannot assume that all words on the stack are pointers; nor can it distinguish pointers from other information on the basis of a bit in the word.

Swym and STUTTER

To examine plex processing from the practical level, Swym - a general plex processing memory management system - was implemented. As an illustration of the capabilities of this system, an interpreter for a small LISP-like language called STUTTER was also implemented.

The central focus of the Swym project was a particular plex structure called a compact list. This form of list can reduce memory requirements by up to half; essentially compact lists do not always require the second pointer in the 2-plex for lists. The details of compact lists are in the section on Swym data structures (1.2) and in the Appendices.

The compact list was derived from and suited for the needs of LISP 1.5. Consequently, STUTTER is similar to that language and has the same basic operations, (though new names, the LISP 1.5 names are in parenthesis):

<u>fst</u>	(CAR)	argument must be a list; <u>fst</u> returns the <u>f</u>irs<u>t</u> element of that list;
<u>rst</u>	(CDR)	argument must be a list; <u>rst</u> returns the <u>r</u>es<u>t</u> of that list after the first element; if the list has only one element, <u>rst</u> returns an atom;
<u>tak2</u>	(CONS)	there must be two arguments, both pointers; tak2 <u>tak</u>es <u>2</u> words from free storage and <u>ta</u>c<u>k</u>s the <u>2</u> arguments together so first is <u>fst</u> of result and second is the <u>rst</u>;
<u>atom</u>	(ATOM)	predicate - <u>true</u> if argument is an <u>atom</u>, <u>false</u> otherwise;
<u>eq</u>	(EQ)	predicate - <u>true</u> if both arguments point at the same plex; <u>false</u> otherwise;

rplf (RPLACA) there must be two arguments and the first must be a list; the first pointer in that list is replaced with a pointer at the second argument.

Unlike many LISP implementations, an interrupt results if fst or rst is taken of an atom. Like LISP, the mnemonics ffst, frst, rfrrst, etc., can be defined (lending credibility to the name STUTTER). As indicated above, tak2 always makes a 2-plex. STUTTER relies on the Swym garbage collector to make compact lists where possible.

Super-parentheses are an important feature of the STUTTER input syntax. Represented by the characters '<' and '>', a pair of super-parentheses can be substituted for any pair of normal parentheses (of which there are many in LISP and STUTTER input). When the input routine finds the right super-parenthesis (>) matching a left super-parenthesis (<), the enclosed ordinary parentheses are forced to balance, either by creating right parentheses or by ignoring characters. If characters are added or deleted, an error message is printed.

Swym has been carefully designed to permit direct numeric computation. Special care was taken in several areas: the stack and free storage permit thirty-two bit numbers, and the value of a STUTTER atom is directly accessible, given the address of the atom. The subroutine linkage mechanism and the storage management techniques also take into account the possible presence of numbers.

Swym was programmed for an IBM 360 under OS/360. This was not only because of the wide availability of the 360, but also because it was something of a challenge to adapt the 360 for efficient plex processing. The Stanford 360 is a model 67 with 32 bit addressing and paging facilities. Swym was designed to test these facilities on a plex processing system, but the operating system did not support them and moreover, Swym was moved to SLAC. Nonetheless, the lessons learned from Swym may have important implications for machine design, as is discussed in the conclusion. Details of Swym and STUTTER are in the Appendices.

Plex Processing Language Implementation

Several interesting languages have been designed primarily for plex processing. The *best known examples are LISP [MCar 62], SNØBØL [Farb 64], L^6 [Know 66], and the earlier IPL-V [Newl 64] and COMIT [Yngv 62]. An excellent review of such languages is in [Bobr 68]. The promise shown by these languages has led to many attempts to define and implement plex processing facilities for existing high-level languages, For instance: SLIP [Weiz 63], records for Algol [Hoar 66, Wrth 66], and the 'based variable' feature in PL/1 [IBM 68b]. Unfortunately, adding a plex processing feature is very unlike adding a new function (say SINE) or even a whole new arithmetic (say complex). Plex processing not only requires appropriate additions to the compiler or interpreter, but can also require extensive revision of the code compiled for all other features. The major problem is that plex processing requires some form of storage management, either by the user, or by the system. This paper surveys the problems encountered if a system is to manage storage. These problems are encountered in the very basic areas of data representation, subroutine linkage, and storage management itself.

In most computer installations, program compilation is a frequent event. Like other non-numeric computation, compilers can make advantageous use of plex processes. For this reason, the concepts and techniques discussed in this paper apply not only to the code generated to implement the features of a language, but also to the features required in the compiler itself. This paper assumes that the language being implemented includes plex processing and consequently requires storage management. It is also assumed that the language permits definition of subroutines (procedures) and that programs written in the language will make substantial use of subroutines and modularity. For two reasons, Swym sheds some light upon the functions required during the execution of a plex processing program. First, Swym is an investigation of plex processing; second -- and less obvious -- Swym required construction of plex processes. The garbage collector, input/output routines and the STUTTER interpreter are all examples of plex processes.

A programming system will be used by many programs over an extended period of time. It is important in the design of such a system to avoid decisions that will slow execution substantially, expecially when a practical alternative is available. Usually many decisions must be based on the trade-off between memory space and execution speed. Before multiprogramming and timesharing the answer was to optimize by saving time at the expense of space since the memory was there. In modern systems there is an expense not only for execution time, but also for memory space. The ratio between these two expenses is critical to the choice of an efficient set of alternatives for a language implementation. One of the goals of this paper is to point out the alternatives. A major effort was made to reduce the size of the data structures as far as possible and to reduce the time and space required for the most basic system functions.

One approach to the definition of execution efficiency is that of the I? systems [Know 66]. That language and system is designed for 'low-levelness'. This has been defined [Mnch pc] as producing code that is no more than ten percent slower than equivalent hand code. STUTTER was designed with a slightly different criteria in mind: the principle of 'relative difficulty of specification.' This principle declares that a language facility should take proportionately as much effort to specify as it does to execute. In this way the programmer can have some feel for how much time the program will take simply from the amount of code he writes.

Several problems contribute to slow running of high-level languages with plex processing facilities. Most of these, however, are inherent, not in the plex processing facilities, but in the implementations. Many plex processing users see only the interpretive LISP or SNØBØL systems. Compiled LISP, however, runs much faster than when interpreted. SNØBØL IV has plexes, and should run faster than SNØBØL III (because string matching can now be avoided in plex operations). While interpreters have their place, they are simply too slow to be used on any problem big enough to justify the use of a computer. But there exist plex processing systems that meet these problems adequately. The ALGOLW [BBG 68] system at Stanford implements plexes, yet is so fast a total system that student programs can be compiled and executed on a 360 in less than a second. In short, the presence of storage management facilities need not automatically mean slow execution.

Although written in terms of language implementation, this paper is really directed toward any program that can be more efficiently implemented by first implementing some tools. These tools might be any one of,

a) write a few macros

b) write macros to interface with an existing memory management system like Swym

c) design a special purpose language

d) design a full general purpose language

The author believes that the most useful approach is probably (b), and he would probably design many more data-specific macros than might another programmer.

Environments of a System Implementation

A program is executed on a computer in a set of environments including not only the hardware, but also service routines and conventions for data representation and program linkage. The environments most directly affected by the requirement for plex processing can be divided into:

- Data Environment
  - Stack Structures
  - Data Structures
- Subroutine Environment
  - Routine Linkages
  - Variable Binding
- Storage Management Environment
  - Memory Organization for Allocation
  - Storage Control

All of these environments interact with the system storage management facility. Not only must they be designed to make storage management possible, but many require plexes for their own implementation.

The relations between the environments must be carefully worked out before system construction is begun. A hasty decision on one environment can be expensive in the implementation of some other. ALGOLW did not provide for marking pointers on the stack. This eventually required that the garbage collector be rewritten. [Baur pc]. Other decisions in ALGOLW require that a 2-plex occupy sixteen bytes. But if a set of environments is well coordinated, more than one language can be implemented within that set of environments. This provides for very efficient linkage between routines written in two or more languages.

Each section below describes one environment of a language implementation. The discussion will center around the effect of the storage management scheme on that environment but will also cover alternative implementations and the relationship of the environment both to the language being implemented and to the machine being used. Each section concludes with a discussion of the relevant features of Swym and STUTTER. This serves for comparison and to illustrate one choice of solutions for the problems posed.

I. Data Environment

Data structures range in complexity from the single bit to organizations covering large quantities of direct access storage. To a certain extent, the data structures in a system are dictated by the needs of the higher level language. But the physical structure of the data may differ from the logical structure manipulated by the higher language programmer. In any case, the data requires storage space and this must be provided by some form of memory management mechanism, either during compilation or during execution. The discussion below separates stack data structures from other data structures for two reasons. First, the stack is the simplest form of execution time memory management. Second, a stack is usually included in a system for program control purposes. In most languages routines exit in the reverse order of entry, so the stack is the natural analog of the progress of the program.

I.1 The Stack

A stack (sometimes called a push down list) is a simple but important system component. Among the advantages of a stack are that few instructions are required to allocate and release space and there is no possibility of fragmentation of space, because there is only one contiguous area of unused space. A stack permits recursive procedures: by allocating temporary variables and saving return addresses on the stack, a procedure can call itself directly or indirectly. Each invocation refers to the correct variables and returns control correctly. Even if there is no recursion in an entire program, a stack is a flexible and efficient method of storage allocation.

There are three basic operations on a stack: addition, deletion, and reference to items; all are-easily implemented. One pointer to the stack is maintained; additions and deletions move the pointer, while items are referenced relative to it. Sometimes a test is made for the bottom of the stack when items are deleted. Other systems assume that the program is correct and that no more deletes will be-executed than additions. Several methods have been implemented for ensuring that the stack does not grow beyond its bounds. The most common is to simply test the stack pointer against a pointer to the end of the stack. A possible hardware method is to check the low order _k_ bits of the stack pointer; if all are zero, the stack is exhausted. This method means that stacks must end on certain boundaries; a restriction that complicates memory allocation. With the PDP-6 hardware stack commands, a stack pointer includes a count that is decremented when the stack increases and incremented when items are deleted. If the count reaches zero, the stack is exhausted.

Stack exhaustion poses peculiar problems; one simple solution is to terminate execution. In paging systems or systems with more than one stack, it may be possible to continue. The difficulty is that the stack is changing most rapidly near the top. If a new page is allocated for the stack, only one or two words may be used before the stack goes back to the old page. If the new page is released, it may need to be reallocated again very shortly. If the new page remains part of the stack, the stack may grow large during one-portion of a program and eat up valuable space during later portions. At the least, paging algorithms must recognize that the bottom of the stack will not be accessed for a reasonably long time while the top of the stack must never be paged out.

When a computer implements a stack in the hardware, it is common to keep the top stack items in faster access memory. The B-5500 had two high speed stac'k locations; the Atlas had sixteen. In these cases, special logic can be incorporated to minimize memory accesses due to fluctuation of the stack pointer. When an item is deleted from the top of the stack, the hardware must decide whether or not to initiate a memory fetch to load the next item of the stack. The answer dependson the expected ordering and frequency of additions and deletions.

In most Algol implementations, a block of temporary storage on the stack is allocated at procedure entry and deleted upon exit. The stack fluctuates more rapidly for B-5500 and Euler-like [Wrth65] implementations: the top elements of the stack are the implied operands for an operation and the result replaces those operands on the stack. Swym permits an in-between method; stack storage is allocated only when it is needed, not necessarily for the duration of the routine.

In plex processing systems three classes of items can be stored on the stack: pointers, return addresses, and non-relocatable data. These must be distinguished because the garbage collector must find all structures referenced by pointers on the stack. It is possible to associate type bits with every word on the stack to identify those that are pointers. But if those bits are in the word itself, it will not be possible to store arbitrary words on the stack as is required for direct numeric computation. (A number might have the pointer bit set wrong.) Numbers could be treated by creating a plex containing the numeric value and storing a pointer to that plex on the stack. But this seriously slows numeric computation by unnecessarily invoking the storage management facilities. LISP 2 proposes that each routine call include a 'stack map' of the storage allocated for the calling routine. This map could be accessed relative to the return address, which would also be on the stack.

Swym Stack

The Swym stack is one 360 word wide and grows downward. That is, additions are made at the lowest addressed end of the stack. In this way, the latest entries to the stack can be addressed relative to the stack pointer. Provision has been made for three varieties of entry on the stack: pointers, return addresses, and stack plexes. The high and low order bits of the word are used to distinguish between these varieties so that the garbage collector can treat each correctly. Every plex has a one-word plexhead specifying its length and type. Numbers and other arbitrary bit pattern words may only be stored in plexes; but note that a compiler can take the plexhead into account and generate code to directly reference numbers stored on the stack.

I.2 Data Structures

Data structures that have been implemented include:

Class I.	bits, words, arrays, strings, stacks, queues, and connection matrices.
Class II.	Lists, plexes, rings, and hash-coded associative structures.
Class III.	Variants of the above for tapes, cards, direct access devices, and transmission.

All classes are alike in that they require memory space to store information. If this space is allocated during execution, there must be some form of execution-time storage management.. Section III of this paper concentrates primarily on management for Class II.

The elements of Class I are simple in that they do not necessarily involve pointers, although they may involve dynamic storage allocation. The data structures in Class I are well covered by [Knth 67]. Stacks have been discussed in Section 1.1. Queues are simply push-through (FIFØ or first-in-first-out) stacks. A connection matrix represents a graph by having one bit for each possible connection between the nodes. If the bit is one, that connection exists. Ordinarily arrays are used to contain information concerning the nodes connected by the matrix.

The data structures in Class II generally involve pointers. These structures are described in [Schr 67] and [Gray 67]. It is interesting to compare LISP lists with connection matrices for describing networks. If there are n nodes, the connection matrix requires n^2 bits. If there are p connections and each list element requires b bits, then the list structure requires $\underline{pb}$ bits. The density (number of connections/number possible connections) of the graph for which the two representations take the same number of bits is $\underline{p'}/\underline{n}^2$ where $\underline{p'b} = \underline{n}^2$. For greater densities, the matrix requires fewer bits than the list. The breakeven density is then $1/\underline{b}$. For $\underline{b} = 64$, the break even density is 1.5%. That is, if more than that percentage of the possible paths exist, then the connection matrix is a smaller representation. Connected graphs under 66 nodes always exceed 1.5% density because there are at least $\underline{n}-1$ paths. The trouble with matrices is that their allocation is very machine dependent. For example, an increase from less than 32 nodes to more than 32 nodes might mean substantial reprogramming.

Two strange schemes have been proposed for LISP list structures, but not implemented. In one, CØNS would hash its arguments and store the dotted pair in a hash bucket. If the pair was already in the bucket, a pointer to the existing pair would be returned. This scheme would make EQ and EQUAL the same simple operation, but would prohibit the efficiencies possible with RPLACA and RPLACD. The major bar to implementation (the IBM 44X was proposed) seemed to be the lack of a suitable garbage collection algorithm. The second scheme was the n-cube addressing scheme. Every word would have associated with it 2^n-1 other words. These can then be addressed with just n bits in the pointer field. (It was proposed that the addresses of the words associated with word x be formed from the address of x by modifying each bit in turn. Thus the associated words would be those connected to x along the edges of the n dimensional hypercube.) In this scheme, though, any function that will build a plex must tell its arguments where to put their result; the consequences are staggering: in general, the computation must terminate before any results are stored.

The CORAL system [Suth 66] is one example of a system based on rings. Essentially, each ring is a list with an explicit ring head; the end of the list points back to the head. In addition, alternate elements of the list contain pointers to the ring head and the reverse pointers that point back to the preceeding reverse pointer. A ring element is a plex, called a block. The pointers constituting the ring are physically stored in these plexes and the beginning of the plex is marked with a word with a special bit pattern (all ones). CORAL is a set of macro statement for the TX computers at Lincoln Laboratories.

Other ring systems are described in [Gray 67]. [Perl 60] describes 'threaded lists'; these are similar to rings but derived from LISP lists. The end of the list is marked by a special bit, and the pointer there points back to the beginning of the list.

An elegant notation for plex processing in higher level languages is the 'record' feature described in [Hoar 66] and [Wrth 66]. Essentially, the declaration of a 'record class' defines a possible type of plex. The class name is implicitly declared as a procedure for generating members of the class. Identifiers attached to the fields of the plex are implicitly declared as procedures to access the contents of records of the class. The arguments to such procedures are records of the proper class. Other identifiers can be declared to be pointers to members of one or more record classes.

Before direct access devices and on-line systems, Class III structures were usually sequential files. But modern Class III structures have been forced to include elaborate indexing and addressing structures. Indeed, there is need for space management in most systems with Class III structures. The most comprehensive existing system for managing file storage is OS/360. Its great flexibility has prompted user grumbles about having to specify too many parameters. For example, one of the facilities offered is a relocating garbage collector for disk packs. This collector is not called automatically, but must be invoked by a special procedure.

One goal in on-line systems is to build a filing system capable of maintaining any file of data. An experimental unified file system was reported in [Frnk 66]. This system encoded the value of each data item as a pointer into a table of possible values for the item. Variable length pointers appear to be necessary to make the scheme work; and even then it seems to entail substantial I/O. Another, more analytic approach to file design is discussed in [Benr 67].

Some systems have used Class III data structures for graphic applications. The MULTILANG file system is the basis for the PENCIL system reported in [vDam 67]. Plexes are stored on a disk and contain keys and elements. A plex may be specified by specifying logical combinations of keys. The LEAP system [Rovn 67b] stores 'triples' of associative information. Each triple is stored three times on the disk; once for each of the components. Thus triples can be retrieved based on any part of their contents.

Several factors must be taken into account when designing a data structure for a language implementation. These include the host computer, the basic operations to be implemented, and the amount of data description that must be available to general purpose run-time routines.

The host computer affects data structure design at the lowest levels. For example, the size of pointer fields depends on the amount of free storage to be addressed. Also, most computers favor certain portions of words by having instructions for manipulating those portions. A physical structure design should take advantage of such natural access aids. The danger in such designs is that a 'cleverness' in some portion of a representation will not save as much space and/or time as is required to get the information into the peculiar form required. In keeping with the principle of relative difficulty of specification, the physical structure should bear some resemblance to the logical structure. For example, variable length plexes could be represented physically as a list of fixed length plexes. But the programmer may reference the last item in the plex frequently, expecting it to be found with address arithmetic, rather than list searching. Numbers should be stored so as to be accessible for the hardware arithmetic operations; i. e., on the appropriate storage boundaries so shifting is avoided.

A large proportion of the time in a plex process is spent accessing the correct piece of data. Since data access can mean descending through many levels of (logical) data structure under control of the program, the best measure of the efficiency of data access is the effort to descend one level in the data structure. In Swym, these 'descent' operations are rst and fst; requiring five and one instruction executions, respectively. Access to a fixed length element of a Swym plex requires one instruction. The 7090 implementation of Lisp required 8 instructions each for CAR and CDR, the only available descent operations. Lisp implementations using temporary storage [Bobr 67] [Cohn 67] typically must test page tables and perform address arithmetic to descend one level in the data structure. Such processing is time consuming and has led to the definition of hardware 'paging' systems like that on the 360/67.

There are several reasons why data structure designs often include descriptive information along with the data. A primary reason is that the garbage collector must determine certain properties of structures before it can collect them. Other reasons might be that each operator checks its argument to see that it is the correct type, or that the operators must know the specifications of the data in order to completely specify the operation. For example, a general print routine must know the type of the data and a string move routine must know the length of the string. The garbage collector needs the location and length of each active data item and the position(s) of any relocatable information in the item.

A data item can be described by its location, length, type, and zero or more type dependent parameters. This information may be specified explicitly or implicitly and may be located with the item, with the reference, or remotely. Information stored with the item usually takes the form of explicit fields referenced relative to the pointer at the item. Storing descriptive information with a reference to an item means that the item can be a part of some other item. The XPL string mechanism [MKee pc] permits two strings to share memory. Remote storage of descriptors has been proposed by D. McLaren [MCla pc]. Plex storage would be allocated from the bottom of a free storage area, while fixed length descriptors were placed in the top. The descriptor corresponding to a pointer could be found by a binary search on the descriptor area. Presumably, the descriptor would be infrequently referenced in that system. Implicit data description is information derived from other characteristics of a data item. For example, the length may be implicit in the type, that is, all items of that type are the same length. The type may be implicit in the fact that the item is within some area of memory. J. Reynolds [Reyn pc] has proposed a minimal encoding scheme having type explicit and implicit with the reference. If the compiler determines (from declarations or by analysis) that a certain field can only point at a plex of one of _n_ types, then the type information can be coded with the reference and requires only $\lceil \log_2 n \rceil$ bits.

Swym Data Structures

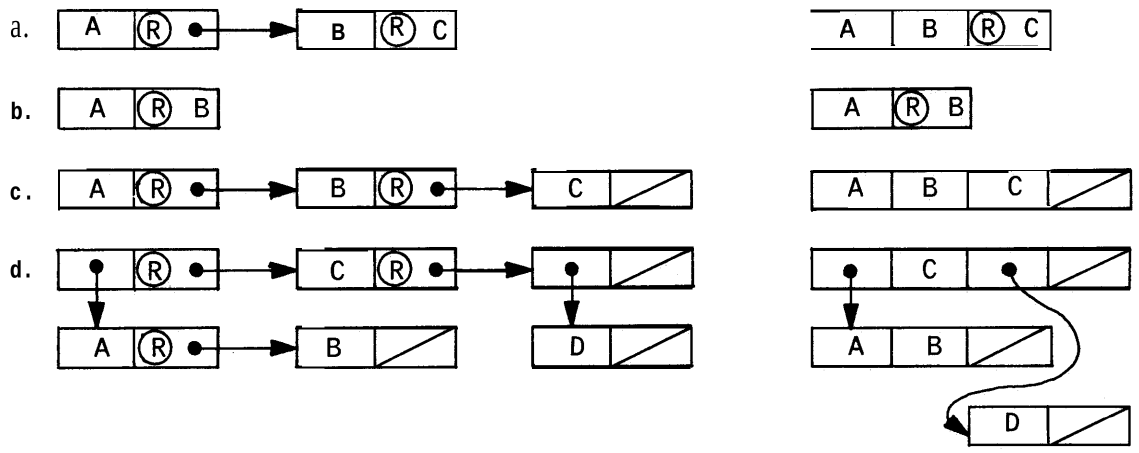
Very complex plexes can be realized under Swym, but this section considers only those implemented for the STUTTER interpreter: lists and atoms. A _list_ is a sequence of pointers. Each pointer is the address of an element of the list. An element, in turn, can be either a list or an atom. An _atom_ is a plex with arbitrary internal structure. Note that Swym lists are special plex structures because the garbage collector can compact them.

The difference between conventional lists representations and compact lists parallels the difference between the IBM 650 and most other computers. 650 instructions had two address fields: one for the operand and one for the next instruction. Most other computers save memory by assuming that the instructions are sequential. When the instruction sequence is broken a 'branch' instruction continues execution elsewhere.

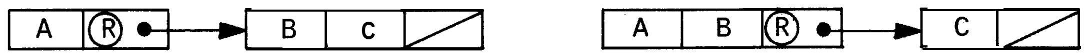
Like the 650, many list representations use two pointers for each element of a list: one to the element and one to the rest of the list. On the other hand, list storage can be conserved by storing lists sequentially in memory; then only the pointers at the elements are required. But if that is the only way lists can be stored, certain list operations can be time consuming. The Swym solution is to allow a 'list branch' pointer. Lists are normally sequential, but when a list cannot be sequential, it is continued with a 'list branch' pointer. Figure I.1 illustrates several list structures in both the old and new representations. Note that a 'list branch' pointer is called a rst pointer because it points to the rest of the list.

An earlier system permitting compact lists intermixed with chained lists has been reported by N. Wiseman [Wise 66]. This system provides for creation of compact lists, but the garbage collector does not rearrange storage to remove rst pointers. Unlike Swym, variables may point at rst pointers and there may be more than one rst pointer between element pointers. But the user must program extra checking to avoid treating rst pointers as list pointers. Wiseman presents no data on the effectiveness of his system.

Swym list words have the format shown in Figure I.2a. If the rst bit is zero, the word points at an element of the list. If the rst bit is one, this pointer is so-called 'list branch' pointer; it points not at an element of the list, but at the continuation of the list. The atom bit is on in a pointer at an atom; this is the distinguishing characteristic of an atom in the Swym system. If both the atom and rst bits are zero, the pointer points at a sublist of the given list. If both the atom and rst bits are one, the end of the list has been reached. A list ending with a pointer at the atom NIL is a normal list; otherwise, it is what LISP 1.5 sometimes calls a general s-expression. The atom NIL is treated as a list with no elements.



All Possible Mixed Representations of C:



1. A pointer at an atom is represented by a character string. (The 'print name' of the atom.)

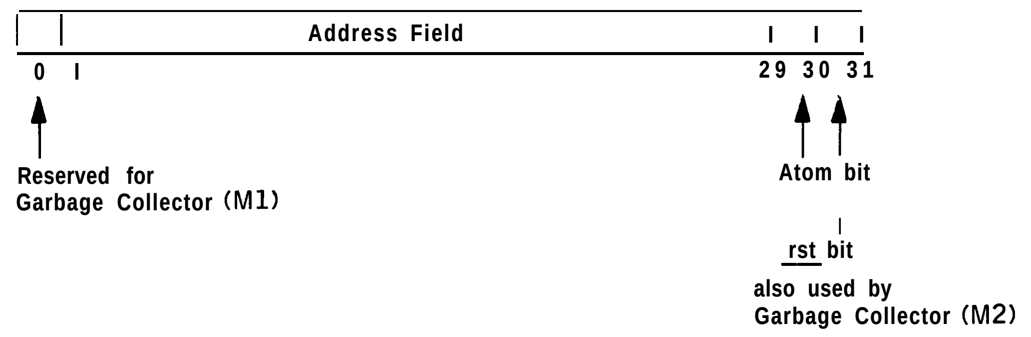
2. A 'list branch' pointer is indicated by Ⓡ (for <u>rst</u>).

3. $\boxed{Ⓡ\ \mathbf{NIL}}$ is written $\boxed{\diagup}$ to indicate the end of a normal list.

4. Any other <u>rst</u> pointer at an atom is the end of a 'general s-expression' list.

FIGURE 1.1

a. List Word



b. P lexhead

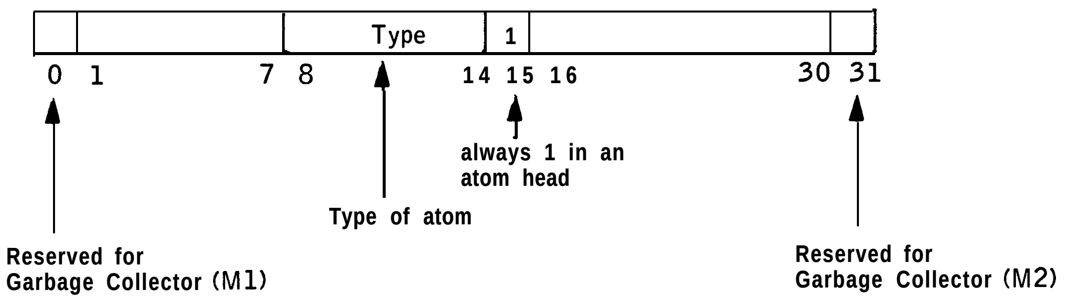


FIGURE 1.2

Associated with each atom is a plexhead - a word containing the type of the atom and two marking bits for the garbage collector. The format of a plexhead is shown in Figure I.2b. The twenty-two unused bits may be used for different purposes for different atom types. Depending on what is desired, a plexhead may be located almost anywhere with respect to any other words in the atom, but usually it is the first word in a plex.

Atoms are addressed by pointing six bytes in front of the first byte of their plexhead. This means that they point at a half word boundary which is <u>not</u> a full word boundary. A pointer at a list always points at a full word boundary. Thus, Swym distinguishes a list from an atom by the pointer pointing at the item (the atom bit is just part of the address). Because atoms are addressed six (not two) bytes in front, the <u>rst</u> operator examines a bit in the middle of the plexhead. Since this particular bit is always on, <u>rst</u> causes a specification error. <u>fst</u> also causes a specification error if applied to an atom. But the components of an atom can easily be referred to with special Swym macros that assemble only one instruction. From a paged memory standpoint, the atom bit has a small advantage: whether or not an element is an atom can be decided without accessing that element. The advantages of the atom bit suggest its use even in a 24-bit address machine.

All atom types are alike in having a plexhead and in being addressed in a strange manner. Only two atom types are defined in the basic Swytn system: symbols and strings. But the user may define other types of atoms simply by coding the primitives to create, manipulate, and garbage collect the new atom types. Since the contents of a plex can be addressed directly if the address of the plex is known, operations on plexes are no more costly than operations on statically allocated storage.

The symbol atom corresponds to the normal Lisp atom. In Swym, such an atom has three components: the plexhead, a value cell, and a property list. The plexhead contains control bits describing the contents of the value cell and the atom's definition as a function. The value cell contains the atom's variable binding as discussed in Section 11.2. The property list is similar to that for LISP 1.5, but the r. ..rst is a pointer to the print name (a string atom).

There are currently three sub-types to the string atoms. All are alike in containing no relocatable information (addresses) and in being stored in a consecutive block following the plexhead. The three sub-types are string, fixed point number, and hexadecimal number. The major difference between these subtypes is in how the print routine handles them; they are not distinguished by the garbage collector. The plexhead of a string atom contains the subtype field and a length field. The string and hexadecimal number may be any number of bytes up to 32767. A fixed point number currently always has a length of four bytes.

Swym free storage is one contiguous block, and new plex structure is created from one end of that block. This storage allocation scheme has proven advantageous in the Cogent system [Reyn 65]. Lists can be created in compact form if all their elements are known. Atoms of any size can easily be created; for example, bit string atoms are always stored in consecutive bytes. Note that the garbage collector requires only two bits in the plexhead; all other words in an atom structure may be full words.

Thirty-two bit addressing is supported by Swym. A pointer may occupy the full word except for three bits: the two low order bits and the high order bit (bits 0, 30, 31). Because the 360 addresses bytes and all Swym pointers point at words, the low order two bits of a pointer are not used for addressing. The high order bit cannot be used either. Difficulties will arise as soon as address arithmetic (especially BXLE and BXH) is attempted on full thirty-two bit addresses; addresses in the upper half of memory are negative and are thus algebraically smaller than zero. Swym uses the three circumscribed bits to good advantage. The low order bit is the rst bit, and it marks a rst pointer. The next to low order bit (bit 30) marks a pointer at an atom. Both the high and low order bits are used for marking by the garbage collector. These same bits have other meanings in control words on the stack.

II. Modular Programming and the Subroutine Environment

Plex processing implies a structured approach to data; the corresponding structured approach to programming is modularity. If a large program is broken down into a series of smaller programs, the latter are easier to write, debug, and modify. Moreover, if the program is carefully divided along functional lines, the large program can often be redesigned simply by rearranging the sub-programs. Modularity is evidenced at many levels. There is always a set of basic operations available to the programmer, and usually there is a mechanism for defining and invoking subroutines. Basic operators can range from machine instructions, to interpreter 'syllables', to sets of macro instructions. Each specifies a set of operations considered by the designer to be convenient and comprehensive for describing the steps of a task. A subroutine mechanism permits the programmer to design his own set of basic operations tailored to the task at hand. While implementing Swym, it was necessary both to modularize the system itself and to provide efficient and convient mechanisms for modularity in languages implemented under Swym.

The most basic example of modularity is the hardware instruction set of the computer. Each instruction is a modular description of a sequence of gating registers onto buses and operating on those buses. On the 360, -yet another level of basic operations called the micro-instructions is introduced between the programmed instructions and the hardware manipulation. W. McKeeman has pointed out [MKee 67] that computer designers must consider the problems of language design in order to optimize computer functions. His work, however, usually emphasizes the design of computers for specific languages. The discussion in this paper attempts to isolate basic operations common to all languages that provide plex facilities.

Most LISP 1.5 implementations provide an interpreter to execute list structure read by the same read routine that reads S-expression data. This provides a simple way to begin building a LISP system. In fact, most LISP compilers are written in LISP and compiled interpretively. The availability of an interpreter also permits treating programs as data and then executing the processed program. The LISP interpreter can be described in LISP itself, a feature that can lead to better understanding of the language. But the most common reason for providing an interpreter is really the design of special purpose computers. By coding an interpreter, the programmer provides a set of operation suitable to implementing the language. Interpreters often have syllabic operation structures like B-5500 machine language. Such code structures provide high code density - thus saving space - because the operands are implied to be the top of the stack and thus need not be addressed explicitly. The only commercial computer specifically designed for implementing languages by making highly efficient interpreters is the B-8502, tantalizing details of which are beginning to leak out. How well suited the B-8502 is to variable length plex processing remains to be seen.

For Swym, a pseudo-machine was implemented by writing a set of macros for the 360 assembler. The facilities offered by this pseudo-machine include those desirable for plex process implementation - both data manipulation and program control. Macros are suitable for designing pseudo-machines because it is not necessary to design a whole machine. Just as much as is desired can be formalized, while other processing is done in terms of hardware operations. In this sense, macros provide more freedom than the interpreted micro operator approach to pseudo-machines.

For a variety of reasons, plex processing programs tend to include many subroutine calls.* Probably the primary reason is that programmers who think in terms of structured data tend to think in terms of structured programs. At the same time, the fact that the data may have similar structure at different levels seems to lead not only to subroutines, but even to recursive subroutines. For instance, in a graphical problem a routine to find all connected nodes might easily be written by finding all nodes adjacent to a given node and then applying the same subroutine to each of these adjacent nodes. Another important reason for subroutines and modularity in plex processing programs is that such programs are usually experimental and subject to change (because non-experimental programs usually cannot afford the overhead currently implicit in many plex processing systems). Since subroutine-call is often the most frequent operation in plex processing programs, attention must be paid to its optimization. This problem is considered at length below.

There are several goals and advantages in modular programming. These are synonymous, because meeting the goals successfully implies taking full advantage of the potential saving in time and effort (in total time, not just initial program writing time). Modularization offers:

(1) Ease of writing. It is very convenient to code an operation by writing the name of a routine or macro that will perform that operation. Not only is total writing reduced, but repetitive writing is eliminated; both reduce the chance of clerical error.

---

*7090 LISP even compiled subroutine calls for CAR and CDR. Even now most LISP implementation compile arithmetic operations as subroutine calls. ALGOLW demonstrates that with a suitable declaration structure, such basic operation can indeed by compiled in-line. Swym has provided the mechanisms necessary for compiling such in-line code while maintaining communication between compiled and interpreted functions.

(2) Ease of modification. Since clearly defined modules perform specific functions, changes in these functions can be made simply by changing the appropriate module. Modules often provide good 'hooks' for adding debugging output or statistics gathering routines. The modularity built into the Swym system was of use on more than one occasion. The subroutine calling conventions were changed several times. The code in all routines was changed by modifying the macros and reassembling. It was also simple to change register usage to communicate better with oS and PL360. The flexibility demanded by the Swym programming standards should prove invaluable in implementing other languages within Swym.

(3) Ease of debugging. Modules are easily tested independently, so that errors can be isolated. LISP is especially amenable to modular debugging for two reasons. First, all data is represented in S-expressions, so the inputs and outputs of a routine can be represented without driving routines. Second, LISP facilitates and even encourages subroutine organization so that less thought is required to put the program into modular form.

Some system design time should be specifically devoted to breaking the 'system into program modules. Likewise, some program design time should be specifically devoted to breaking the program into appropriate subroutine modules-. Likewise, some subroutine design time should . . . .
Time so spent will be returned with interest in the coding and debugging phases and will probably be returned many times over during modification of the program. In designing Swym, subroutine modularization was not difficult because several LISP implementations demonstrate not only a good system modularization, but also the basic operation that should be provided to the programmer. Nonetheless several guidelines were discovered.

An important guideline for modularization is to restrict each module to a single definable function. This function need not be very basic, but its definition should be consistent with the single definable function of all other modules. Consistency means that the set of modules implementing a higher level module should have mutually exclusive functions, and those functions should be directed toward accomplishing the function of the higher level module. Thus a data accessing module could be defined to also update a counter or set a bit, but only if in the encompassing module the counter or bit was always associated with that data access operation. On the other hand, operations should be divorced if they only occur together accidentally. If "accidental neighbors" are combined in a single module, sooner or later they will be needed separately. It is better to err in the direction of too much separation since change is such a common feature of programs. One compromise is to introduce another modular level. A macro (for instance) could be defined to call two accidental neighbors, leaving the two as separate modules.

Another important guideline in the construction of modular systems, is to provide for transparency. A completely transparent subroutine can be called at any point in a routine with no resulting change in the output of the routine. For example, the LISP PRINT routine prints its argument, but does not modify any location in memory. Ordinarily, a routine will not be completely transparent, but will affect one or more variables in the calling routine, or will produce output (doing both might also satisfy the well-definedness guideline); but the quantities modified by a routine should be implicit in its well defined single function. One example of transparency is the block structure limitation of the scope of variables in ALGOL 60.

A pre-coded routine can be included in any program and will not create conflicts with existing identifiers in that program (the same is not true of most assemblers). A good example of the need for transparent code is in the definition of debugging packages to be executed when required in the program.

Since routines must preserve the state of the computer system in order to be transparent, the system must make this a convenient operation. Some systems facilitate state preservation by automatic stacking, or at least provide other ready access to the system variables. Other systems do not even provide the capability to determine parts of the current state of the system. Satterthwaite has a discussion of coding transparent routines under OS/360 [Satt pc]. Swym attempts to provide the facilities necessary for writing transparent routines; the stack can be used for storing arbitrary information. Also, the 'internal variable' convention [Reyn65] has been adopted for accessing and controlling the state of system variables (for example STIVQMØ and STIVCCH control the READ routine, see Appendix C.).

Two system components are vital to modular programming: routine linkage and variable binding. The efficiency of these operations dictate the level of modularity permissible. The PL/I macro facility is necessary not only for compile time computation, but also to provide modularization that would not be -practical using the cumbersome PL/I procedure invocation mechanism (involving two subroutine calls for storage management). Routine linkage and variable binding are each discussed in detail below. There is a two fold relation between these system components and the storage management mechanism: (1) they require storage for control information; (2) if there is a garbage collector, they must identify pointers and distinguish them from non-pointer information.

II.1 Routine Linkage

The code required to call a subroutine and return is critical to system efficiency. The speed of any individual routine is far less critical because it is executed less frequently than subroutine linkage; the latter is required between all subroutines. Routine linkage includes several functions:

- save return address and status
- locate and execute subroutine
- restore status and continue at return address.

The primary interaction between routine linkage and storage management is control of the space for the status information. This information can include control bits and register contents. It also includes current variable bindings, but this is considered in the next section. The space management must be coordinated with the storage management required for data plex operations. In particular any pointers that are saved must be available to the garbage collector.

Ordinarily, status information can be saved on a stack because routines exit in exactly the reverse of the order in which they are entered. But some languages like Gedanken [Reyn 69] permit labels as values of variables. A routine may store a local label in a global variable; after the routine exits, it may be reentered in the middle by a branch to that global variable. Not only must the routine be entered, but the status must be restored to the status existing when the label was stored in the variable. Thus, for Gedanken, status information (and variable binding) must be stored in plexes just as data. Storage for both can be managed with the same plex mechanisms.

Labels can introduce problems even in Algol implementations. Algol permits a routine to branch to a label in an outer block (this label may even be specified as an argument to the current routine). If status information is stored on the stack and includes the stack pointer for the dynamically enclosing block, then a goto to an outer block must interpretively unwind the stack to find the correct status for the outer block, That is, the goto must keep restoring the stack pointer until the storage for the correct block is found. This problem can be solved for Algol with the DISPLAY mechanism mentioned below. In EULER [Wrth 65], though, all operators take their operands from the stack and replace them with a value. This means that the DISPLAY mechanism is much more cumbersome for the goto problem.

The implementation computer can influence routine linkage. The PDP-6 has a single instruction to store the return address on a stack and pass control to a routine; The routine can branch to the return address and delete it from the stack with a complimentary instruction. The 360, on the other hand, has no stack instructions and requires provision for the addressability of the calling and called routine.

Two common techniques should be avoided in designing routine linkages. 1) Routine linkage should be in-line rather than a call on a service routine. The latter technique effectively doubles the number of routine linkages. Also, service routines often waste time retrieving linkage parameters from a para--meter list, while parameters can be implicit in in-line code. 2) Not all registers should be saved on entry to a routine. The time expenditure is small but the storage expense is large. Although it is possible for the called routine to save and restore only those registers it destroys, the calling routine usually has an even smaller number of active registers. Moreover, the calling routine has the information needed to mark each register as pointer or not-pointer.

It is not necessary for the called routine to return to the instruction immediately following the call. In the 360, a call might be:

```
          L         15, Address of called routine
          BALR      14, 15
CØNTINUE  DS        0  H
```

The called routine exits with BR 14. But other information may be included between the BALR and CONTINUE with little extra cost. The called routine would simply have to return with B n(14), where n depends on the amount of included information. This information can be used for several purposes. CØGENT conditional execution is based on the FAILURE mechanism. The failure return point is an extra branch instruction in the calling sequence, executed by the called routine if it fails. The calling sequence could also include information to facilitate debugging. Pointers to the name of the routine and the values of its variables could be referenced by information in the calling routine. This is also the place for the pointer to the stack map discussed in Section 1.1.

<u>Swym -- Routine Linkage</u>

In Swym, three instructions are required to call a routine, three more are used for routine entry, and three are used for routine exit. These instructions provide for:

(1) establishing addressability for the called routine
(2) branching to the new routine
(3) marking the return address so it won't be garbage collected
(4) storing the return address in the stack (two instructions)
(5) recovering the return address from the stack (two instructions)
(6) returning to the calling program
(7) re-establishing addressability for the calling routine

One register (B) is designated as the base register for all routines. Before branching to the routine, this register is loaded from a 'transfer vector.' This area is always addressable (via register S) and contains the entry point addresses for all routines. Thus establishing or re-establishing addressability requires one load instruction. Space is saved because only one address constant is required for the address of each routine.

Strict conventions govern saving and restoring the eight registers available for general use. (Eight is enough if BXLE and BXH are avoided,) If-an assembled routine wants a register saved it must save it itself or be certain that the called routine preserves that register. In the latter case, a comment in the called routine must describe the calling routines and registers which must be left intact. Compiled functions must save the active registers when calling another function.

Swym provides some debugging information with no extra storage in the call. The return address is the stack makes it possible to find the BCD name of the calling routine. The BCD name is assembled just before the entry point to a routine. The entry point can be found because the instruction at the return address refers to the location in the transfer vector table of the entry point address.

A Gedanken interpreter was designed to run under Swym. The label variables mean that an interpreter like the LISP EVAL cannot use a stack because the status at any point might have to be restored. Consequently, the designed interpreter used plexes to contain status information and return addresses for the interpreter. A second type of plex contained status information for routines being interpreted. The latter also contained variable bindings.

II.2 Variable Binding

To refer to items of data, a routine has *variables*. Usually, each variable is named with an *identifier* (a character string). But two identifiers may refer to the same variable (Fortran EQUIVALENCE) and one identifier may refer to different variables in different routines, so an identifier is not the same as a variable. The binding of a variable at a given time is the value that variable would have if it were referenced and the information changed if a value were assigned to the variable. Along with more complicated data structures and program control, higher-level languages have introduced more complicated relations between variables and their values. Variable binding affects the garbage collector both because most variable binding schemes require memory and because the garbage collector must find all active structures that are pointed at by variables. This section will cover three topics: types of variables, types of bindings, and the special problems introduced with LISP global variables.

Types of Variables:

The variables of a routine may be *local*, *argument*, or *global*. A variable is local to a routine if it is declared in that routine. Space is allocated on entry to the routine and the routine uses the local to hold a value. A compiler can usually compile straightforward code to access a local variable.

An argument to a routine also establishes a local variable, but the value and/or storage allocation may be supplied by the calling routine. Arguments are passed to routines in at least four different ways: *value*, *result*, *reference*, *name*. A value argument is treated exactly like a local variable except that it is initialized to the value of the actual parameter. A result argument is treated like an uninitialized local, except that when the routine exits the final value is assigned back to the actual parameter, which must be a variable. Value and result variables are like locals in that storage is allocated for them during execution of the routine. Reference arguments refer directly to the allocation of storage in the calling routine. If an actual parameter for a reference argument is an expression, a temporary variable is created in the calling routine and the argument refers to that created variable. Call by name arguments are evaluated each time the argument variable is referenced. Name arguments can slow execution substantially because a complex expression may be repeatedly evaluated, and because each evaluation requires reestablishment of the environment for evaluation of the name argument.

A global variable is any variable that is referenced, but not declared in a routine. It may have been either a local or an argument in the routine where it was declared, In block structure languages like Algol, a global variable must have been declared in a typographically enclosing block. The compiler must compile a reference to the variable that will be created in that outer block. Because it has no block structure, LISP global references (called free variables in LISP) are references to the nearest dynamically enclosing declaration of the same identifier. (A routine dynamically encloses all routines called during its own execution.)

In a given implementation, global variable binding may be either static or dynamic. The distinction is based on the treatment of variables during execution of functions that have been passed as values. Static binding means that variables always have their most recent binding. Dynamic global binding means that variables have the binding they had at the time the functional value was created. LISP is defined to require dynamic global variable binding. Examples of the problems involved are given below.

Types of Binding

There are four types of binding: register, static storage, stack, and free storage.

Register variable binding is often used for system functions. The arguments are placed in registers and the function is called. This technique is used even for compiled functions in PDP-6 LISP and can be used for compiled functions in other language implementations. Register binding is convenient because the calling routine usually must compute the arguments and the result is in a register. Moreover, the argument may stay in the register until a subroutine is called. Problems arise when a subroutine is called: the registers must be saved. If any sub...subroutine globally refers to a quantity bound in a register, then the reference must be not to the register, but to the location where the register is stored. Usually this is either static storage or the stack. Furthermore, if the subroutine might invoke a garbage collector, any variable that is a pointer must be stored in a location accessible to the garbage collector and must be identified as a pointer.

Register binding of variables is satisfactory for direct numeric computation (i.e. the value in the register might be a number). Suitable declarations in the called routine enable the compiler to treat the number correctly. But when the number is saved across a subroutine call, it must be identified so that it cannot be mistaken for a pointer.

If a routine is not recursive and not reentrant, space for variables can be allocated by the compiler. Such variables are statically bound, that is, their binding never changes and all references are to the same location in memory. Fortran variables are allocated in this manner. This binding technique can require excess space because storage is allocated for all variables even though several sets of routines may never call each other. (They could use the same variable storage space.) One problem with static binding is that the garbage collector must find all plexes that are pointed at from static storage. This can be handled either by allocating all pointers together or by building a list of statically allocated pointers. A second problem is that a large structure can be referenced by a single static variable and will remain active even if it is no longer needed.

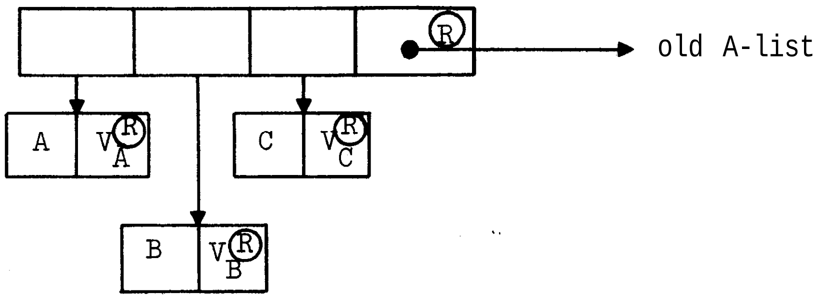
To provide for recursive and reentrant code and to ensure that variables are allocated only as long as they are needed, variables can be allocated on a stack. In Algol, all variable storage (except the controversial dynamic <u>own</u> arrays) can be allocated either statically or on a stack. When stack storage is allocated on entry to a routine, care must be taken that any variable marked as a pointer contains a valid pointer. Otherwise the garbage collector may become confused and the program may have a bug that depends on the previous contents of memory. The garbage collector does not need to determine which quantities on the stack are variables; all it needs is to determine which are pointers.

ALGØLW utilizes an elegant extension of the DISPLAY mechanism discussed in [R&R 64]. The variables for each routine are allocated on the stack when the routine is entered. One pointer to the stack is maintained (in the general registers) for each typographically enclosing block. With this technique, code can be compiled to reference any global variable directly. Moreover, the environment for an argument called by name can be established by simply loading the stack pointer registers.

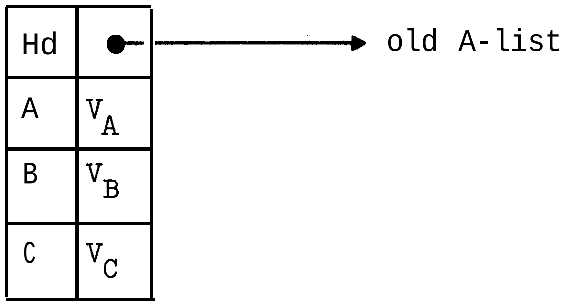
Free storage must be used for binding the variables of complex languages like LISP and Gedanken. The original reason for this in LISP was that the technique was easy to describe in the LISP formalism and easy to implement for the interpreter. However, the discussion of the global variable problem below will show that given the features of LISP, free storage variable binding cannot be avoided. Several techniques have been employed including the A-list, the APVAL, and the VALUE cell.

The A-list was used in the early LISP 1.5 implementation. It is described implicitly in the description of EVAL in [MCar 62]. Basically, each time a routine is entered a dotted pair is created for each variable; the CAR is the variable name and the CDR is the value. These dotted pairs are CONSed onto the front of the current A-list. When the interpreter must find the value of a variable, it scans the A-list looking for a pair whose CAR is the variable name. Note that this handles global variables as a straightforward extension. When a function is passed as an argument, both the expression for the function and the A-list current <u>at the time the function was passed</u> are passed. Thus, when that passed function is invoked, the old A-list is used so that global variables have their correct values. A major disadvantage of the A-list, besides search time, is the fact that it is continually allocating and releasing free storage and thus increases the frequency of garbage collection.

It is possible to improve on the structure of the A-list and still use the A-list. As suggested by John Reynolds [Reyn pc], this method would create a plex on the A-list for each function invocation. The method is best illustrated with an example. Suppose a routine binds the variables A, B, and C. The new portion of the A-list would be (with compact lists):



The new method would create this plex:



The searching procedure would be slightly more complex, but there would be a saving of space.

In a block structured language, a function can only address variables declared in itself or in statically enclosing blocks. The A-list can take advantage of this structure by pointing not at the A-list formed for the calling routine but at the A-list for the smallest statically enclosing block. This is another extension of the DISPLAY mechanism. A Swym interpreter for Gedanken [Reyn 69] was designed to take advantage of the block structure of that language.

In LISP 1.5 some frequently referenced atoms such as T and NIL are only bound at the outermost level. This would mean searching the entire A-list to get the appropriate value (*T* for T and NIL for NIL). To avoid this, Lisp 1.5 permitted the APVAL property on property lists (usually a shorter list than the A-list). If an atom had an APVAL, that was its permanent binding. Thus evaluation of variables meant searching first the property list for an APVAL and then searching the A-list.

More recent Lisp systems have extended the APVAL concept by always storing the value of an atom in a cell on the property list (under the property VALUE in PDP-6 Lisp). As the atom is rebound, the old binding is stored on a special push down list. Thus interpreted functions need only search the property list for variables. Moreover, the location of the VALUE cell never changes so the compiler can compile code refering to it directly. By reducing the number of types of binding in LISP, the VALUE cell reduces the complexity of the language. All variables are the same, whether they are declared in a PRØG or a LAMBDA or are undeclared but have been given a value external to all routines. But as discussed below, there are valid LISP programs that the VALUE cell cannot implement.

Global Variables in LISP

Global variables (LISP uses the term 'free variables') contribute both the best and worst features of LISP. The global reference scheme defined by the A-list mechanism is neat and simple, and yet very general. But the A-list is time consuming; it requires list searching time and garbage collection time. The worst features of LISP are the problems of compiling functions to interface with interpreted routines and the contortions necessary when attempting to replace the A-list.

Compiled LISP routines usually use the stack for variable binding because that is the most efficient technique. But if a variable is to be used globally in some other routine it must be accessible. LISP 1.5 provides two types of global bindings for compiled routines: SPECIAL and COMMON. A SPECIAL variable is bound to a special cell on the property list of the atom representing the variable. (PDP-6 LISP uses the VALUE cell.) This special cell never moves so code is compiled to access it directly. But if the variable must be referenced by both compiled and interpreted functions, it must be bound on the A-list. This is precisely the treatment given to any variable declared to be COMMON. But SPECIAL and COMMON are attributes of variables and all references to the same variable are treated the same. Thus if X is declared COMMON, all routines referencing X must refer to it on the A-list, even though only two or three routines use it as a global variable. Primarily this problem is a fault of the LISP syntax because there is no place for declarations in the S-expressions that are interpreted.

The most difficult problems are introduced into LISP by the provisions for allowing functions as arguments and values of routines. The difficulty is that a function is a pair consisting of a piece of code and an environment for interpretation of that code. Consider these functions:

```
MAP[a;x] = if null [x] then NIL
        else cons [a[fst[x]]; map[a;rst[x]]]
ACONS[a;x] = MAP[function[λx.cons[x;a]];x]
```

The call ACONS [NIL; (A B C)] should return ((A) (B) (C)). Note that the <u>a</u> inside the <u>function</u> must refer to the first argument of ACONS. The A-list treats this case properly because <u>function</u> returns a FUNARG. This is a list withthree elements:

(FUNARG {function S-expression] {A-list]).

When a[fst[x]] is interpreted, the A-list used is the A-list current when <u>function</u> was executed. The binding for a on this A-list is indeed the first argument to ACONS. The SPECIAL cell or VALUE cell would not work because the most recent binding for a is the value returned by <u>function</u>.

PDP-6 LISP avoids the problem in MAP by having _function_ save both the code and a pointer to the stack. When the _function_ is invoked, the stack is unwound down to that level; that is the old bindings are taken from the stack and placed in the VALUE cells of the appropriate atoms. To remember the current environment, however, as each binding is unwound the current binding is saved on the binding stack. Thus the mechanism for _function_ is very clumsy using the VALUE cell approach. This certainly violates the principle of relative difficulty of specification.

The VALUE cell mechanism does not work at all if functions are permitted to return functions as values. Consider this valid LISP function:

$$\mathrm{PLUSX}(x) = \underline{\mathrm{function}}\ [\lambda y.x+y]$$

The value of PLUSX is a _function_ containing the global variable x. This global variable must be evaluated in the environment existing when the _function_ operator was applied. Subsequently the value of,

$$[\mathrm{plusx}[3]][2]$$

should be five. In short, the variable _x_ must retain its value after PLUSX exits so that that value can be referenced by the function returned by PLUSX. (The problem of global variables in functions that return functions is carefully explained in [Weiz 68]).

There is such a wide diversity of requirements for variable binding that it seems necessary to consider a comprehensive declaration structure like PL/I. Variables can usually be bound on the stack efficiently, but other techniques must be available to handle those cases that cannot be so simply handled.

Swym Variable Binding

Swym uses many of the variable binding techniques described above and can support all them because it has variable length plexes. Arguments are passed to system functions in the general registers and remain there unless it is necessary to call a sub-routine. A few variables controlling input/output are bound in statically allocated storage. Six general registers are used for passing arguments to compiled functions; no compiled function may have more than six arguments. Swym provides a comprehensive set of macros for storing and accessing information on the stack. The standard Swym approach is to save a word on the stack when it must be saved and remove it when it is no longer needed.

The STUTTER variable binding scheme is similar to that used for PDP-6 LISP. Every symbol atom has a value cell (the word following the plexhead in memory). When the interpreter is asked to evaluate a single symbol atom, it simply returns the contents of the atom's value cell. Before entering a routine defined by an S-expression, the arguments supplied are appropriately evaluated and the values are placed in the value cells of the formal argument atoms in the LAMBDA expression. The old contents of the value cells are stored in a block on the stack. This block contains alternately the formal argument atoms and their old values. When the routine terminates, the block is removed from the stack and the old values are restored to the atom's value cells. Currently, only static global variable binding is implemented. To communicate with interpreted code, compiled code would store the required value in the value cell of the appropriate atom.

A compiler could compile code to access numeric operands directly, either in the registers or in the value cells. The values in the register could be stored on the stack in a stack-plex indicating the presence of one or more full words. A non-relocatable value can be stored in a value cell by resetting the relocatability bit in the plexhead for the atom. The cost of these features is a little additional bit testing in the interpreter and the garbage collector.

III. Storage Management Environment

Fortran is a static language; all storage can be allocated at compile time or loading time. More complex languages require more complex memory allocation mechanisms. Algol 60 has dynamic array sizes, but still its memory allocation can be handled with a stack mechanism. Plex processing routines, however, create structures that can be referenced after the routine exits. Moreover, plex processes create and delete plexes of various sizes at random times throughout the computation. The bookkeeping necessary to keep track of the allocation of memory to the different plexes is called storage management. A plex remains active as long as it can be referenced by theprogram either directly or via a series of pointers. The memory not allocated to any plex is called free memory., free pool, free storage or free plexes. An active plex cannot be deleted because that would destroy the program's data. Under some systems and high-level languages the programmer must write code to keep track of the active plexes and to free those that are no longer active. In other systems, a routine called the garbage collector traces through all active structures and returns to free storage any inactive plex. Use of a garbage collector demands disciplined use of pointers because it must be able to find all active structures and must be able to distinguish a pointer from other data items.

Storage management schemes can be classified as relocating or non-relocating. In a relocating system, a garbage collector moves all the active plexes so they occupy a contiguous area of memory and leave a contiguous free area. This process is time-consuming, but the process of allocating a plex is simple: one end is allocated from the free area. Non-relocating systems do not move the active plexes; they simply keep track of the plexes that are-free and can be allocated. In such systems the process of allocating a plex can be time-consuming because it involves a search of the free plexes to find one that is large enough. If a free plex of the required size cannot be found, a larger plex is split; part filling the request and part being returned to the free list.

Non-relocating systems risk encountering the fragmentation problem. If a request is made for a plex larger than any free plex but smaller than the total of all free plexes, then core is said to be fragmented. When this occurs, a system may

(1) terminate execution

(2) relocate all active storage as an emergency procedure

(3) call-a user routine to free any little-needed plexes.

Since (3) is highly problem dependent, its use can only be considered in special situations. Some research seems to indicate that the probability of fragmentation is low enough to justify solution (1). The argument is that if fragmentation occurs, then all of storage will soon be exhausted anyway. The compromise approach (2) above is often suggested, but this combines the disadvantages of both relocating and non-relocating systems merely to guarantee that the system will fill memory before terminating. The extensive bookkeeping for the non-relocating system is required, as well as the disciplined use of pointers for the relocating system. D. Knuth [Knth 67] has collected numerous storage management techniques and analyzed many. His emphasis is on non-relocating systems that terminate when fragmentation occurs. The current paper concentrates on relocation schemes, both because the non-relocating are covered by Knuth and because the Swym garbage collector is a relocator.

Possibly, there are more storage control techniques than languages. Language implementers often discard several techniques before selecting the one that best suits their language. (On the other hand, system implementers often discard several languages before selecting the one that best suits their storage control technique.) But all systems have two components, a memory organization suitable for storage allocation and a mechanism for control of that allocatable storage. The memory allocator is the part of the memory management system that provides a plex on request. This mechanism is vital to the efficiency of a system because, typically, plexes are created frequently. The storage control mechanism has the responsibility of structuring memory for allocation. In some systems, this is a continuous bookkeeping problem. In other systems a garbage collector is called when the allocatable space is exhausted.

III.1 Memory Organization for Allocation

There are three classes of memory organization for allocation: fixed-size, variable-size, and hierarchical. The fixed-size organization is very simple. Memory is structured into a list of free plexes, all of the same size. An allocation request is met by taking the first element from this 'free list'. Since all plexes are the same size, their relative position and the ordering of the free list is unimportant. Consequently fixed-size systems do not usually have relocation. Variable-size systems permit requests for plexes of different sizes. Such systems have been built both with and without relocation. The choice of fixed or variable for a system depends on the data structures being implemented. Fixed organization is simpler, but data usually comes in units of more than one size. Fixed techniques are important, though, for the part they play in hierarchical organizations.

The newest and most promising class of memory organizations for allocation are the hierarchical schemes. In these, a large plex is allocated for some purpose and smaller plexes are allocated from within the large one. In advanced schemes, the smaller plexes are themselves suballocated. There are several advantages to hierarchical allocation schemes. If a large plex holds smaller plexes of only one size, then within the large plex the garbage collector can use simple fixed-size collection techniques. Hierarchical allocation schemes can be useful for segregating the frequently changing from the seldom changing. The garbage collector ought to ignore the latter as much as possible. One possible approach is suggested by the lifetime block concept which has been proposed but not yet implemented. If a language has begin-end blocks like Algol and also has structure class declarations, all structures of a class can be deleted when control exits from the block containing the class declaration. Thus, the 'outer lifetime block' of an element of a structure class is that block containing the class declaration. Hierarchical structures might be used for life time blocks by simply releasing the large plex. Structures have a second kind of lifetime block; those blocks within which the structure will always exist. This might be, for example, an inner block making no operations on structures of a certain class. The garbage collector can assume that any structure is active if control is within this 'inner lifetime block'. Constant list structure is a limiting case; it always exists, so the entire program is its inner lifetime block.

There are not yet many hierarchical allocation systems. The L^6 [Know 66] allocation scheme, sometimes called the 'buddy system', is a cross between a hierarchical and a variable-non-relocating system. Each plex is the size of a power of two (up to 128 words on the 7090). Allocation may, if necessary, divide a free plex into two plexes half the size; these two plexes are called 'buddies'. A separate free list is maintained for each plex size. When a plex is freed, it is recombined with its buddy if possible. UNCLLL [Mnch 67] is a version of L^6 for the 360. Its allocation scheme distinguishes large (>8) and small (<7) plex requests. Small requests are met by suballocating fixed sized plexes from within a single large plex. The large plex size chosen for a given small plex size is such that these large plexes are about the same size. Both L^6 and UNCLLL maintain a bit table indicating free plexes. This permits rapid recombination of plexes. ALGOLW allocates pages of 4096 bytes (the 360/67 page size, although paging is not otherwise particularly facilitated). Each page is restricted to containing records (plexes) of only one record class,--and thus, only one size. Within each page standard fixed plex length garbage collection is employed. Two important hierarchical systems are those defined for LISP 2 and AED; they are described in Section III.2 under hierarchical garbage collectors.

Swym Memory Allocation

Swym employs a relocating variable-sized allocation organization. A garbage collector relocates all active plexes to one end of free storage. Plexes are allocated by moving a pointer that points to the beginning of the unallocated area. An additional advantage of this organization is demonstrated by- the Swym input routines. Arbitrary length strings can be read; each character is put into the next available location of free storage. When the end of the string is reached, a plexhead is provided and the string is automatically a character string atom. The same technique could be used when computing multi-word integers.

III.2 Storage Control

A language permitting dynamic storage allocation must have some form of storage control. The type required can depend on other language features:

> is there a 'release storage' instruction?
>
> are common sublists and common tails permitted?
>
> are circular lists permitted?
>
> are variable length plexes implied in the language?

Based on the answers to these questions, storage control techniques can be divided into classes similar to the classes for Memory Organization for Allocation:

> fixed - release
>
> fixed - no-release
>
> variable - non-relocating
>
> variable - relocating
>
> hierarchical

where

> 'fixed' and 'variable' refer to the size of plex allocated,
>
> 'release' refers to the presence of a 'release allocated storage' instruction in the language,
>
> 'relocating' refers to moving the plexes in storage.

Systems without 'release* usually depend on a garbage collector to find all active storage. Variable-non-relocating systems usually have 'release', because they are designed to have a minimum system. Variable-relocating systems do not have 'release' because they must do a large amount of processing anyway to relocate all of memory. Before the description of each class below, there is a list of systems in that class and suitable references.

<u>Fixed-Release</u>

IPL-v	[Newl 64]
SLIP	[Weiz 63]
REFCØ-III,SAC	[Coll 60] [MBth 63] [Coll 67]
AL, LEAP	[Feld 65] [Rovn 66, 67a, 67b]
TSA	[Toll pc]

In all these systems, except AL and LEAP, a list is an entity with a controlling list head; it is not possible to point at a part of a list without a list head. A list is released by pointing at the list head and issuing the release instruction. Storage is also released by deleting an element from a list. Lists can be pointed at by other lists or by the program variables. If a given list is pointed at by two or more pointers, the release operation is ill-defined: one routine may release a list that is still required by some other routine.

The systems solve this multiple-reference problem in different ways. IPL-V, the earliest popular system, required that the programmer be sure that a list was no longer required before releasing it. To aid in this task, programmers assigned certain bits in IPL-V structures as 'responsibility bits.' Routines could pass responsibility for lists by changing these bits. The REFCØ-III and SAC systems associated a *reference count' with each list head. This count kept track of how many pointers were pointing at a given list. The release process reduced a list's reference count by one. When the count reached zero, the list was purged. Unfortunately, the reference counts require a substantial amount of memory. In TSA, no list is ever referenced by more than one pointer. All operators destroy their arguments and make a new copy of any information to be saved. This applies to procedures as well: when a procedure is called, the arguments passed to the procedure are copies of the actual arguments. When a procedure exits, the storage for its arguments is released. TSA avoids garbage collection and bookkeeping at the expense of frequent list copying. In fact, none of these systems has a garbage collector, primarily because they are designed to be minimal and conceived of the garbage collector as detrimental to efficiency. But each of the above systems has a fault: programmer bookkeeping, memory consumption, or copying.

SLIP introduced a form of rings, two way connected lists. The programmer still must keep track of what list can be referenced and release any no longer needed. But the task is somewhat easier because lists can be traversed either forward or backward. SLIP discovered that it was not best to immediately scan a released structure and reduce it to a linear list on the free list. Instead it was more efficient to put the whole structure on front of the free list. The allocation mechanism is then designed to handle a structured free list rather than a linear one.

AL and LEAP are unlike any other languages in this report although they are intended for the same kinds of programs. They use plex processing internally but only to chain together the elements of hash buckets. Otherwise, the language is phrased in terms of attribute-object-value triples. These are stored in hash coded form on direct access storage. There is an operation to destroy a triple, but this simply means deletion of the link from the hash bucket. No garbage collector is required during execution, but if a file is saved it can profitably be reorganized.

Fixed-No-Release

LISP 1.5	[MCar 60, 62]
WISP	[Schr 67]
ALGØLW	[BBG 681
LISP 2	[Styg 671

In these systems the language designer relied on a garbage collector to find all inactive storage and create a new free list. Typically such routines are two passes: a marking pass finds allactive storage, a scanning pass finds all unmarked storage and structures it into a new free list. The marking pass may mark each active element with a bit either in the word itself or in a bit table. If a bit table is used, extra computation is required to relate bits in the table to addresses in free storage. But marking words themselves complicates direct numeric computation. [Schr 67] has an excellent review of scanning and marking techniques. It proposes a technique that avoids using the stack for temporary storage.

The ALGØLW garbage collector is included in this section because it is primarily a fixed-no-release system. Free storage is allocated in pages of 1024 words. Each page contains plexes of one fixed size, and there is a separate free storage list for each page. Each plex contains a marking bit for the garbage collector. The marking pass goes through all plex storage tracing and marking the active storage. The scanning phase creates a new free list on each page. If a page is empty, it is returned to the operating system; on the other hand, if a plex must be put on a full page, a new page is created for the required structure class. One problem with this scheme occurs when a class is nearly full and a process is creating and deleting members of that class. The garbage collector may be called several times before a new page is created. But the garbage collector blindly rescans all active storage even if only one class is changing. Insufficient experience has been gained with ALGØLW plexes to propose a better garbage collection strategy.

Only one portion of the LISP 2 garbage collector fits in this section; the rest is discussed in the section on hierarchical garbage collectors. There is a requirement in LISP 2 to relocate the fixed length list cells so they only occupy the bottom of their free storage area, After the marking pass, the lowest free word is swapped with the highest active word. The new address of the active word replaces its old location. This process, called folding compaction, continues until all active words are at the bottom, A final pointer correction pass is required. Any pointer into the free area is replaced with the new address stored in that location.

Variable-Non-Relocating Systems

L^6	[Know 663
ASP	[Gray 67] [Lang 68]
APL	[Dodd 66]
UNCLLL	[Mnch 671
CØRAL	[Suth 66] [Kant 66]
ØS/360	[IBM 68a]

Knuth concentrates on systems in this class [Knth 67], so the discussion in this section is brief. His analysis and simulations indicate that fragementation occurs with a tolerable low frequency, Given that assumption, the techniques in this class are to be preferred for their low overhead. If the language permits common sublists or circular lists and requires the programmer to release inactive plexes, then he must write code to keep track of how many pointers point at each plex. But some problems seldom require common sublists.

For such problems the variable-non-relocating systems are attractive.

ASP and APL use the L^6 buddy system for allocation, but, like CORAL, they organize the data into rings. Nodes of a ring are plexes, and each may have several ring connections and several data fields. Nodes can only be accessed along the rings, so the only delete operation needed is to delete a node from a ring. When a node is connected to no rings, it can be returned to the free storage list. There is the problem that circular structures may never be freed even though inaccessible.

When a requested plex is larger than the largest free area, the schemes in this class must try combining adjacent free plexes into larger free plexes. In some systems, recombination is attempted every time a plex is freed. In an application with many plexes of about the same size, however, the likelihood is that the recombined plex will soon be broken up again. Recognition of neighboring free plexes is not always trivial. One technique is to sort the free list according to core location and then compute adjacency from locations and lengths. CORAL has a plexhead marked by containing one field of all ones. Checking for a free neighbor in the upwards direction is easy (the next plexhead follows the current plex); but finding a preceeding neighbor means searching back to find a plexhead. UNCLLL associates a bit table with free storage. A bit is set for the first and last word of each active plex.

Operating System/360 dynamically allocates variable length blocks (GETMAIN and FREEMAIN macros) and requires some form of storage management. Relocation is impossible because programs manipulate absolute addresses and the system cannot know where a problem program has stored an address. Free storage is structured in blocks chained together in sequence by their size. Allocation is accomplished by-finding an appropriately sized block or dividing a larger block. When a block of storage is returned to free storage, it is placed on the chain according to its current size. When a sufficiently large block is not available, OS tries to combine adjacent free blocks into larger free blocks. This is accomplished by maintaining an additional chain pointing to the blocks in sequence by core address. The garbage collector scans this second chain trying to combine each block with the next higher block in memory. If no sufficiently large block is built to satisfy the user request, he is either terminated or given a return code indicating his request was not met.

<u>Variable-Relocating</u>

COGENT	[Reyn 65]
	[Hadd 66]
EPL	[MCla pc]
EULER	[Wrth 651
MUTANT	[MKee 663
XPL (strings)	[MKee pc]
SWYM	(this paper)

A variable-relocating garbage collector completely ignores the garbage. Instead, it builds a new structure isomorphic to the old with respect to the permitted data access operations. The time for this process depends on the size of the active structure and sometimes on the incidental arrangement of the elements of that structure. Many systems relocate storage by coalescing the active plexes; that is, moving them all toward one end of memory, without rearranging them. Others, like SWYM, not only move all plexes, but also change their order. In SWYM this process tends to move together lists and their elements, an important property for paging systems, But there is a disadvantage in rearranging memory. In non-relocating and simply coalescing systems, the address of a plex can be used as an arbitrary ordering function. Such functions have utility when manipulating otherwise unordered sets. In systems that rearrange storage, such pseudo-ordering functions are difficult to define.

Most variable-relocating garbage collectors have four phases in some order or another. As identified in [Styg 67] these are Find, Plan, Fix, and Move. The Find phase is responsible for finding all active structures. The new address of each structure is computed by the Plan phase. During the -Fix phase all pointers are changed to point at the new locations of the structures. Finally, the Move phase relocates all structures.

In the Find phase, a tracing algorithm goes down all chains of pointers starting with the pointers on the stack and in the static variables. To identify the active plexes and to avoid processing a plex more than once, a visited plex is usually marked in some way. If bits are available in each plex, the plexes can be marked within themselves. Otherwise, a bit table can be used. In the latter case, extra computation is required to find the relation between a word address and a bit in the table. If a plex contains more than one pointer, the tracing algorithm must be applied to all of these. There must be some way to remember those pointers that have not yet been traced. One simple solution is to put all the pointers from the plex on the stack. The tracing routine always takes the top pointer off the stack. But this system can use large amounts of stack space. Space requirements can be reduced by stacking a pointer to the plex and a counter indicating how many of the pointers in the plex have been collected. If room for this counter can be found in the plex itself, then the WISP technique [Schr 67] can be used to eliminate the need for a stack.

The other three phases must also be designed with care. During the Plan phase, the new address of each plex must be saved for succeeding phases. Some plex encodings leave room in each plex for the garbage collector to store this new address. Others use the free-areas to store information to compute the new addresses. In a system that merely collapses storage by moving it all down, it is sufficient to compute the change between the old address and the new address. Systems, like SWYM, that rearrange the plexes must be prepared to associate an entire new address with each plex. The Fix phase, like the Find phase, must locate all pointers. Processing a pointer twice, however, is not only time consuming as it is in the Find phase, but is also fatal as the second update might access erroneous data. Some systems create a list of pointers during the Find phase for use by the Fix phase; ordinarily, though, this is an exhorbitant waste of space. The most common solution is some form of marking bit. During Move, care must be taken that no plex is overwritten with a new plex before it itself has been moved. In push down relocation, this is accomplished simply by moving plexes starting with the lowest in memory. SWYM, on the other hand, relies on secondary storage to hold the new contents of memory.

The COGENT system uses a bit table for marking the active words of storage. Each plex contains a type field, Depending on the type, the garbage collector determines exactly which components in the plex are pointers. The yet-to-be-traced pointers are remembered by stacking a pointer to the plex and a count of the number of pointers that have been traced. The relocation factor for each block of storage is stored in the first word of the next free area. The Fix phase precedes the Move phase.

Haddon and Waite [Hadd 66] have described a push down garbage collector that creates a table of relocation factors during the Move. This table is then sorted on the 'old address' field. The Fix phase is last: each pointer is found in the table by binary search and the associated relocation factor is applied to correct the pointer.

Don McClaren [MCla pc] proposes to use a modification of the preceeding plan. Descriptors for each plex are stored not with the plex but in the upper portion of free storage. (His system is PL/1-like and the assumption is that the descriptors are infrequently referenced.) The garbage collector can find the descriptor for each plex by a binary search on the table. The 'descriptors contain room for the relocation address of each plex. The point of this approach is that the garbage collection features have very low cost if they are not used. Indeed, the descriptors can be removed altogether with little reprogramming (if the garbage collector is not used).

W. McKeeman has written several garbage collectors, including those for EULER, MUTANT, and XPL (strings). These systems rely on descriptors and store all lists (strings) as a plex of pointers (characters). A descriptor contains the beginning location of the item and its length. In XPL, a portion of a string can be identified'as a separate string by simply specifying a different beginning and length; this corresponds neatly to PL/1 SUBSTR expressions. The MUTANT and EULER garbage collectors are similar; each beginning by scanning all active structure and abstracting all descriptors. These descriptors are stored in a newly created array (using B-5500 Algol). Note that this requires a substantial amount of temporary storage. This descriptor array is then sorted by the location of the list. In the Move phase, active blocks are moved down; the new address of each block is stored in a field of the descriptor reserved for this purpose. The last step is to scan through memory and update the address fields of all descriptors. The XPL string garbage collector improves on this process by creating a list of pointers to the descriptors, rather than a list of the descriptors. Since only the string area is being garbage collected, the descriptors will not move. This list of pointers to descriptors is sorted based on the address fields in the descriptors. Finally, in a single pass all active portions of strings are moved downward and the new addresses are stored in the descriptors.

Hierarchical

LISP 2	[Styg 67] [Hawk 67]
LISP 1.5	[Barn 68]
AED	[ROSS 67]

Hierarchical storage control schemes are characterized by allocating plexes within larger plexes, called super-plexes. In the more general schemes, super-plexes are allocated within larger super-plexes. Hierarchical schemes can use different garbage collection techniques for different super-plexes. This approach permits each type of data to be collected by a routine specifically written for that data type. Such specific routines can avoid type testing and can thus reduce garbage collection time.

A major problem in a hierarchical system is deciding the size of the space that should be allocated to each super-plex. One approach is that used by ALGOLW and described above. But this system can call the garbage collector frequently if pages are nearly full. One solution to this problem is to attempt to determine the rate of change in the storage requirements for each class. Garwick has proposed and implemented such a scheme for the array feature of GPL [Garw 68 and Knth 68]. In that system, array declarations must specify an upper bound but the current upper bound dynamically depends on how many of the cells are full. At garbage collection time, a new length is calculated for each array as a function of its current length and its length at the time of the preceeding garbage collection. A similar system is used in the SDC LISP 1.5 for the 360 -[Barn 68] to assign to each storage area an appropriate number of 256 word blocks.

One other serious problem can occur in an allocation scheme like that used by ALGØLW: two large structures can be created simultaneously and occupy many pages. If only one of these structures is required later in the program and if no other structure is created in the given storage class, then all pages remain active for the storage class although they are only partially occupied. The probability of this problem occurring is program dependent, but the loss of storage can be large. This can be avoided by relocation, or by splitting the class into two classes. The problem is more complex when pages are being swapped; the decision must be made as to whether the time to relocate memory is less than the time spent in swapping the inactive portions of pages.

Memory is allocated hierachically in both LISP 2 and AED that is, plexes are allocated from within other plexes. But the details differ; LISP 2 permits only a system defined hierarchy and garbage collects it very efficiently, AED sacrifices some efficiency to permit complete user control of allocation.

In the LISP 2 system, different types of program values are stored in different areas of memory. Some areas contain only fixed length plexes, others contain variable length plexes. The areas are paired; each pair is assigned a super-plex and one member grows up from the bottom while the other grows down from the top. Thus the folding compaction described above is necessary for the fixed length areas. Provision is also made for changes in the size of the plex assigned to each pair. No indication is given of the basis for these size changes.

The AED system defines an allocation scheme that is essentially non-relocating. However, provision is made for the user to write routines to be called when storage is exhausted in a super-plex. Thus the user can define his own garbage collector. The system provides a plethora of primitives to assist in writing this garbage collector. Adding to the confusion in the field, the AED system defines a GARBCOLL mode. This mode can be set on for a super-plex that controls sub-plexes with a variable-non-relocating (with release) scheme. When GARBCOLL is in effect, a released plex is automatically combined with any adjacent free plexes. When GARBCOLL is off, freed plexes are merely kept on a list (which AED calls a string).

Basic Swym Garbage Collector Algorithm

Swym contains a variable-relocating garbage collector that creates a set of structures isomorphic to all active structures with respect to <u>rst</u> and <u>fst</u>. Most unnecessary <u>rst</u> pointers are eliminated. This set of structures is in a new core image, created sequentially and written to a temporary storage device. After collection, the new core image is read into one end of the plex storage area and the remainder of that area becomes the new free storage area.

The idea of using external storage was suggested by Marvin Minsky in an internal MIT memorandum [Mnsk 63]. But the algorithm reported there would not work for even the simplest cases (for instance, the structure in Figure III.2). The Swym garbage collector works not only for the simplest cases, but also for the most complex cases of mutual circularity. The complete garbage collector is described in Appendix E; the current section presents a minimal version of the garbage collector to illustrate the central ideas. This minimal version is satisfactory only for structures that never have more than one pointer at any given word of the structure.

COLLECT (x), the portion of the garbage collector presented here, has as its argument a pointer at a piece of list structure. It then writes that list structure sequentially to the new core image. Other functions exist to call COLLECT for each possible pointer at active structure, to collect atoms, and to read in the new core image.

The contents of a list are address pointers to the elements of that list. When a list is written to new core, the contents of that list must be the new-core addresses of the elements of that list. Consequently, the elements of a list must be COLLECTed before the list itself can be written to the new core. COLLECT (x) proceeds in two recursively intertwined passes. The first pass applies COLLECT to each element of the list x. The second pass writes the new representation of the list x to the new core image. To remember where a piece of list structure is in new core, its fst is replaced (rplf) -with the address of that structure in the new core. The head of an atom is used to store the address of that atom in new core.

Three operators must be defined in order to describe the garbage collector:

ATCOL (x) x must be an atom. If x has not been garbage collected, it is collected and written to the new core image. The plexhead of x is replaced with the address of x in the new core. ATCOL calls separate routines to garbage collect each type of atom.

GCPUT (x) x is any full word. This word is written to the next available location in the new core image. The value of GCPUT is the address of that location. An internal variable is advanced to point at the next available location in the new core image. GCPUT handles I/O and writes buffers to the external device when necessary.

HD (x) x must be an atom. HD returns the plexhead of that atom; after ATCOL, the plexhead contains a pointer to x in the new core. If x is non-atomic, processing is interrupted.

The basic garbage collection algorithm is given in Figure 111.1 in a notation similar to Algol. The declarator list declares a variable which may point at a piece of list structure. The declarator word declares a variable whose value is one full word. Note that rstbit is initialized to the value 1. This corresponds to the value of a word with just the rst bit on. rstbit is used to 'or' the rst bit into a word written to the new core image. The result of applying COLLECT to a simple structure is shown in Figure III.2.

'Garbage collection' is truly a misnomer for this algorithm. COLLECT examines only the active list structures, while the garbage is completely ignored and has no effect on the processing. 'Storage reclamation' describes the process no better. Possibly better terms might be 'storage reorganization' or 'garbage control'. But the term 'garbage collection' is so widely used and so colorful as to preclude replacement.

Some limited experiments have beenconducted with the Swym garbage collector. On one list structure, representing a program, there was a 25 per cent saving of storage using compact lists instead of standard lists. This corresponds to an average list length of only two elements. The correspondence

Figure III.1

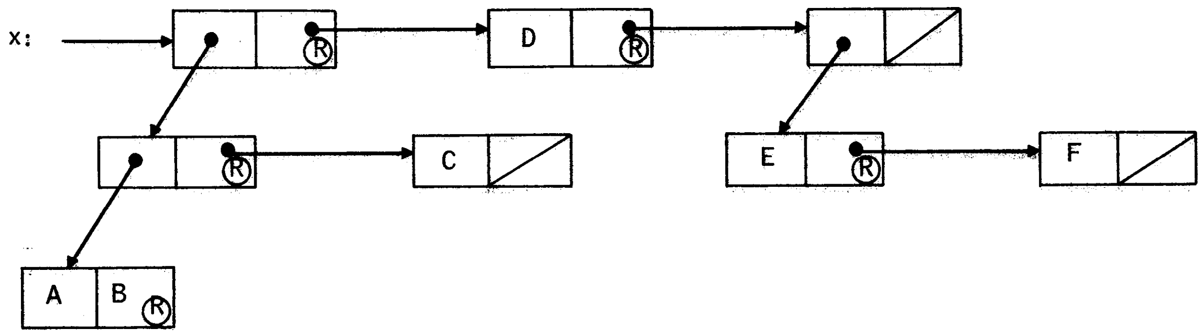
Simplified Swym Garbage Collection Algorithm

```
COLLECT (x)  = begin list r, t; word rstbit := 1;

      r := x;
chkloop: t := fst (r);
      if atom (t) then ATCOL (t) else COLLECT (t);
      t := rst (r);
      if atom (t) then ATCOL (t)
      else begin r := t; goto chkloop  end;

      r := x;
wrloop: t := fst (r);
      rplf (r, if atom (t) then GCPUT (HD (t))
               else GCPUT (fst (t)));
      t := rst (r);
      if atom (t) then GCPUT (HD (t) ∨ rstbit)
      else begin r := t; goto wrloop  end
end COLLECT
```

Figure III.2

Example of Swym Garbage Collection

Initial Structure:



At wrloop on the highest level:

Old Memory

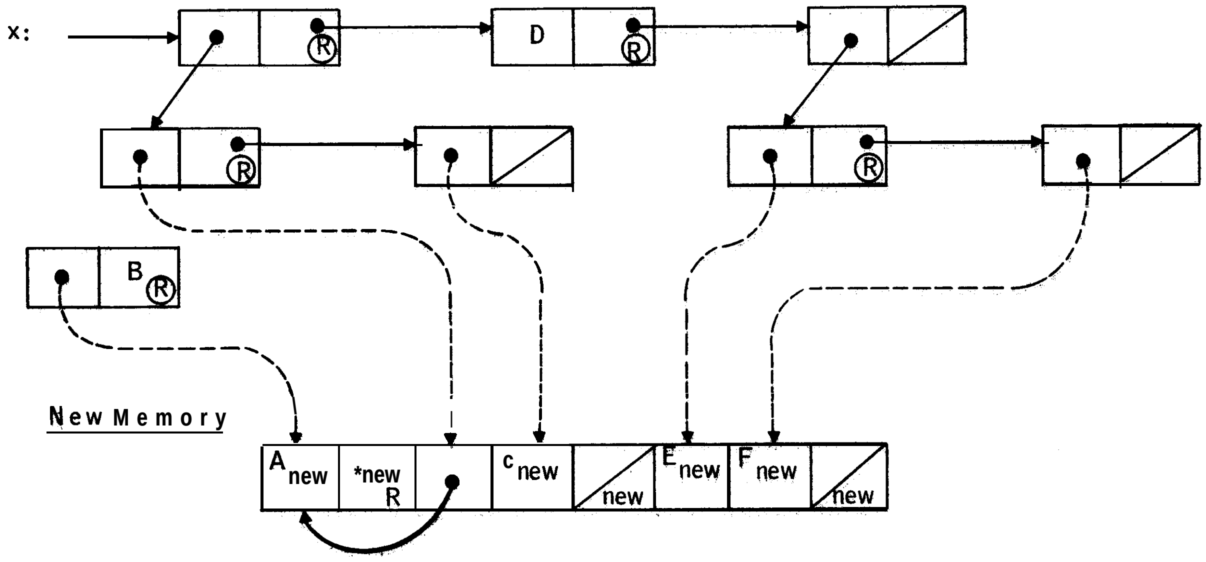
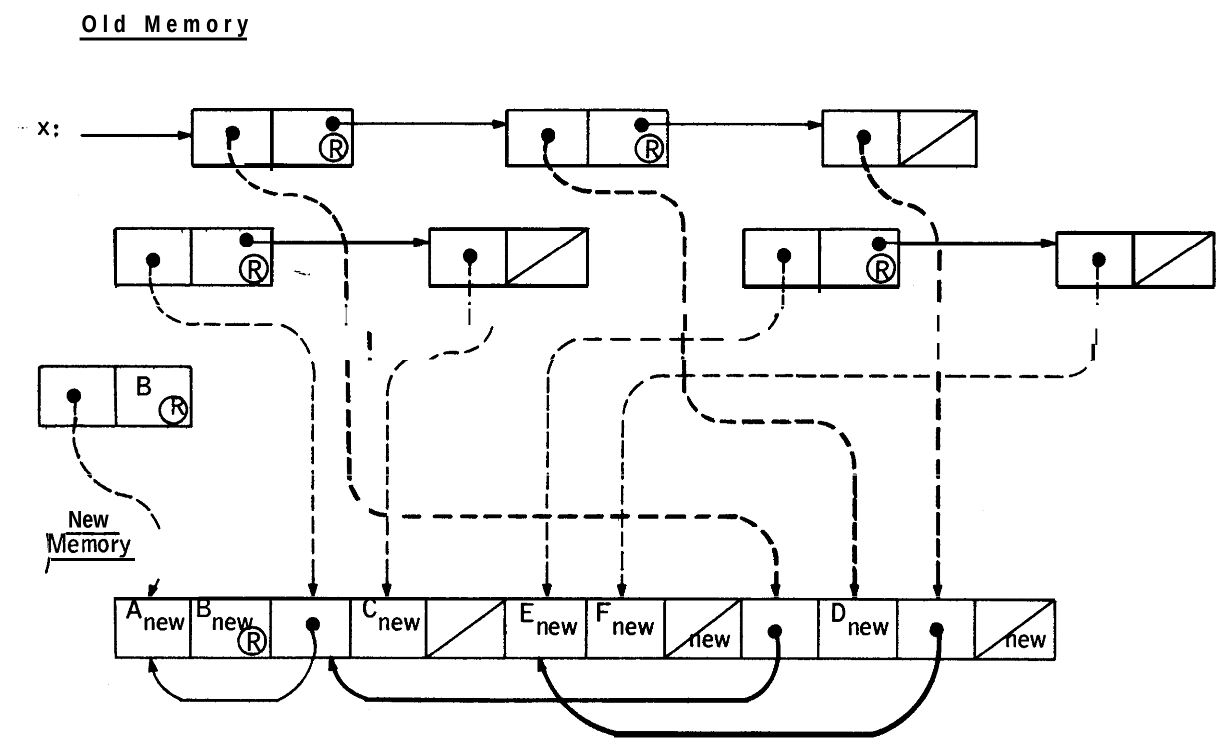


Figure III.2 (Cont)

Example of Swym Garbage Collection

At completion:



Note: ▬ **new pointer**

─ **pointer unchanged from preceding diagram**

- - - } **pointer at location a word will occupy after the new core image has been read in.**

is easy to compute: A normal list of length $\underline{n}$ requires $2\underline{n}$ pointers. The corresponding compact list requires n+1 pointers, for a saving of $1 - \frac{\underline{n}+1}{2n}$; when $\underline{n}$ is 10, the saving is 45 per cent. Every symbol atom takes at least four words of storage plus the length of the print name, so the number of symbol atoms is also a factor.

For every active word of storage, roughly forty instructions were executed during garbage collection. This was computed by dividing execution time into amount of active storage. The experimental system did not use external storage; instead, memory was shuttled between two alternate core areas. Thus the time to write out memory is the maximum of the time to write out the active structure and the time to execute forty instructions for each active word. The time to read in memory is dependent solely on the number of active words. The Swym garbage collector speed can be contrasted with the speed of that routine in the Stanford LISP360 system. This is a standard LISP 1.5 implementation with a fixed-no-release garbage collector. LISP cells are stored in double wards. The garbage collector executes approximately fifteen instructions for each active double word. In addition, the linear scan through free storage requires four instructions for each of the double words in free storage. These rates were computed based on execution of several large programs on a 360/75.

Several applications for the Swym garbage collector are conceivable, even apart from compact list structure. The Swym garbage collector could be valuable in a system with roll out and roll in. If the monitor set a signal for the program to roll itself out, the program could garbage collect for free the next time a <u>cons</u> was executed. Even without memory swapping, external storage of structures has always been a problem for plex processing systems. The Swym garbage collector provides analgorithm for scanning lists and storing them in a compact form on an external device. Another application for this algorithm is in the transmission of list structures between two machines over a slow channel. If the new storage is written starting at location zero, the address fields can be small. Only as the size of the structure passes a power of two would the length of each address "field have to increase.

The implemented garbage collector stores partially collected structures on the stack, but uses a trick to avoid saving return addresses during recursion. It would be possible to use the WISP technique [Schr67] to avoid using the stack during collection. This was not done because it would involve at least two more passes over the data. In a memory sharing environment, it is sometimes possible to acquire temporarily the needed extra storage for a stack; otherwise, sufficient stack must be available to hold at least twice the length of the longest _fst_ chain.

CONCLUSION

The best conclusion to this paper would be to point to a specific set of environments and say, "These are the best for implementing a plex processing language." But this cannot be done because storage management is highly problem dependent. A set of environments satisfactory for one language may be very poor for some other language. For completeness, four storage management schemes are necessary: fixed-release, fixed-no-release, variable-relocating, and variable-non-relocating. The most universal approach is a hierarchical system offering each of these types of storage control; current work holds the promise of making this approach as efficient as the least efficient of the facilities actually used. That is, it seems possible to 'charge' the user the 'cost' (time or memory) of only the storage management technique he uses. Alternatively, large projects should consider implementing a language and system suited to their own particular needs. Since all environments can be conveniently implemented with a combination of a stack and variable length plexes, a general storage management system like Swym is a suitable basis for the development of specialized languages.

The paper will close with (1) a summary of the SWYM solution to a variable-relocating storage management system and (2) the implications of plex processing languages for hardware design.

Summary of Swym Environments

Stack: The Swym stack stores pointers, return addresses, and stack plexes. The three are distinguished by the high and low order bits of the word. For plexes these bits are in a plexhead and all other words in the plex can be full 32-bit words. The stack grows toward lower addresses so routines may address local variables they store on the stack.

Data structures: To permit compact lists, Swym distinguishes between lists and all other plex structures. The distinction is based on the pointer at the item, plexes being addressed six bytes in front of their plexhead. List operators will not work on plexes and vice versa. But this is advantageous in debugging, and neither type of operation is slowed because this checking is done by hardware. All plexes have a plexhead, which is memory consuming if many small plexes are used.

Routine linkage: The stack is essential to routine linkage: return addresses are stored on the stack, and the calling routine stores any active registers on the stack. The address of each routine is available from a transfer vector table.

Variable binding: STUTTER variables are bound in a value cell associated with the atom representing the variable. A bit in the plexhead indicates whether the value is a pointer or a full word of information, so a compiler can compile direct numerical operations. When an atom is rebound, the current binding is saved on the stack and the new binding placed in the cell. Dynamic free variables are not permitted.

Memory allocation: Memory is allocated from one end of a single large free area. This could be used like a stack, but this is rare in STUTTER.

Storage management: The Swym garbage collector creates a representation of all active structures on secondary storage. This representation is then read into one end of the free storage area. In this process lists are compacted, and related structures are relocated near each other.

Implications for Hardware Design

Because storage management is very problem dependent, hardware design should not favor one technique over others. But three features would facilitate storage'management and language implementation: 1) extra bits in every word, 2) stack operations, 3) subroutine operations. Other operations, like data access and program control, seem to be adequately handled by the 360 hardware. Appendix K contains one proposal for instructions implementing these proposals.

Extra bits in every word: Swym utilizes high and low order bits of pointers in many ways. But careful control is necessary to avoid confusion with numbers. Much bit testing and indirection could be avoided if each word included two or more bits that did not participate in arithmetic operations. This idea has been implemented in at least the B-5500 and other Burroughs machines. But very careful design would be required to integrate extra bits into the design of the 360, because so many different kinds of instructions can access different portions of each word. One approach would be to associate four bits with each word that could be set and tested with special storage immediate instructions but would not otherwise participate in arithmetic operations. These bits could be considered as one per byte to mark the ends of strings, or could be considered as four per word with different configurations marking pointers, integers, floating point numbers, or other data types. One or two of the bits with a word could be used for marking by a garbage collector. In a carefully worked out language implementation, the special bits would only have to be set when memory was allocated.

Another possible approach to associating bits with every word would be to provide an instruction that translates a word address into a bit address (and possibly tests or alters that bit). With this approach the user would have no expense if he did not use the facility. But if he did, memory allocation would be required both for data and for any associated bit tables.

Stack operations: A stack can be invaluable in many programs and is essential in implementation of plex processing languages. Moreover, the required operations are relatively simple and non-controversial: add an item, delete an item, and reference an item. With no provision for checking the ends of the stack, the add and delete operations can be placed in micro-code, and the reference operations can use ordinary base-displacement addressing. End checking is a little more complex. One approach is to make the stack pointer a pointer at a descriptor giving the ends and the current location of the stack. But this prevents using the stack pointer to reference items on the stack. An alternative is to use special settings of the special bits to indicate the ends of the stack. The special bits would then be checked by the micro-code.

Subroutine operations: Like stack operations, these are easy to implement and are of general utility. The basic subroutine operations are call and return, using the stack to store the return address. Storage of registers and other status information is more language dependent and should be controlled by the calling routine.

BIBLIOGRAPHY

[Barn 68] Barnett, J.A. and Long, R.E. The SDC LISP 1.5 System for IBM 360 Computers. System Development Corporation Document SP-3043. (Jan, 1968).

[BBG 681 Bauer, H.R., Becker, S. and Graham, S.L. ALGOL W Language Description, Computer Science Department, Stanford University, (Jan, 1968).

[Baur pc] Bauer, H. Stanford University, Stanford, California.

[Benr 67] Benner, F.H. On designing generalized file records for management information systems. AFIPS V. 31, 1967 FJCC. Thompson Books, Washington, D.C., pp. 291-303.

[Bobr 67] Bobrow, D.G. and Murphy, D.L. Structure of a LISP System Using Two-Level Storage. Comm. ACM 10, 3 (Mar, 1967) pp. 155-159.

[Bobr 68] Bobrow, D.G. (Ed.) Symbol Manipulation Languages and Techniques. North-Holland Publishing Co., Amsterdam, Netherlands, 1968.

[Cohn 67] Cohen, J. A use of fast and slow memories in list processing languages. Comm. ACM 10, 2 (Feb, 1967) pp. 82-86.

[Coll 60] Collins, G.E. A method for overlapping and erasure of lists. Comm. ACM 3, 12 (Dec, 1960) pp. 655.

[Coll 67] Collins, G.E. The SAC-1 list processing system. Computer Science Department and Computing Center, University of Wisconsin, (July, 1967).

[Comf 64] Comfort, W.T. Multiword list items. Comm. ACM 7, 6 (June 1964), pp. 357.

[Dodd 663 Dodd, G.G. APL, a language for associative data handling in PL/1. AFIPS Conf. P. V. 29, FJCC, Spartan Books (Wash, 1966) pp. 677-689.

[Farb 64] Farber, D.J., Griswold, R.E., and Polonsky, I.P. SNOBOL, A string manipulation language. J. ACM ii, 1964, pp.21-30.

[Feld 65] Feldman, J.A. Aspects of associative processing. TN 1965-13 Lincoln Lab MIT 1965.

[Frnk 66] Franks, E.W. A data management system for time-shared file processing using a cross-index file and self-defining entries. AFIPS Conf. P. V. 28 1966 SJCC pp. 79-86.

[Garw 68] Garwick, J.V. GPL, a truly General Purpose Language. Comm ACM 11, 9 (Sept,1968), pp. 634-638.

[Gerl 60] Gerlernter, H., Et Al. A FORTRAN-compiled list-processing language. J. ACM 7, 1960, p. 87.

[Gray 67] Gray, J.C. Compound data structure for computer aided design; a survey. Proc. ACM 22nd Nat. Conf. 1967, Thompson Books, Washington, D.C., pp. 335-365.

[Gris 67] Griswold, R.E.,Poage, J.F., and Polonsky, I.P. Preliminary Report on the SNOBOL 4 Programming Language II, Bell Telephone Laboratories, Holmdel, New Jersey (Nov, 1967).

[Hadd 663 Haddon, B.K. and Waite, W.M. A compaction procedure for variable-length storage elements. Computer J. 10, 8 (Aug, 1966).

[Hawk 671 Hawkinson, L. Lisp 2 Internal Storage Conventions. Systems Development Corp. TM-3417/550/00 (Apr, 1967).

[Hoar 66] Hoare, C.A.R. Record handling. IFIP working conference, Pisa (Sept, 1966).

[IBM 68al International Business Machine&IBM System/360 Operating System MVT Supervisor, Form Y28-6659-2 Kingston, New York, (Jan, 1968).

[IBM 68b] International Business Machines. IBM System 360 PL/I Reference Manual. Form C28-8201-1, Kingston, New York, (March, 1968).

[Ilif 62] Iliffe, J.K. and Jodeit, J.G. A dynamic storage allocation scheme. Comput. J. 5, 10 (Oct, 1962) p. 200.

[John 631 Johnson, T.E. Sketchpad III 3-D graphical communication with a computer. ESL-TM-173 M.I.T., Cambridge, Mass., 1963.

[Joyc 671 Joyce, John D. and Cianciolo, Marilyn, J. Reactive displays: improving man-machine graphical communication. AFIPS Conf. P. V. 31, FJCC (Sept, 1967) pp. 713-721.

[Kant 661 Kantrowitz, W. CORAL - A Questionnaire for language consultants. ACM Special Interest Committee on Symbolic and Algebraic Manipulation, Comparison of Languages Subcommittee (May, 1966).

[Know 663 Knowlton, K.C. A programmer's description of L^6 . Comm. ACM No. 8, 9 (Aug,1966).

[Knth 683 Knuth, D.E. The Art of Computer Programming, Vol. I. Addison-Wesley, Menlo Park, California, 1968.

[Lang 68] Lang, C.A., and Gray, J.C. ASP - A ring implemented associative structure package. Comm. ACM 11, 8 (Aug, 1968) pp. 550-555.

[MBth 63] McBeth, J.M. On the reference counter method. Comm. ACM 6, 9 (Sept, 1963) p. 575.

[MCar 60] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part I Comm. ACM 3, 4 (April, 1960) p. 184.

[MCar 62] McCarthy, J., et. al. LISP 1.5 Programmer's Manual. The MIT Press, Cambridge, Mass., 1962.

[MCla pc] McClaren, M.D., Argonne National Laboratory, Argonne, Illinois.

[MKee 66] McKeeman, W.M. An Approach to Computer Language Design. Technical Report CS48, Computer Science Department, Stanford University, (Aug, 1966).

[MKee 67] McKeeman, W.M. Language directed computer design. AFIPS Conf. P. V. 31 FJCC 1967, Thompson Books, Washington, D.C. pp. 413-417.

[MKee pc] McKeeman, W. University of California at Santa Cruz, Santa Cruz, California.

[Mnch 67] Manacher, G.K. and Dewar, R.B.K. The UNCLLL List-Processing Language: A Preliminary Description. University of Chicago, 1967.

[Mnch pc] Manacher, G.K. University of Chicago, Chicago, Illinois.

[Mnsk 63] Minsky, M.L. A LISP garbage collector using serial secondary storage. MIT Artificial Intelligence Memo. No. 58, Cambridge Mass., (Oct, 1963).

[Newl 64] Newell, A. (Ed.) Information Processing Language-V Manual, 2nd ed. Prentice Hall, Englewood, N.J., 1964.

[Perl 60] Perlis, A.J. and Thornton, D. Symbol Manipulation by Threaded Lists. Comm. ACM 3, 4 (Apr, 1960), pp. 195-204.

[R&R 64] Randell, B. and Russell, L.J. Algol 60 Implementation. Academic Press, London, 1964.

[Reyn 65] Reynolds, J.C. Cogent Programming Manual. Argonne National Lab. report no. ANL-7022. Argonne, Illinois, (Mar, 1965).

[Reyn 69] Reynolds, J.C. GEDANKEN - A Simple Typeless Language which permits Functional Data Structures and Coroutines. Argonne National Laboratory, (May, 1969).

[Reyn pc] Reynolds, J.C. Argonne National Laboratory, Argonne, Illinois.

[Ross 61] Ross, D.T. A Generalized Technique for Symbol Manipulation and Numerical Calculation. Comm. ACM 4, 3 (Mar, 1961) pp. 147-150.

[Ross 67] Ross, D.T. The AED free storage package. Comm. ACM 10, 8 (Aug, 1967) pp. 481-492.

[Rovn 663 Rovner, P.D. Investigation into paging a software simulated associative memory system. Document No. 40 10 90, University of California, 1966.

[Rovn 67a] Rovner, P.D. and Feldman, J.A. An Associative Processing System for Conventional Computers. Lincoln Laboratories, TN 1967-19.

[Rovn 67b] Rovner, P.D. and Feldman, J.A. The Leap Language and Data Structure. Lincoln Laboratories, November 1967.

[Satt pc] Satterthwaite, E. Computer Science Department, Stanford University.

[Schr 671 Schorr, H. and Waite, W.M. An efficient machine - independent procedure for garbage collection in various list structures. Comm. ACM 10, 8 (Aug, 1967) pp. 501-506.

[Styg 671 Stygar, P. LISP 2 garbage collector specifications. systems Development Corp. TM-3417/500/00, Santa Monica, Calif., (April, 1967).

[Suth 633 Sutherland, I.E. Sketchpad: a man-machine graphical communication system T 296, Lincoln Lab., M.I.T., Lexington, Mass., 1963.

[Suth 663 Sutherland, William R. The Coral Language and Data Structure (Appendix C) from Lincoln Laboratory, TR No. 405, May 1966.

[Toll pc] Tolliver, B.L. Computer Based Laboratory, Stanford University, Stanford, California.

[vDam 67] Van Dam, A. and Evans D. A compact data structure for storing, retrieving and manipulating line drawings. AFIPS P. v. 30 SJCC 1967, pp. 601-610.

[Weiz 63] Weizenbaum, J. Symmetric list processor. Comm. ACM 6, 9 (Sept, 1963) p. 524.

[Weiz 68] Weizenbaum, J. The Funarg Problem Explained. MIT, Cambridge, Mass., (Mar, 1968).

[Wilk 64a] Wilkes, M.V. An experiment with a self-compiling compiler for a simple list-processing language. Annual Review in Automatic Programming, Vol. 4. Pergamon Press, N.Y., 1964, pp. 1-48.

[Wilk 64b] Wilkes, M.V. Lists and why they are useful. Proc. ACM 19th Nat. Conf., (Aug, 1964).

[Wise 66] Wiseman, N.E. A simple list processing package for the PDP7. Second European Seminar of DECUS, (Oct, 1966).

[Wrth 65] Wirth, N. and Weber, H. Euler: A generalization of Algol and its Formal Definition. Technical Report CS20, Computer Science Department, Stanford University, (April, 1965).

[Wrth 66] Wirth, N. and Hoare, C.A.R. A contribution to the development of ALGOL, Comm. ACM 9, 6 (June, 1966), pp. 413-431.

[Wrth 68] Wirth, N. PL360, A programming language for the 360 computers, ACM 15, 1968, pp. 37-74.

[Yngv 62] Yngve, V.H. et. al. COMIT Programmer's Reference Manual, the MIT Press, C-bridge, Mass., 1962.

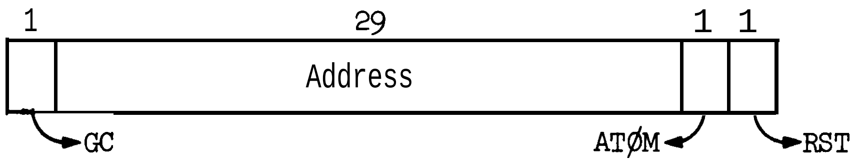
Appendix A. Details of Swym Structures

There are many different information structures in Swym. Free storage contains lists and plexes (also called atoms), while the stack contains pointers, return addresses, and plexes. All currently implemented varieties of these structures are described below.

A.l. Free Storage Structures

a. Lists

A list word has the structure



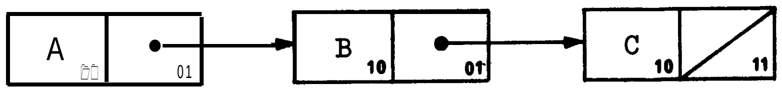
ADDRESS. May point at another list element, or at an atom.

GC. Is used by the garbage collector for marking (bit Ml).

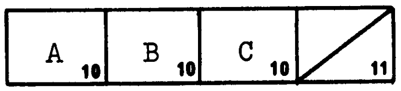
RST. Is on to indicate that the continuation of the list is at location ADDRESS. RST is also used by the garbage collector (bit M2).

-ATOM. Is on to indicate that ADDRESS points at a plex (or atom). ATOM is on automatically because a pointer at a plex points six bytes in front of the plex.

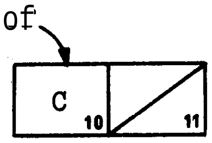
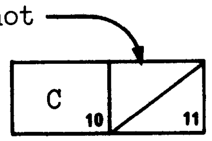
In the following examples, the two low order bits of each pointer are indicated explicitly. A pointer at an atom is indicated by the printname of the atom and the presence of the ATOM bit. The list (A B C) may be represented by



or



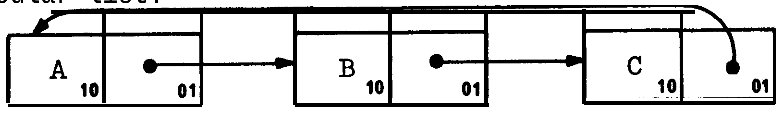
Note 'that the rrrst of either structure is a pointer at the atom NIL.

That is, rst of  is not 

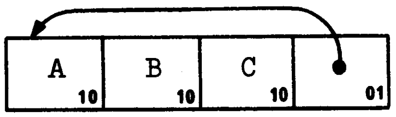
but is the pointer at NIL (contents of the second word). It is important to note that no valid pointer will point at a list element with the RST bit on.

The Swym list structure can represent both circular lists - which cannot be printed, and lists with common subelements - which are not printed correctly.

Circular list:



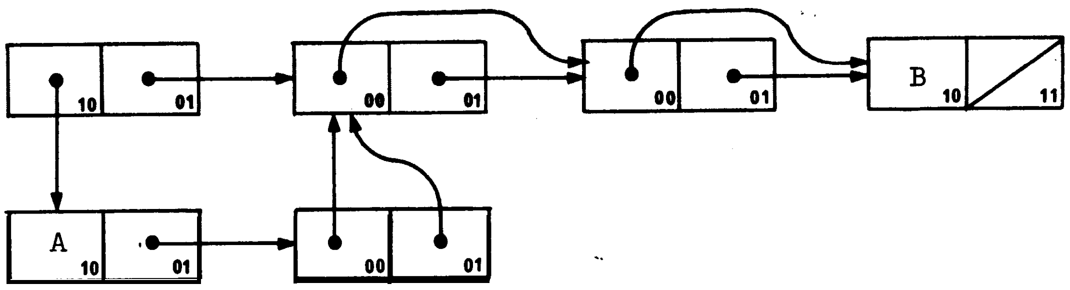
or



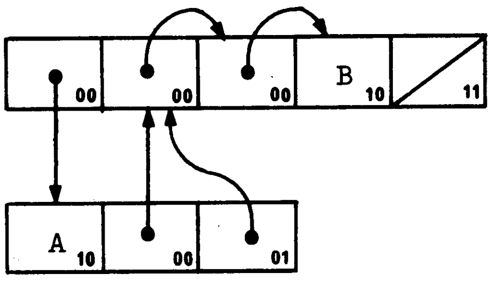
List with common subelements: The example below would print as

((A (((B) B) (B) B) ((B) B) (B) B) ((B) B) (B) B)

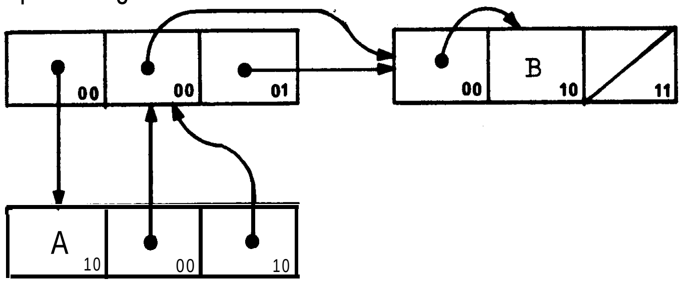
but note that B occurs exactly once in all representations of the structure.



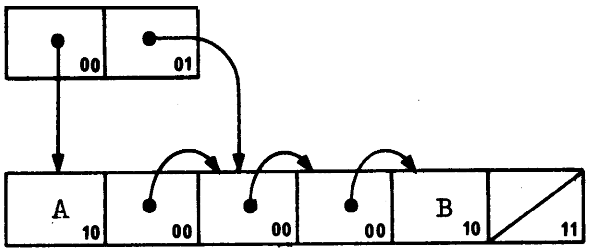
or



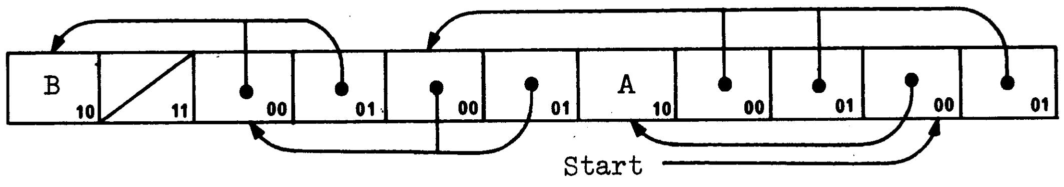
Lists may use any mixture of adjacency and list continuation elements. The last example might also be



or even



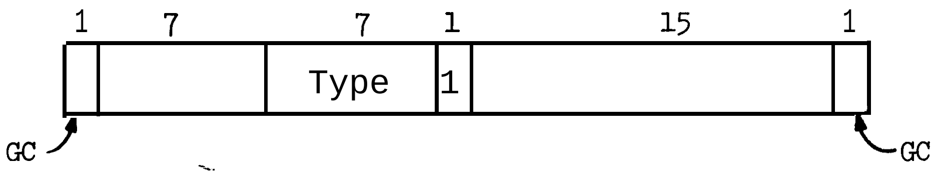
The garbage collector would rearrange this structure to occupy memory as:



b. Plexes (or atoms)

Two types of plexes have been implemented: one similar to the LISP 1.5 atom, the other a variable length string. Other types may be implemented as required by an application. All plexes have a plexhead aligned on a full word boundary; a pointer at a plex points six bytes in front of the first byte of this plexhead. This offset ensures that the atom bit is on in a pointer at a plex and thus distinguishes between pointers at lists and pointers at plexes.

The standard fields of a plexhead are



GC. These two bits are reserved for the garbage collector.

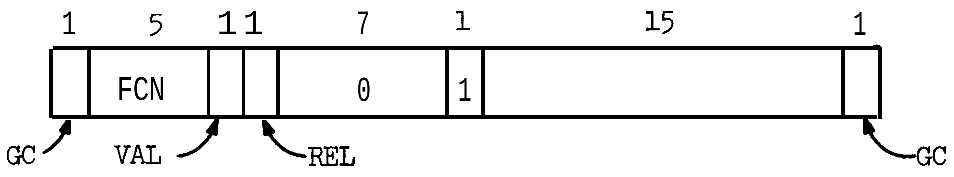
1 in bit 15. This bit, in conjunction with the offset addressing of plexes forces the RST routine to make a specification error if its argument is a pointer at a plex.

TYPE. This field distinguishes between different plex types. Currently types 0 and 1 are implemented.

The blank fields may be defined for individual plex types.

Plex Type 0 - Symbol (LISP atom)

This plex is a three part entity: plexhead, value cell, and property list. The plexhead has the format



VAL. If this bit is a one, the atom is bound to the value currently in the value cell. If 0, the atom's function definition is in the value cell.

REL. If this bit is a one, the contents of the value cell are relocatable, that is, the garbage collector will treat them like a pointer.

FCN. If the atom is not a function name this field is zero. Otherwise, this field encodes what type of function definition exists. The coding is

1	SUBR
2	FSUBR
3	EXPR
4	FEXPR

The fifteen bit blank field can be used as required. It is proposed to use these bits as marker bits indicating the presence or absence of properties on the property list.

Thus routines could find out if the indicator were present without searching the property list. Also the extra bits can be used to replace the "flag" feature of Lisp 1.5.

The atom's value cell is the next word after the plexhead. This cell -holds the current binding of the atom, that is, the value that is to be returned for EVAL of this atom. There is 8 unique string atom with the printname 'UNBØUND', that is only pointed at by value cells. If an atom has no function value and is not bound, the value cell points at 'UNBØUND'. When EVAL finds an atom with this value an error is indicated and control returns to the top level. If an atom has 'UNBØUND' in its value cell, VAL and REL are both one, because the atom is bound to a relocatable value. Note that given a pointer at the atom, the value cell can be addressed directly.

This means that **no** searching must be done to find the value of a routine's argument. Normally, when the STUTTER interpreter is running, the Value cell contains 8 relocatable value, a pointer at either 8 list or another atom. Provision is made, however, for compiled functions to store non-relocatable quantities in the value cell. This means that compiled functions can, indeed, do direct numeric computation.

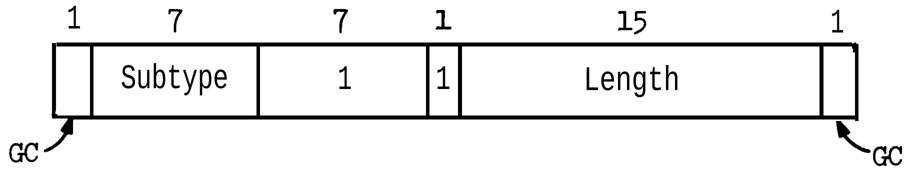
If an atom is not currently bound, the value cell may instead contain the function definition of that atom. For FEXPR and EXPR, the REL bit is on and the value cell points to the list defining the function. For SUBR and FSUBR, the REL bit if off, and the value cell contains the entry point of the subroutine. Since function names are not usually variable names, the interpreter normally does very little searching to find function definitions. Regardless of where the function definition is stored, bits are set in the atomhead to indicate what kind of definition it is; that is EXPR, FEXPR, SUBR, FSUBR. Thus when the definition is sought on the property list only the correct indicator need be used.

The property list of an atom is a standard Swym list, except that the r...rst is not NIL, but a pointer to the printname of the atom (a character string atom (type 1)). There is no PNAME indicator. The first word of the property list is the word after the atom's value cell. If there is no property list, the word following the value cell is a pointer to the print-name with the RST bit on. By convention, the property list always consists of indicator value pairs; there are no flags as there are in LISP 1.5.

GET, PUTPROP, REMPROP, and EVAL all obey the above conventions for the value cell and the property list. BINDERY, however, will not bind a value to an atom having a function definition. See the description of BINDERY in Appendix D.3.

Plex Type 1 - Strings

This plex type illustrates Swym variable length plexes. The plex format is



LENGTH. Number of bytes in string. String is right padded to occupy an integral number of full words.

SUBTYPE. This describes further the type of string. Currently, it affects only the print routine. Three subtypes are defined:

0	character string
4	fixed point number
8	hexadecimal number

Fixed point numbers are restricted to length four.

A.2. Stack Structures

The garbage collector must be able to scan the stack collecting those structures which are currently active. Thus, it must be possible to distinguish pointers from numbers and other random bit patterns. The high and low order bits of each stack word are used for this purpose and are interpreted as:

00	pointer	(collected)
01, 11	return address	(not collected)
10	stack plex	(collected by special routine)

Any non-relocatable information which may have a zero low-order bit must be stored on the stack in a stack plex. A plex head is stored after the plex on the stack because the garbage collector scans the stack from latest entry to earliest. The stack plexhead format is:

1	23	7	1
1		Type	0

TYPE. Determines what type of plex this is. The garbage collector invokes an appropriate type dependent routine. Two types of stack plex are implemented: the non-relocatable plex and the binding plex.

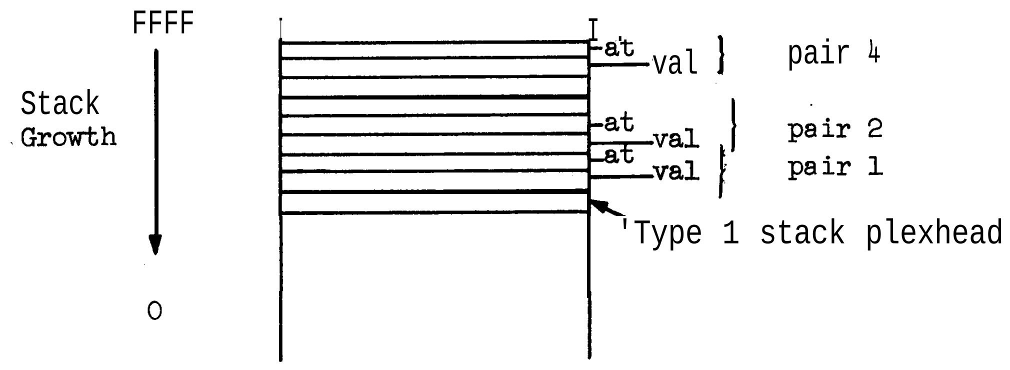
Stack Plex Type 0 - non-relocatable

1	15	8	7	1
1		Length	0	0

LENGTH. This many prior words in the stack are non-relocatable. They are ignored by the garbage collector.

Stack Plex Type 1 - Binding

This type of plex is used by BINDERY to store the old bindings of atoms before changing them. The plex must be removed from the stack by UNBIND for proper stack synchronization. Bindings are stored in atom-value pairs, thus the stack binding plex looks like-



The plexhead format is:

1	15	8	7	1
1	reloc bits	length	1	0

LENGTH number of pairs in plex.

RELOC BITS These define the relocatability of the value member of each pair. Bit 15 corresponds to pair 1. If the bit is on, the value is relocatable, that is, it must be collected. Up to fifteen pairs with a relocatable value may be stored. The atom pointers are always assumed to be relocatable.

Appendix B. SWYM Macros

An essential factor in the development of the Swym system was the creation of a collection of macros. In effect, these macros create a machine suitable for processing Swym data structures. The operands to most macros are register names, therefore a knowledge of Appendix I, "Swym Register Assignments", will be useful. For purposes of description, the macros have been divided into eight classes. An index indicates the class to which an individual macro belongs. The classes are

1. LISP - The Basic LISP Operations.

   FST, RST, NULL, ATOM, RPLF, EQ

2. Atom - Operations on Atom Fields.

   CELL, RPLCEL, HEAD, TAIL, RPLHD

3. Freest - Free Storage Creation.

   STRAT, MATOM, SUBR, FSUBR, CHAR, QCHAR, VALUE

4. Stack - Stack Manipulation.

   PUSH, POP, POPN, TOP, TOPN, RPLTOP, RPLTOPN

5. Bit - Named-bit Operations.

   BIT, SETBIT, RESETB, INVERTB, TESTB

6. Link - Subroutine Linkage.

   SUB, RET, CAL, TVMAK, XB

7. Control - Flow of Control.

   IF, THEN, ELSE, ENDIF, AND, ORX, NOT, BCMAC, GOTO

8. Misc - Miscellaneous

   CHTBL, SWEAR, INST4, GCPUT, FIXUP

Also in the Swym macro library is a piece of code which must be COPY'ed during a Swym assembly. Called SWYM, this code is described in Appendix M.

Unless otherwise indicated, the label field of a macro is attached to the first executable instruction.

MACRO INDEX

Macro	Class	Number of Positional Operands	Keyword Operands
AND	Control-7	0	
ATOM	LISP-1	1	TGØ, FGØ
BCMAC	Control-7	0	TBR,FBR,TGØ,FGØ
BIT	Bit-5	1	
BITTBLMK	Bit-5	0	
CAL	Link-6	2	P,B,S
CEIL	Atom-2	2 →	
CHAR	Freest-3	SYSLIST	
CHTBL	Mist-8	SYSLIST	
ELSE	Control-7	0	
ENDIF	Control-7	0	
EQ	LISP-1	2	TGØ,FGØ
EVCH	Freest-3	1	
FINDBIT	Bit-5	1	
FIXUP	Misc-8	2	
FST	LISP1	2 4	
FSUBR	Freest-3	SYSLIST	
GCPUT	Misc-8	1	
GETNAME	Atom-2	1	
GETNUM	Atom-2	2 4	
GØTØ	Control-7 .	1	

Macro	Class	Number of Positional Operands	Keyword Operands
. HASH	Freest-3	1	-
HEAD	Atom-2	2 4	-
IF	Control-7	0	-
INST4	Mist-8	3	
INVERTB	Bit-5	2	ATHD
MATØM	Freest-3	3	
NØT	Control-7	0	
NULL	LISP1	1	TGØ,FGØ
ØRX	Control-7	0	
PØP	Stack-4	1	P
PØPN	Stack-4	2	P
PUSH	Stack-4	1	P
QCHAR	Freest-3	SYSLIST	
RESETB	Bit-5	2	ATHD
RET	Link-6	1	R, E, P, B
RPLCEL	Atom-2	2 ←	
RPLF	LISP-1	2 ←	-
RPLHD	Atom-2	2 ←	
RPLT#P	Stack-4	1	P
RPLTOPN	Stack-4	2	P
RST	LISP-1	2 →	-
RSTMAK	LISP-1	1	
SETBIT	Bit-5	2	ATHD
STRAT	Freest-3	1	
SUB	Link-6	0	R,E,P,B

Macro	Class	Number of Positional Operands	Keyword Operands
SUBR	Freest-3	SYSLIST	-
SWEAR	Misc-8	1	-
TAIL	Atom-2	2 →	-
TESTB	Bit-5	2	TGØ,FGØ,ATHD
THEN	Control-7	0	-
TØP	Stack-4	1	P
TØPN	Stack-4	2	P
TVMAK	Link-6	SYSLIST	-
VALUE	Freest-3	2	-
XB	Link-6	1	-

NOTES:

1. The number following class name is the section number of that class in this appendix.

2. →. Both arguments must be register names. If this macro has one argument, it computes the function of that argument and assigns the value back to that argument, If a second argument is supplied, the value is assigned to this second argument and the first argument is unaffected.

3. ←. Always has two arguments. Value of second is stored in location referred to by first.

4. SYSLIST. The &SYSLIST(i) feature is used to reference up to 256 arguments.

B.1. LISP - The Basic LISP Operations

FST, RST, ATØM, NULL, EQ, RPLF, RSTMAK

FST a,b. (This is the LISP 1.5 CAR). a and b must be register names. FST finds the first element of the list pointed at by a. If b is present, the result is placed in register b, otherwise, the result is placed back in register a. Assembles as either

L a,0(a) or L b,0(a).

RST a,b. (This is the LISP 1.5 CDR). a and b must be register names. RST finds the list formed by deleting the first element from the list pointed at by the register a. The result is placed in b if present, otherwise in a. Assembles as either BAL L,RSTxx where xx is a or LR b,a; BAL L,RSTxx where xx is b. The routine RSTxx is created by the macro RSTMAK. In the current swym system there exist RSTA1, RSTA2, RSTA3, RSTT, and RSTTT; these are the only registers whose RST can be taken. Note that if b is specified, it must be among A1, A2, A3, T, TT while a need not be. If b is not specified, a must be among that restricted set.

ATØM a,TGØ=tgo,FGØ=fgo. This is a predicate macro; see section 7 of this Appendix, especially the description of BCMAC. a must be a register name; its contents are tested to see if they point at a plex (or atom). The code generated is

```
LA TT,2
NR TT,a
BCMAC TBR=BM,FBR=BZ,TGØ=tgo,FGØ=fgo
```

Note that ATØM destroys the contents of register TT.

NULL a,TGØ=tgo,FGØ=fgo. This is a predicate macro; see section 7 of this Appendix, especially the description of BCMAC. a must be a register name; its contents are tested to see if they point at the atom NIL. The code generated is

```
          CR a,N
          BCMAC TBR=BE,FBR=BNE,TGØ=tgo,FGØ=fgo
```

EQ a,b,TGØ=tgo,FGØ=fgo. This is a predicate macro; see section 7 of this appendix, especially the description of BCMAC. a and b must be register names. They are tested to see if they both point at the same identical entity. The code generated is

```
          CR a,b
          BCMAC TBR=BE,FBR=BNE,TGØ=tgo,FGO=fgo
```

RPLF a,b. (This is the LISP 1.5 RPLACA). a and b must be register names. The list structure pointed at by a is modified so that the first element of the list is the structure currently pointed at by b. Neither a nor b is changed. The code is ST b,0(a).

RSTMAK a. This macro generates the routine needed by the RST macro. Note that this routine must appear in an addressable section when a RST calls it. The code generated is

```
RSTa      TM     7(a),X'1'      is there a RST bit?
          BØ     RSTLDa         yes, branch
          BXH    a,C4,0(L)      no, incr ptr and return
RSTLDa    L      a,4(a)         load list cont ptr
          BCTR   a,L            remove RST bit and return
```

B.2. Atom - Operations on Atom Fields

HEAD, RPLHD, TAIL, CELL, RPLCELL; GETNAME, GETNUM

HEAD a,b. a and b must be register names. Accesses the plexhead of the atom pointed at by a. If b is present result goes in b, otherwise into a. Result is a bit pattern and is not relocatable. a may be a pointer at any plex (not just type 0). Assembles as L a,6(a) or L b,6(a).

RPLHD a,b. a and b must be register names. a must point at a plex. b should contain a bit pattern which is a valid plexhead. The result is that the plexhead pointed at by a is replaced by the contents of b. The contents of a and b are not changed. Assembles as ST b,6(a).

TAIL a,ba and b must be register names. a must point at a type 0 atom (not checked). The result is a pointer to the property list of the atom. If b is specified the property list is put in b, otherwise a. Assembles as LA a,10(a); RST a or LA b,10(a) ; RST b. Note that the restriction to RST applies to the last argument of TAIL.

CELL a,b. a and b must be register names. a must point at type 0 atom (not checked). The result is the contents of the value cell of a (not a pointer to the value cell). If b is specified, the value cell is placed in b, otherwise in a. Assembles as L a,10(a) or L b,10(b).

RPLCEL a,b. a and b must be register names. a must point to a type 0 atom (not checked). The value cell of the atom is replaced by the contents of b. Assembles as ST b,10(a).

GETNAME a,b. a and b must be register names. a must point at type 0 atom (not checked). The result is a pointer to the printname of the atom. If b is specified, it receives the result, otherwise a receives the result. Assembles as

```
LA a,10(a)          or   LA b,10(a)
RST a               or   RST b
ATØM a,FGO=*-10     or   ATØM b,FGØ=*-10
```

GETNUM a,b. a and b must be register names. a must point at a type 1 plex of subtype 4, that is, a string which is a fixed point number. This is not checked. The result is the value of the fixed point number. If b is specified the result replaces b, otherwise a. Assembles as L a,10(a) or L b,10(a).

B.3. Freest - Free Storage Creation

VALUE, SUBR, FSUBR, CHAR, QCHAR; MATØM, STRAT, HASH, EVCH

There are three levels of free storage creation macros. The highest level macros create atoms with properties required for the interpreter: VALUE, SUBR, FSUBR, CHAR, and QCHAR. These macros call on MATØM to actually create an atom. The third level macros are called by MATØM as utilities: HASH, EVCH, and STRAT,

In addition to assembling the structure required for an individual atom, these macros create the object list and the character objects list, These lists are the values of ∅BLIST and CHAR∅BS, respectively, as described in Appendix H.

The macro MAT∅M takes care of creating the ∅BLIST. Each time an atom is created using MAT∅M, the print name is hashed (using the HASH macro), and a bucket link is created. Created labels are used to link the members of a bucket together. These labels have the form

```
     BUC xx L nn
```

where xx is the hash number and nn is the number of the items in the bracket. Thus the oblist itself is

```
     ∅BLIST   DC     A(BUC1LO)
              DC     A(BUC2LO)
               .
               .
              DC     A( BUC64LO)
              DC     A(NIL+RBIT)
```

When an atom is created, two words are created to link the atom in the porper bucket. They are

```
     BUCxxLn  DC     A(atom)
              DC     A(BUCxxLm+RBIT)
```

where xx is the bucket number, n is the number of items already in this bucket, m is n+1, and A(atom) is a pointer at the atom. RBIT has the value 1 and is used to put in the RST bit where required.

The initial contents of free storage are discussed in Appendix H.

<u>nm</u> VALUE <u>pname</u>,<u>val</u>. The structure for one atom is created. The label <u>nm</u> is given the value by which the atom should be addressed. The printname is <u>pname</u>. The value cell is a pointer to <u>val</u>. The plexhead is marked to indicate that there is a quantity in the

value cell and that it is relocatable. The assembly is performed by calling

```
nm       MATØM pname,RELB+VALB,A(val)
```

RELB and VALB are equated to the bits REL and VAL (see Appendix A.1.b).

SUBR pname1, pname2, . . . , pnamen. An atom is created for each pname in the list. The printname is pname and the value cell is the address of the SUBR with that name. The atom head is marked to indicate that the atom has a function definition, it is a SUBR, and the address of the routine is in the value cell. The pname is declared EXTRN to communicate with the assembly in which the SUBR is defined. For each pname on the list, the code assembled is:

```
         EXTRN pname
         MATØM pname,SUBRB,A(pname)
```

SUBRB is equated to 1, the function definition code for SUBR's. Any label field on SUBR is ignored.

FSUBR pname1, pname2,..., pnamen. Same as SUBR, but FSUBRB is used instead of SUBRB.

CHAR char1, char2,..., charn. An atom is created for each character in the list of characters. The print name is just the character. The value cell is set to point at the 'UNBØUND' error atom. The plexhead bits are set to indicate that the value cell is relocatable and has a value. In addition, the appropriate entry in CHARØBS is set to point to the created character atom. Each atom is created by

```
         MATØM chari
```

If there is a label on the CHAR it will be equated to the atom for the first character on the list.

The following characters are valid arguments to CHAR:

A-Z, 0-9, blank, and these special characters

+ | $ 1 / % ? : # " ¢ ! $\begin{bmatrix}0\\2\\8\end{bmatrix}$ * = _ ⟨ ⟩ @ - . ;

note that $\begin{bmatrix}0\\2\\8\end{bmatrix}$ prints as -, while ¢ and ! print as blank.

QCHAR. Same as CHAR, but expects arguments to be quoted: viz. '(',)", and ','.

nm MATØM pnm, celbits, plist. Creates an atom with the print pnm. The label nm is equated to the offset address of the atom's blockhead. If celbits and plist are not specified, the atom head is marked to indicate relocatable binding and the value cell is a pointer at the special atom 'UNBØUND'. If celbits are specified, that quantity is assembled (AL1 (celbits)) as the first byte of the atom head. The rest of the atom head is 010000; indicating a normal type 0 atom. The members of plist-which may be a 360 assembler sublist - are assembled following the atom head. Thus the first element of plist is the contents of the atom's cell. Other elements of plist must be in indicator-value pairs for the property list. After the property list, a pointer to the printname and the printname itself are assembled. The code assembled for missing celbits and plist is

```
BUCxxLn  DC     A(* + 8 - AT)                put atom in bucket
         DC     A(BUCxxLm+RBIT)              link to nxt bucket item
nm       EQU    *-AT                         equate name to atom ptr
         DC     AL1(RELB+VALB),X '010000'    assemble atom head
         DC     A(UNBØUND)                   value cell
         DC     A(*+4-AT + RBIT)             null prop list is ptr
         STRAT  'pnm'.                       print name
```

where

xx	is hash code of <u>pnm</u>,
n	is number of prior entries in bucket xx,
m	is n+1,
AT	is atom offset (6),
RBIT	is rst bit (1),
RELB+VALB	put in relocatable and variable-bound bits.

The code generated with <u>celbits</u> and <u>plist</u> is the same except the atom head is

```
        DC    AL1(celbits),X'010000'
```

and the elements of <u>plist</u> precede DC A(*+4-AT+RBIT).

<u>nm</u> STRAT '<u>string</u>'. Creates a string atom (type 1). The atom head is

```
        DC    X'0003',AL2(2*L'string)
```

which indicates character string atom. Following words are four character at a time chunks of <u>string</u>. <u>nm</u> is equated to the offset location of the string atom. That is, the first assembled instruction would be <u>nm</u> EQU *-AT. String atoms are not placed on the ØBLIST or CHAROBS.

BASH <u>string</u>. HASH evaluates the hash function for the <u>string</u>. The result is left in an assembly time global variable (GBLA &HVAL) whose value can be used by a calling macro. BASH calls on EVCH three times to evaluate the three character values needed by the hash function (first, third, and last).

EVCH ch. Unbelievably with 360 assembler, there is no simple way to determine from a character the number corresponding to that character's EBCDIC code. EVCH performs this feat by a large test:

```
chval = if ch = 'A' then 193
           else if ch = 'B' then 194
           else if ch = 'C' then 195
           ...
           else if ch = 'Z' then 233
           else if ch = '0' then 240
           else
           .
           else error ('illegal character - evchr')
```

The value is left in a global variable (GBLA &CHVAL) whose value can be used by a calling macro, for instance HASH. The following characters are valid to EVCH: A-Z, 0-9, blank, comma, >,), period, <, (, /, +, , !, $, *, ;, ¬, -, ¢, %, __, ?, i, #, a, =, ", $\begin{bmatrix}0\\2\\8\end{bmatrix}$.

B.4. Stack - Stack Manipulation

PUSH, PØP, PØPN, TØP, TØPN, RPLTØP, RPLTØPN

The stack is allocated in units of one word. The basic macros are PUSH and PØP. The former puts one word on the stack, the latter removes a word from the stack. A routine must do exactly as many PUSH'es as PØP's unless very special care is taken.

Swym stack macros use negative stack growth. That is, the first stack location allocated is the highest address and successive words are in successively lower locations. This means that since the stack pointer points at the last entry on the stack, all recent entries to the stack can be addressed with simple displacement addressing. Thus a routine may do three PUSH'es to allocate three words of temporary storage; then it can address all three locations.

A Swym stack pointer must be in a register when the stack is referenced by a stack macro. The standard Swym stack is always pointed at by register P. All stack macros have a keyword parameter "P=". If P= is omitted, P=P is assumed.

Currently, no check is made for going off either end of a stack. Several techniques are possible to ensure that other storage is not destroyed or that too many PØP's are not executed. The simplest is to generate code to check the stack pointer at each PUSH and PØP. This is time consuming and inelegant. An elegant method would be to use a PDP-6 which has hardware PUSH and PØP with built-in checking. (Unfortunately, the 360 does not have PDP-6 mode). It is proposed for Swym that the stack be first in the user partition. When the stack is exhausted, a protection interrupt will terminate the computation.

All stack macros except PUSH have an 'N' form, indicated by N at the end of their name. The first argument to the N-form is a number in the range 1-1024. The action of the macro takes place but rather than affecting the top of the stack, it affects the Nth element of the stack. The latest entry on the stack is N=1. Thus xxxN 1,y is equivalent to xxx y although different code may be generated.

PUSH r,P=p. r may be the name of a register or a sublist of register names. If the former, as in

PUSH A1

then the stack pointer (P since none other is indicated by P=) is incremented and the contents of Al are stored on the new top of the stack. If a sublist is coded

```
PUSH (Al, A2, Al, T, A4)
```

then the required number of locations are allocated on the stack and the named registers are placed on the stack. The last named register is at the top of the stack. The first named register is the first placed in the stack. Note that in the example, Al is placed in the stack twice. A PØP TT immediately following the example will put the old contents of A4 into TT. The code generated for each element of the sublist r is

```
SR  p,C4
ST  r,0(p)
```

where C4 is a register whose contents are always 4.

PØP r,P=p. Like PUSH, r may be simple or a sublist. If simple, then the top element of the stack is placed in the named register and the stack pointer is decremented. If a sublist,

```
PØP (Al, A2, Al, T, A4),
```

then the registers are filled in the reverse order from PUSH. That is, the right thing happens and this example will exactly restore the contents of the registers as stored by

```
PUSH (Al, A2, Al, T, A4)
```

The code generated for each element of the sublist r is

```
L  4,0(p)
AR p,C4
```

where C4 is a register whose contents are always 4.

PØPN n,r,P=p. The nth element of the push down list p is popped into register r. Also the stacked is popped so that the current (n+1)st element of the push down list is the new first element. The current top of the stack is n=1. The code generated is:

```
L  r,4*(n-1)(p)
LA  p,4*n(p)
```

TØP r,P=p. The first element of push down list p is put in register r. The code generated is

```
L  r,0(p)
```

TØPN n,r,P=p. The nth element of push down list p is put in register r. The code generated is

```
L r,4*(n-1) (p)
```

RPLTØP r,P=p. The first element of push down list p is replaced by the contents of register r. The code generated

```
ST  r,0(p)
```

RPLTØPN n,r,P=p. The nth element of push down list p is replaced by the contents of register r. The code generated is

```
ST  r,4*(n-1)(p)
```

B.5. Bit - Named-Bit Operations

BIT, SETBIT, RESETB, INVERTB, TESTB; BITTBLMK, FINDBIT

nm BIT bitno. Using this macro defines nm for all the other bit macros. nm is defined as being the bitnoth bit of a word. Because all the other functions use SI instructions both the bit within a byte and the byte within a word must be stored for each BIT declared. The former is stored by equating nm to

BITTBL(bitno-bitno/8*8+1)

where BITTBL has the quantities

X'80', X'40', X'20', X'10', X'8', X'4', X'2', X'1' .

The byte within a word is stored in an assembly time array (GBLA &BITS (64)). It is computed by bitno/8. A corresponding array (GBLC &BITNAMS(64)) contains the name of the bit so table lookup can be performed.

SETBIT r,bit,ATHD=T. This macro sets a bit in a word in memory. r must be the name of a register. The register will be assumed to point to the required word. bit must be the name of a bit declared with the BIT macro. If the ATHD=T parameter is present, the pointer in r is assumed to point at a plexhead and the pointer is suitably adjusted. The code generated is ØI bl(r),bit or ØI bl+AT(r), bit. In either case, FINDBIT is used to find the value bl, the byte-within-the-word for bit.

RESETB r,bit,ATHD=T. Same as SETBIT but turns the bit off by using NI bl(r),X'FF'-bit or NI bl+AT(r),X'FF'-bit.

INVERTB r,bit,ATHD=T. Same as SETBIT but complements the bit by using XI bl(r),bit or XI bl+AT(r),bit.

TESTB r,bit,ATHD=T,TGO=tgo,FGO=fgo. This is a predicate macro; see section 7 and especially the BCMAC macro. The word pointed at by register r is tested to see if bit bit is on. If it is control goes to label tgo, if not control goes to label fgo. TGØ or FGØ or both may be specified. The omitted condition will simply drop through. Between IF and THEN, both TGØ and FGØ may be omitted. If ATHD=T is specified, r will be assumed to point at a plexhead and the appropriate offset will be assembled. The code assembled is either

```
        TM    bl(r),bit
        BCMAC TBR=BØ,FBR=BZ,TGØ=tgo,FGØ=fgo
or      TM    bl+AT(r),bit
        BCMAC TBR=BØ,FBR=BZ,TGØ=tgo,FGØ=fgo
```

The macro FINDBIT is used to compute bl, the byte-within-the-word for bit.

BITTBLMJS. This macro is called exactly once at the beginning of an assembly to create the array BITTBL used by the macro BIT. It stores these character strings into the elements of BITTBL:

X'80', X'40', X'20', X'10', X'8', X'4', X'2', and X'1' .

The name field and any arguments are ignored. No code is assembled. (BITTBLMK is coded in the GSSWYM control section. See Appendix M.)

FINDBIT bit. This macro finds the byte-within-the-field for the bit named bit by a BIT declaration. The result is left in a global variable (GBLA &BITLØC) for use by the calling macro (SETBIT, RESETB, INVERTB, or TESTB). The name bit is looked up in the array BITNAMS created by BIT. Corresponding to the entry for bit is an entry in the array BITS giving the correct byte-within-the-field. No code is assembled.

B.6. Link - Subroutine Linkage

SUB, RET, CAL, TVMAK, XB

Subroutine linkage occurs at three points: the calling point, the entry point, and the exit point. Swym has a macro for each point. Note that for a given routine the entry point and exit point occur within that routine, but the calling point occurs wherever some routine calls that given routine.

The basis of Swym subroutine linkage is a table of transfer vectors which is always addressable via register S. This table contains the address of each routine which can be called by any routine in another module or by compiled functions. Entries in the table are created by the TVMAK macro. TVMAK may also be used within a module to address routines used only within that module.

Two conventions are assumed for subroutines. First, registers must be saved by the calling program if it expects them to be saved. Second, the entry point to a routine is the first byte of code and a base register will contain that address during execution of the routine.

Three standard registers are vital to subroutine linkage:

S	Swym base, base for transfer vectors

B	base for all routines; must be loaded by calling routine
P	push down list pointer.

<u>nm</u> SUB R=NØ,E=NØ,P=<u>p</u>,B=<u>b</u>. This macro assembles subroutine entry code. The parameters supplied should be identidal to the parameters supplied for any corresponding RET macros. SUB must occur exactly once and then only at the beginning of the subroutine it defines. . The normal case has no parameters coded. If R=NØ is coded, the routine will not be recursive; that is, it will not push its return address onto the stack. If E=NØ is coded, the subroutine name <u>nm</u> will not be ENTRY'ed. In this case, no other module may refer to the routine and a TVMAK for it must be included in its own module. The P= parameter determines onto which push down list the return address will be pushed. <u>p</u> must be a register name. If omitted, the standard push down list pointed at by register P is used. <u>b</u> must be a register name. It is the base register declared for this routine. If omitted the standard base register B is assumed. The standard case of no parameters generates:

```
         USING  nm,b
         DC     C'nm'     supplied for debugging
         ENTRY  nm
nm       BCTR   L,0       make odd so GC ignores
         PUSH   L, P=p
```

If R=NØ is coded, the last two lines are replaced by <u>nm</u> DS OH.

RET <u>nm</u>, R=NØ,E=NØ,P=p,B=<u>b</u>. This macro assembles subroutine exit code. The <u>nm</u> parameter must be the name on the nearest preceding SUB. The other parameters must be the same as for that SUB. If R=NØ is coded, the pushdown list is not popped and the return address is assumed to be in register L. p is the register name of the push down list pointer; if P=p is omitted, the standard push down pointer register pointer P is assumed. b is assumed to be the name of the base register of the current routine; if omitted, the standard base register B is assumed. The standard case is with only nm specified. The code assembled is

```
          POP  b,P=p
          B  1(b)
```

If R=NØ is coded, the code is

```
          BR L
```

CAL nm,regs,P=p,B=b,S=YES. This macro assembles subroutine calling code. nm is the name of the routine to be called. It is also possible to specify registers to be saved before the call and restored afterward. The operand regs may be any name or sublist acceptable to the PUSH and POP macros. p is the push down pointer for the register saving; normally P is assumed. b is the name of the base register for the routine nm and for the current routine (last SUB). If B=b is omitted, the standard base register B is assumed. If S=YES is coded, no base register is loaded after return, the assumption being that the current routine is addressable via some preserved register. With S= omitted, the code generated is

```
          PUSH regs,P=p     if regs specified
          L     b,#nm
          BALR  L,b
```

```
            L       b,#self
            POP     regs, P=p      if regs specified
```

#nm is the label of the address'of routine nm in the transfer vector table, #self is the label of the address constant for the current routine. The name self was the name on the most recent SUB macro.

TVMAK nm1, nm2, . . ., nmn. This macro creates entries in the transfer vector table. One entry is created for each element in the list. The label on the entry is created by concatenating a "#" on the front of the first seven characters of nmi. If nmi is not defined in the current assembly, it is EXTRN'ed. This decision is made on the basis of the type attribute of nmi. Care must be taken that nmi is not the label on EQU. (That pseudo-op gives its label the type attribute 'U'). The code generated for each entry is

```
            EXTRN   name     if required
#name       DC   A(name)
```

XB rtn,label. This macro is provided for jumping into the middle of some other routine. Because this is considered evil, XB generates an MNOTE statement which goes into the error listing. XB does not modify the stack; this.must be accomplished by RET in rtn. The second argument may be omitted and the code generated is:

```
            L       B, #rtn
            B       8(B)
```

#rtn is the label of the transfer table entry for rtn. Execution of rtn begins just after its SUB macro (which must not specify R=NØ).

If the second argument is specified, label must appear somewhere in rtn and rtn must be assembled in the current module. Control is transferred to label in rtn by the code:

```
        L       B,#rtn
        B       label-rtn(B)
```

B.7. Control - Flow of Control

IF, THEN, ELSE, ENDIF; AND, ØRX, NØT; BCMAC, GØTØ

There are three groups of control macros. IF, TEEN, ELSE, and ENDIF must occur in that sequence; they avoid many user generated labels, AND, ØRX, and NØT may occur only between IF and THEN. BCMAC and GØTØ generate branch instruction; the former conditional, the latter unconditional.

The macros in the first two groups ignore any arguments. Instead they affect the flow of control to the code between them. The primary purpose of these macros clarify what code is executed under what conditions.

The key to the flexibility of the IF-THEN-ELSE is BCMAC and the concept of predicate macros. A predicate macro calls on BCMAC to assemble a conditional branch to a label depending on the context. Predicate macros need not supply branch labels if they occur between IF and THEN because BCMAC uses labels generated by the preceding IF. Currently, the predicate macros are ATØM, NULL, EQ, and TESTB.

IF, THEN, ELSE, ENDIF. There are two forms: IF-THEN-ENDIF and IF-TEEN-EISE-ENDIF. The expression IF-THEN-ELSE will mean-both.

The first form may be represented

```
         IF
               predicate-part
         THEN
               true-part
         ENDIF
```

The code generated is

```
               predicate-part
THENx    EQU  *     (if ØRX occured in predicate-part)
               true-part
ELSEy    EQU  *
```

where x and y are unique four digit numbers, The IF macro generates the labels THENx and ELSEy and stores them on an assembly-time global stack. Predicate macros in the predicate-part simply test for the falsehood of the predicate and branch to the ELSEy on top of the stack. ØRX and NØT in the predicate-part modify the action of BCMAC so that the desired result is accomplished (see the descriptions of those macros).

The second form may be represented

```
         IF
               predicate-part
         THEN
               true-part
         ELSE
               false-part
         ENDIF
```

The code generated is

```
                  predicate-part
      THENx  EQU  *    appears only if ØRX is in predicatepart
                  true-part
             B    DONEz
      ELSEy  EQU  *
                  false-part
      DONEz  EQU  *
```

where x, y, and z are unique four digit numbers. The label DONEz is created by the ELSE macro and stored atop the label stack. IF-THEN-ELSE's are permitted to nest (up to 60 levels). That is, they may appear in either the true-part or the false-part. But IF-THEN-ELSE is not permitted in the predicate-part.

AND, ØRX, NØT. The second group of flow of control macros may appear only in a predicate-part. They control the code generation in BCMAC.

NØT. This macro reverses the sense of any BCMAC occurring before the next AND, ØRX, NØT, or THEN. Two NØT's cancel eadh other. While NØT is in force, BCMAC makes tests for true and branches to the ELSEy on top of the label stack.

ØRX (not ØR because IBM used it). This macro makes tests parallel. It assembles the code

```
             B    THENx
      ELSEy  EQU  *
```

Also it turns off any outstanding NØT, sets an indicator so that THENx EQU * will appear, and creates an ELSEw (on the IF label stack) for subsequent false tests to branch to.

AND. The only action by AND is to turn off any outstanding NØT. But use of AND makes explicit the fact that all sequential tests must be met before the true-part is executed.

BCMAC TBR=tbr,FBR=fbr,TGØ=tgo,FGØ=fgo. This macro assembles one branch conditional instruction. If either TGØ or FGØ (or both) is specified, BCMAC assembles a branch to tgo, fgo or both. The operator for tgo is tbr; the operator for fgo is fbr. h fbr and tbr are assumed to exist. The code generated is

```
tbr  tgo     if only tgo exists
fbr  fgo     if only fgo exists
tbr  tgo  }  if both tgo and fgo exist
B    fgo  }
```

If neither tgo nor fgo exists, the BCMAC must occur in the predicate-part of an IF-THEN-ELSE. If NØT is not in force, the code generated is

```
fbr  ELSEx
```

If NØT is in force, the code generated is

```
tbr  ELSEx
```

GØTØ label. This macro assembles into a branch to label:

```
B    label
```

B.8. Misc - Miscellaneous

CHTBL, SWEAR, INST4, GCPUT, FIXUP

CHTBL loc{,what,where} .... (. . . indicates that ', what, where ' may be repeated up to 127 times). This macro is intended for creating character tables for the translate instruction (TR) and the translate and test instruction (TRT). As such, loc is assumed to be the address of a table. CHTBL then ØRG's into that table and puts values at the required places. For example, a TRT to scan for blanks might be written

```
BLTBL   DC    256X'00'
        ØRG   BLTBL + C' '
        DC    X'01'
```

This scheme is documentary in that the ØRG tells exactly where something goes, while the DC tells what that something is. Using CHTBL, the example might be written

```
BLTBL   DC    256X'00'
        CHTBL  BLTBL,1,C' '
```

The name field is ignored in call on CHTBL.

The loc field may be any expression. It will be assumed to be the beginning of a table 256 bytes long. The last instruction generated is an

```
        ØRG  loc+256
```

That what field may be either a decimal number or an argument for DC. In the first case, the macro generates DC FL1(what); in the second case, DC what. The cases are distinguished because a decimal number must be three or less characters and the general DC argument must be four or more.

The where field may be a (360 assembler) sub-list. Each element of the sub-list may be either a single character or a non-relocatable term. The latter must be more than one character. In the first case the macro generates

```
         ØRG   loc+C'where'
```

While the non-relocatable term generates

```
         ØRG   loc+where
```

The following example illustrates all of the above

```
HEXTBL   DC    256X'00'
         CHTBL HEXTBL,4,(A,B),4X'4',C, 10AL1(8),C'0'
```

will generate

```
HEXTBL   DC    256X'00'
         ØRG   HEXTBL+C'A'
         DC    FL1'4'
         ØRG   HEXTBL+C'B'
         DC    FL1'4'
         ØRG   HEXTBL+C'C'
         DC    4X'4'
         ØRG   HEXTBL+C'0'
         DC    10AL1(8)
         ØRG   HEXTBL+256
```

Note that using a sub-list for where can lead to large object module decks. (Each ØRG forces a new output card image).
Note also that good documentation requires that each what - where pair go on a separate continuation card.

SWEAR error-code. This macro generates a call on the STUTTER internal routine: SWERROR. The error-code must be two characters. These characters will be supplied as a character string to the error routine: ERROR. The code generated is

```
LH      L,*+8              load error-code in REG L
B       SWERRØR            go to system error routine
DC      C'error-code'
```

Note that SWERRØR is always addressable via register S.

INST4 op,r,rand. The purpose of this macro is to avoid the overly cautious assembler's "ALIGNMENT ERROR" message. This is done by assembling first the OP and R1 fields and then the B1-D1 field. The R2 field can not be used with this macro. Two forms are possible: r present

```
op    r, 0
ØRG   *-2
DC    S(rand)
```

r omitted

```
op    0
ØRG   *-2
DC    S(rand)
```

GCPUT type. . This is a special purpose macro for writing the garbage collector. It is called to place a word in new core. For further discussion see the routine GCPUT in Appendix E. The code generated depends on type.

type omitted

```
          BAL   L,GCPUT
```

type S T

```
          NR    TT,NØTM1
          BAL   L,GCPUTFUL
```

type =FULL

```
          BAL   L,GCPUTFUL
```

If some other type is coded, GCPUT assumes 'type omitted', but generates an error message.

FIXUP pt,new. This is a special purpose macro for the garbage collector. It makes an entry in the fixup table. pt and new must be register names. Register pt contains the address of a word in old core which will eventually contain a correct new core address. new contains a pointer to new core showing where to put the eventual contents of pt. Register FIXPTR points at the fixup table; so the code generated is:

```
          ST    pt,0(FIXPTR)
          ST    new,4(FIXPTR)
          LA    FIXPTR,8(FIXPTR)
```

Appendix C. READ Routines and Syntax

The READ routines convert a character string on an input medium into an internal plex structure. The syntax is similar to the LISP 1.5 syntax. The major innovation is the super-parenthesis. The parser guarantees that all regular parentheses within a pair of super-parentheses will match. The syntax is described in section C.l. A second section describes the internal routines. (External routines are described in Appendix F.) Section C.3 details the variables in CSSWYM used by the READ routines. Flow charts of the main READ routines are in the last section. All error codes are collected in Appendix J.

C.1. The Syntax

Input expressions are punched free-form in the first 71 columns of the input cards. Column 72 is used for the continuation as described in the paragraph on (string). Columns 73-80 are ignored. Column 1 of one card immediately follows column 71 of the preceeding card. Comments may be included; the characters'_/' are ignored and terminate scanning of a card. A card with under bar - slash in the first two columns is printed, but otherwise ignored. Allcharacters must be in the IBM 029 character code. The BNF of the syntax appears in figure C.l. The highest non-terminal is the s-expression, abbreviated (sexpr). The following paragraphs specify the semantics of selected syntactic types.

(super list). The less-than and greater-than characters bracket a (super list). When a greater-than is reached before all subordinate structures are terminated, parentheses are created as required to

Figure C.1

```
⟨sexpr⟩ ::= (list) | (super list) | ⟨atom⟩
(list) ::= ( ) | ( ⟨sexpr⟩    (list tail)
(list  tail)  ::= ⟨sexpr⟩ ) |.⟨sexpr⟩ ) |
                  ⟨sexpr ⟩    (list  tail)

(super list)  ::= < > | < ⟨sexpr⟩ (super list tail)
(super list tail) ::= ⟨sexpr⟩> | . ⟨sexpr⟩ > |
                 ⟨sexpr⟩ (super list tail)

⟨atom⟩  ::=  (symbol) | (string)

(symbol) ::= ⟨letter⟩ | (symbol) ⟨alpha-num⟩ |
   @ ⟨char⟩ | (symbol) @ ⟨char⟩

(string) ::= ⟨num string) | - ⟨num string) |
        Z ' (char string) ' | X' (hex string) '|
        W ' (bit string)'

⟨num string)  ::= ⟨num⟩ | ⟨num string) ⟨num⟩
⟨num⟩ ::=  0|1|2|3|4|5|6|7|8|9

(char string) ::=  (char) |''| (char string) ⟨char⟩|
    (char string) (char) ''

(hex string) ::= (hex digit⟩ | (hex string) (hex digit)
(hex digit) ::= (blank) | ⟨num⟩ | (hex letter)
(hex  letter)  ::= A|B|C|D|E|F

(bit string) ::=  0|1|0 ⟨bitstring⟩ | 1 (bit string)
                (blank) ⟨bitstring⟩ | ⟨bitstring⟩ (blank)
```

```
(other letters)  ::= G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

(letter)  ::= (hex letter) | (other letters⟩

(alpha-nun)  ::= (hex digit) | (other letters)

(char) ::=  (alpha-mm) | . | ( | ) | > | < | @ | - | + | | | $ | ; |
            ¬ | / | % | ? | : | # | " | ¢ | ! | 028 | * | = | , | ⟨blank⟩
            & | _
```

close all structures. When all internal structures are closed and an extra right parentiesis is encountered -- where a greater-than is expected -- characters are discarded until the matching greater-than is found, As will be seen from the flow chart, whole structures are discarded, so that the matching greater-than is found rather than just the next greater-than, (For example, '<)A<AO>()>' is parsed as 'NIL').

⟨list tail⟩. Note that a degenerate form of the ⟨list⟩ is the LISP 1.5 dotted pair, This syntax reflects the "general s-expression" form as supported by most LISP read routines,

⟨symbol⟩ . This is parsed into a type 0 atom, If a type 0 atom with the same string exists on the OBLIST, a pointer to that existing atom is returned; otherwise, a new atom is created, Note that '@' preceding any character causes that character to be treated as a letter. Only one character, the seoond, is stored in the created print name, For example, the (sexpr) @@ returns a pointer to the symbol atom with the one character print name '@'. This atom already exists.

⟨string⟩. Arbitrary string atoms may be input,, Both ⟨hex string⟩'s and ⟨bit string⟩'s are converted into hex string type string atoms internally. Numbers are currently always four bytes, but the other two classes may be up to 2^{15}-1 bytes. Hex strings are filled with zeroes from the right to make an integral number of' bytes. Floating point numbers are not defined so there is no dot ambiguity problem; however, this problem could be solved with F'... '.

Any string within quotation marks may be continued from one card to the next. Column one of the second card immediately follows column 71 of the preceding card. In this case, column 72 must contain a dash ('-'). Otherwise, column 72 must be blank. This convention was adopted from CØBØL in order to attack the quote mismatch recovery problem. This problem occurs if there is a missing or extra quote mark. Thereafter, everything which looks like it should be in quotes is outside and vice-versa. There is sufficient redundancy in the Stutter syntax for recovery at some later point. Because there was insufficient experience with thelanguage to have a feeling for reasonable recovery heuristics, the mismatched quote problem was not attacked other than to specify what should be an adequate syntax.

(blank). The general rule is that blanks may appear where they do no harm. They are only required to separate the strings representing symbol atoms. Blanks may appear between any two elements of the (list), (list tail), (super list), and (super list tail). More than one blank will be treated as a single blank except inside a (char string). Blanks may also appear within the quotes for (hex string) and (bit string).

(char).: In flow charts, two special characters are used: '␣' represents a single blank; '⊓' represents underbar.

C.2. Internal Routines

The routines described in this section are service routines available only within the read package. The routines available through the stutter interpreter are described in Appendix F. The entire CSREAD control section is reentrant. All temporary storage is in CSSWYM.

All read routines make use of three global bytes: RDSTAT, RDCHAR, and RDCLASS . These are described in Section C.3.

The get-a-character routine, GETCH, puts a single character into RDCHAR and puts the class of that character into RDCLASS. The class of a character is a number chosen to simplify distinctions like "Is this character possibly the first character of an atom?" The classes and their members are in figure C.2. RDCHAR can be set and tested by a STUTTER program with the functions STIVCCH and IVCCH. This can be important because the general rule is that the read routines interpret the character in RDCHAR and then read another character for the next routine to interpret.

The RDSTAT byte is composed of eight status bits. They are used to communicate between the various routines. One of these bits may be manipulated by a stutter program as an internal variable (STIVQMO, IVQMO). The defined bits are described in figure C.3.

The symbol NOCARDS also bears explanation. It is the address branched to when the input file is exhausted. The routine there provides for orderly termination of the job.

The remainder of this section is a discussion of each of the internal read routines:

Figure C.2

class	members	comments
0	0,1	bit string
4	2,3,4,5,6,7	octal string
8	8,9	number
12	A,B,C,D,E,F	hex string
16	G,H, . . . Z, @	alphanumeric
20	-	atom start
24	(,<	list start
28	blank	
32	.	dot
36	),>	list terminator
40	all other keypunch characters	

All non-keypunch characters are in class 255. They cause an error and are converted to blank before being processed.

Figure C.3

seton	setoff		interpretation
QUOMON	QUOMOFF	on:	GETCH passes each character in turn. '-' must appear in column 72.
		off:	if last char was blank., GETCH scans for non-blank. Column 72 must be blank. '_/' in two columns means ignore those characters and the rest of the card,
NEGNON	NEGNOFF	on:	detected -(num string) construct (used in RDAT)
GJFND	GJNFND	on:	GETOBJ found the symbol atom already on the OBLIST, RDAT releases any new storage allocated,
SKIPMON	SKIPMOFF	on:	skipping to find right super-paren. Used by RDSE when skipping to avoid recursive RO error messages.

A bit is set on with the instruction

```
01 RDSTAT,seton
```

The same bit is set off with the instruction

```
NI RDSTAT,setoff
```

error routines

RDERR, RDERRCNT

character fetching

GETCH

string construction

PBOPEN, PUTBYTE, PBCLOSE

recursive parser

RDSE, RDLIST, RDAT

RDERR. This routine prints a two byte error code. The code must be in the right half of register Al on entry. RDERR also prints a pointer indicating the last character scanned.

RDERRCNT. This routine prints a read error message by using RDERR. RDERRCNT's second argument is a number in A2. This number is printed at the far right of the RDERR message.

GETCH. This routine GETs one character from the current input card and puts it in RDCHAR; its class is put in RDCLASS. GETCH reads a new card when required and maintains two pointers - one to the current character, the other to the end of the card. Initially, both pointers are zero to force the reading of the first card. GETCH converts strings of blanks to a single blank by ignoring blanks if RDCHAR (the last character read) is blank. Illegal characters (not on keypunch) are converted to blanks. When quote mode (QUOMO) is on, all blanks are sent to the calling routine. The '_/' terminates scanning of a card unless QUOMO is on, in which case both characters are passed to successive GETCHes.

PBOPEN, PUTBYTE, PBCLOSE. While RDAT is scanning a character string, no TAK2's are performed. The character string for the atom name is constructed directly on top of free storage. PUTBYTE takes one character from register Al and stores it in the next position in the new string. PBOPEN initializes the process. Its argument is a full work in Al which is stored at the beginning of the string as its atom head. PBCIDSE terminates the process and stores the length of the string into the atom head. PBCIDSE returns a pointer to the new string atom. PUTBYTE must provide for exhaustion of free storage. When this occurs, the temporary string is converted to a bona fide string atom and a pointer to it is put on the stack. The garbage collector is called. On return, the temporary string is copied to the top of free-storage and PUTBYTE'ing continues. PBOPEN saves the address of the atom head in PBHD. If a type 0 atom is being created and GETOBJ finds an old instance of an atom with the given print name, storage allocated for the new print name is recovered. The free storage pointer is simply reset from PBHD.

RDSE. This routine has no arguments. It scans the input string for an s-expression and returns a pointer to that expression. RDCHAR is assumed to contain a legal character for the start of an s-expression, otherwise characters are skipped (and an error message is printed) until a legal character is found. RDSE checks to see if the string is an atom, a list, or a super list. In the first case it calls RDAT to read the atom. In the other two cases, it calls RDLIST to read the list. RDSE has the function of destroying structures if a right super paren is not found. It also prints the error message indicating how many parentheses were created. No parentheses are actually created;

the number is simply a count incremented as RDLIST exits each level of recursion for a missing right parenthesis. Normally, this count will be 1. That is, RDLIST did not find one right parenthesis before a right super-paren.

RDLIST. This routine has no arguments. It scans the input string and takes one list off the front. On entry, RDCHAR must contain either '(' or '<'. RDLIST calls RDSE to read each element of the list. RSLIST terminates when it finds either) or >. The former it changes to blank so no other routine reads it. The latter it leaves in RDCHAR so the next higher level can process it. In the latter case, a count is incremented' indicating that one parenthesis was created. While creating the structure for a list, RDLIST maintains two pointers, one to the beginning of the list, the other to the end of the list. After each element is parsed, a dotted pair is created of that element and NIL. Then a RST pointer to that new pair is stored in place of the NIL at the current end of the list. In this limited context, the operation RPLR (not a macro) works because a RST pointer always exists to be replaced.

RDAT. This routine scans the input string and takes the characters for one atom off the front of the string. It returns a pointer to that atom. The atom may be either a (symbol) or one of the (string) types as indicated in the syntax. A numeric character or dash in RDCHAR at the start of RDAT causes a branch to RANSCN. This routine scans a number and creates a number atom. Currently, the number must fit in eight digits because that is the size of the internal buffer used. An alphabetic character may be the start of either a symbol or some quoted string. The latter is distinguished by the quote following the alphabetic character. Quoted strings are scanned by RABITS which in turn passes control to RABX, RABW, or RABZ for hexadecimal, bit, and character strings respectively. After a string atom is created for the print name of a symbol atom, GETOBJ is called. GETOBJ either finds the old atom with the same print name, or makes a new symbol atom using the new character string atom as the print name. In the former case, storage for the new string atom is recovered.

C.3. CSSWYM Fields Used by READ Routines

RDCOL, RDEND, RDLNG. These fields control the scanning of the card by GETCH. RDCOL contains the address of the last character read, the character now in RDCHAR. RDEND points at the last character to be read from the card. RDLNG contains the number of characters to be read from a card. Normally, RDLNG is 71 because the continuation character is in column 72.

RDCHAR, RDCLASS. These one byte fields contain respectively the most recent input character and its class. The class of a character is illustrated in figure C.2.

RDSTAT. This byte contains bits representing the state of the read routines. These bits are detailed in figure C.3.

RDERMS, RDERNO, RDERLOC, RDERCT. These fields form the line printed for READ errors generated by RDERR and RDERRCNT. RDERMS is the address of the string passed to PUTSTR. RDERNO is the error number (the argument to RDERR). RDERLOC is the field beneath the card image and is set up with a single pointer ('<') to the last character scanned (character in RDCHAR). RDERCNT is used by RDERRCNT to store the number of parentheses created for error R2.

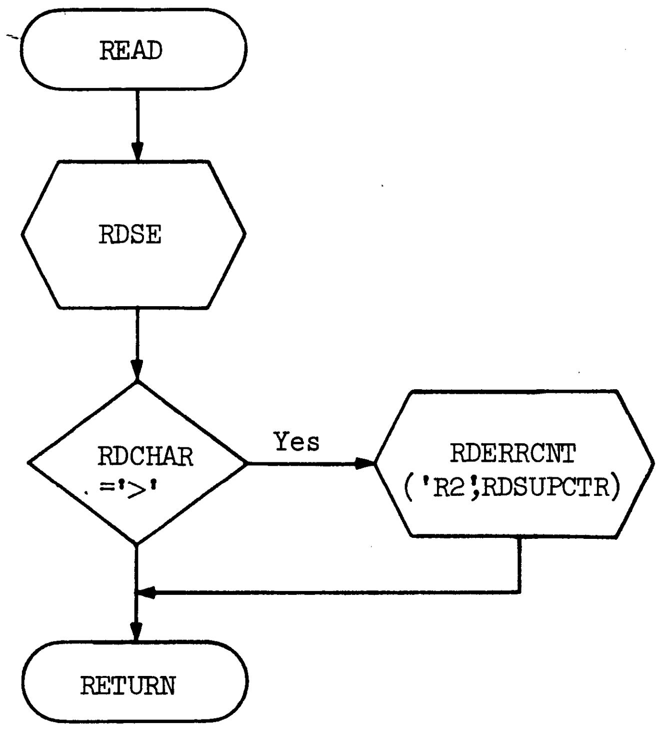
RDSUPCTR. This field accumulates the number of parentheses created before a right super-parenthesis. It is incremented each time RDSE exits due to finding a '>' instead of a ')' at the end of a list. When recursion returns to a level of RDSE looking for '>', RDSUPCTR contains one more than the number of parenthesescreated. RDSUPCTR is zeroed both before and after reading a list bounded by super-% parenthesis.

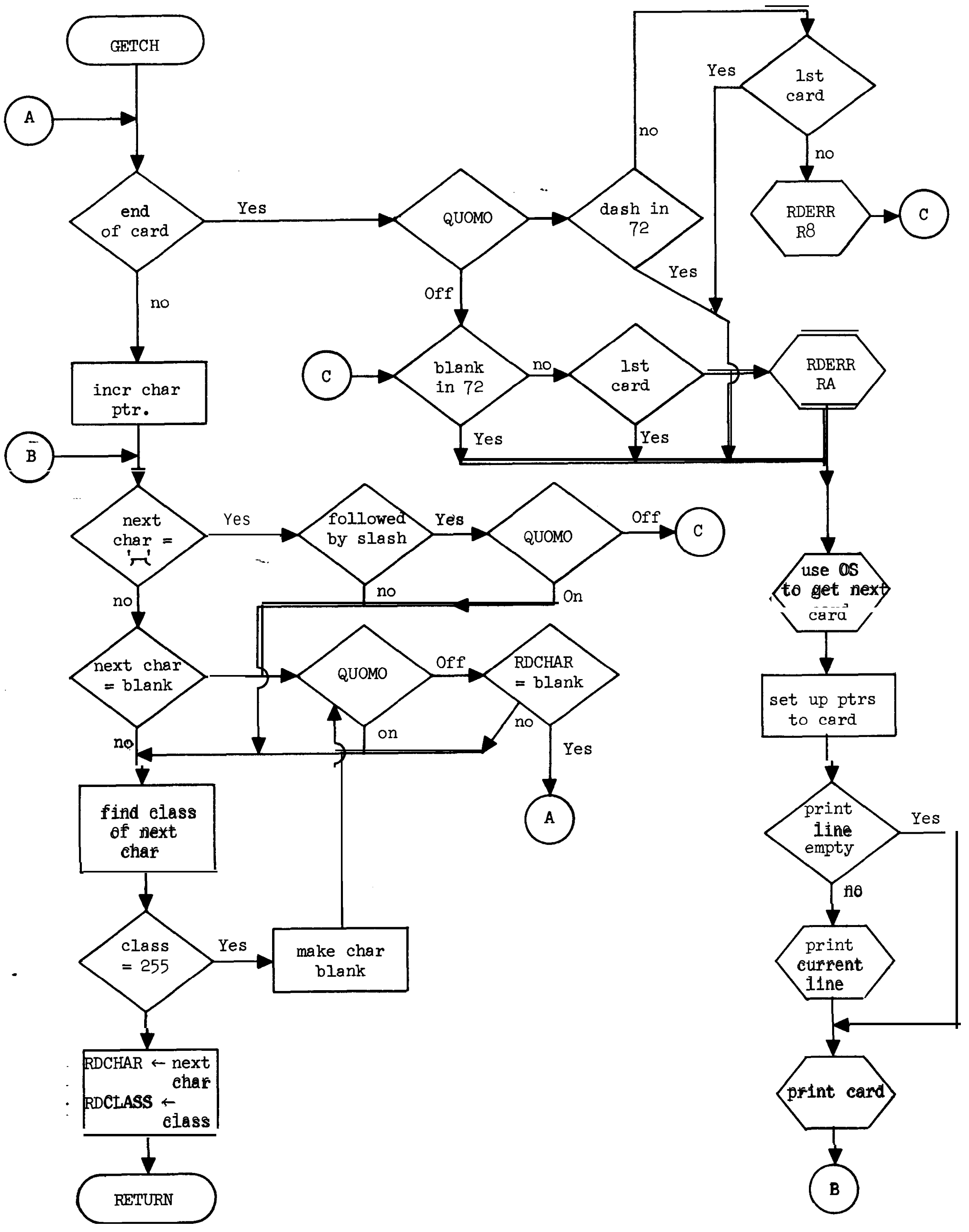
ATAMT. This half-word contains the atom offset. Atom pointers point [the quantity in ATAMT} bytes in front of the atom they reference.

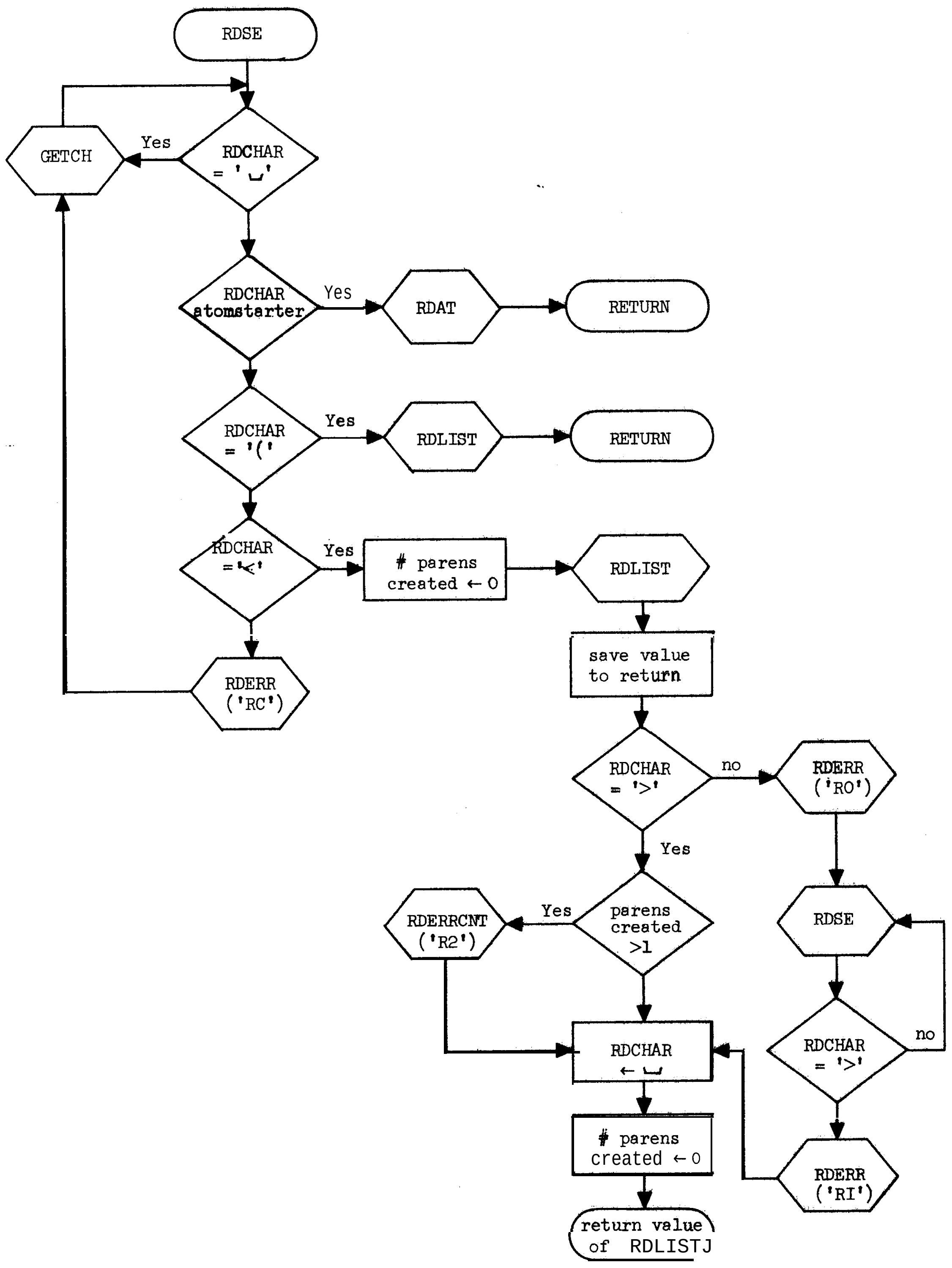
PBHD. While PUTBYTE is being used to create a character string atom on top of.free storage, register F points at the location to store the next byte. PBHD contains the contents of F before PBOPEN was called. PBHD - ATAMT will be the address of the created character string atom.

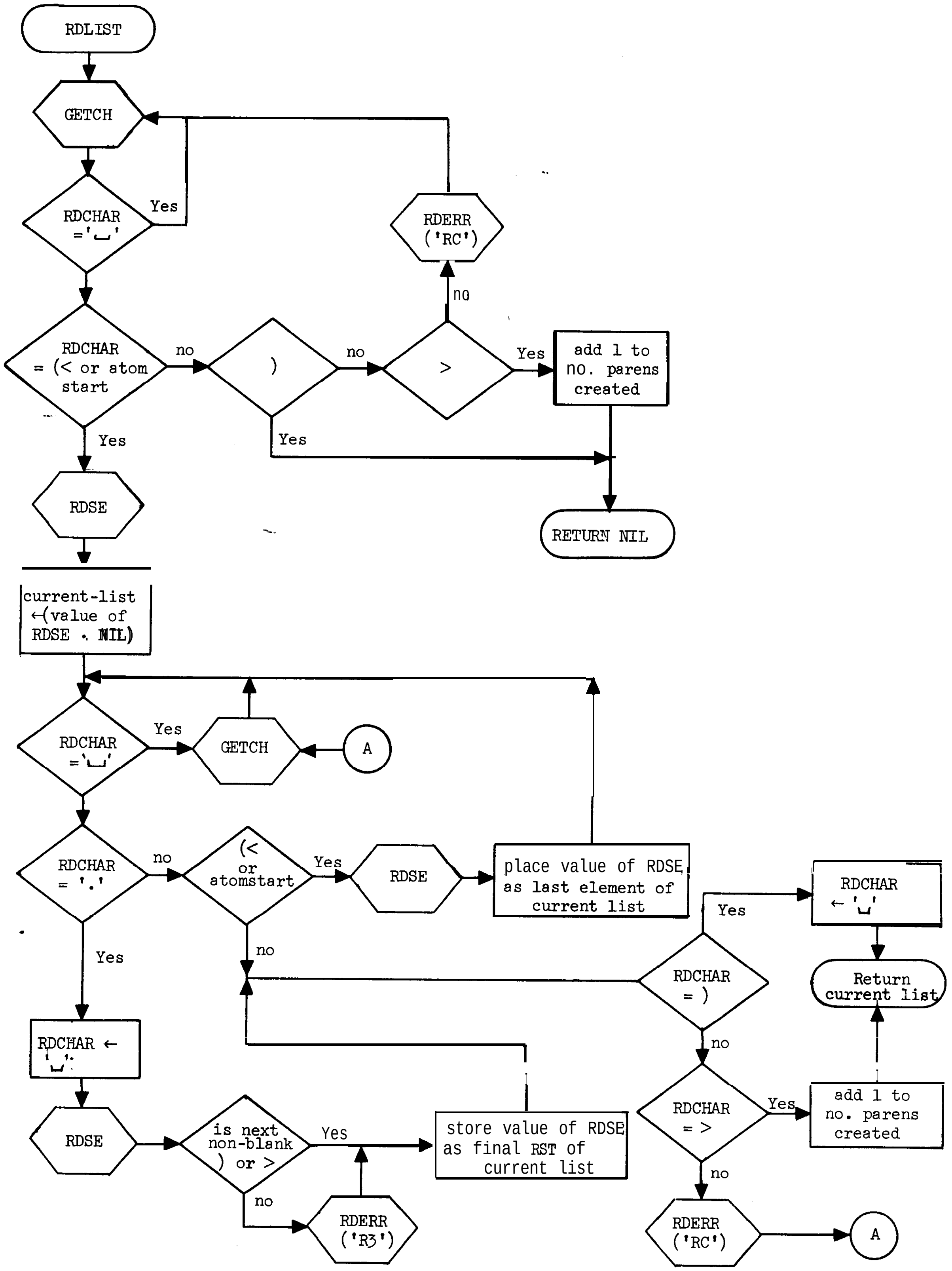
C.4. Flow Charts

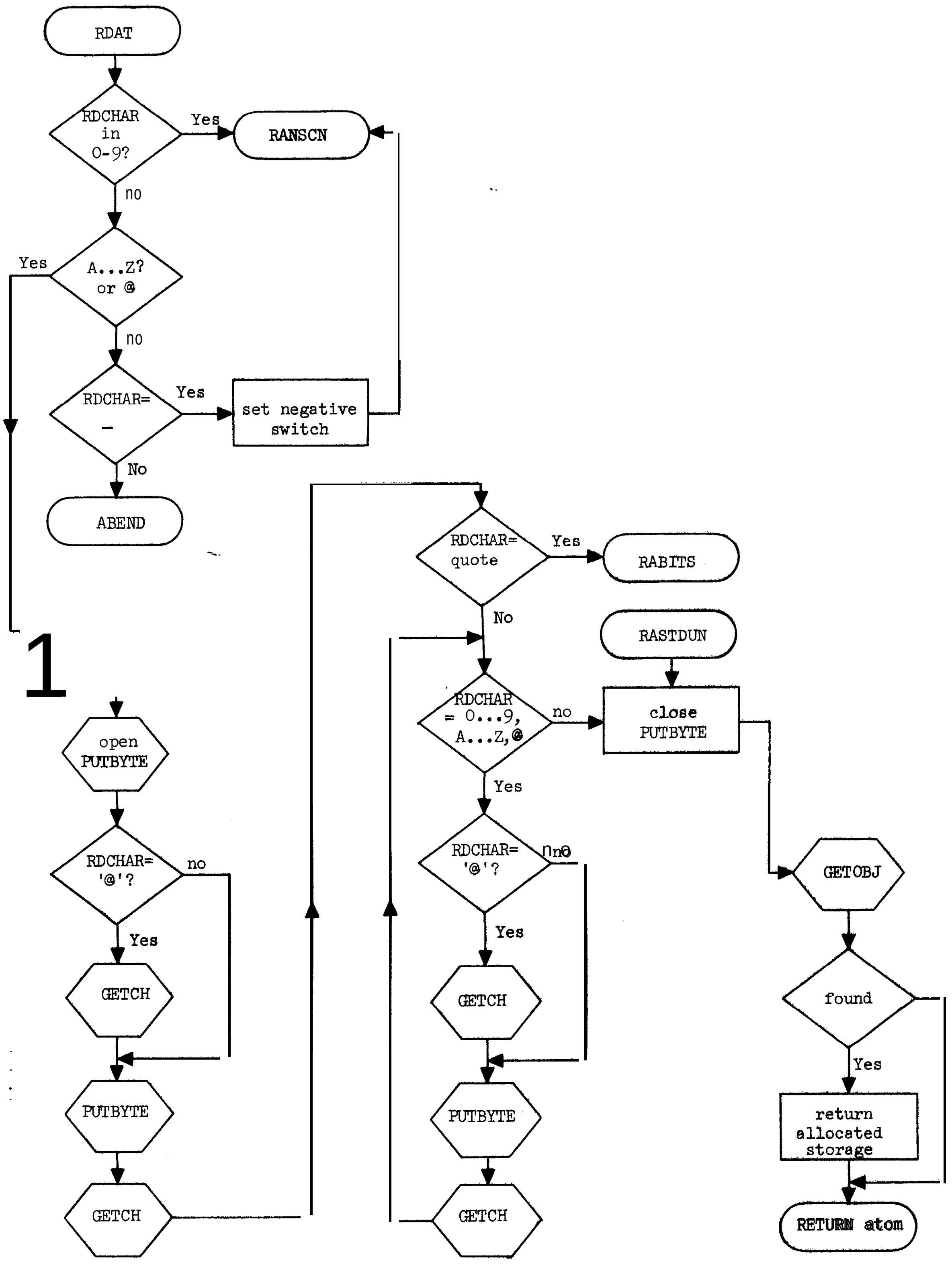
Flow charts are included in this section as the most concise means of describing the parsing algorithm in complete detail. The parser is similar to the parsers compiled by Cogent. The syntax is designed so there is never any ambiguity in the string. That is, from the current location in the program and the next incoming character, it is always possible to decide the type of the forthcoming input construct. Then the appropriate routine is called to handle the indicated type.

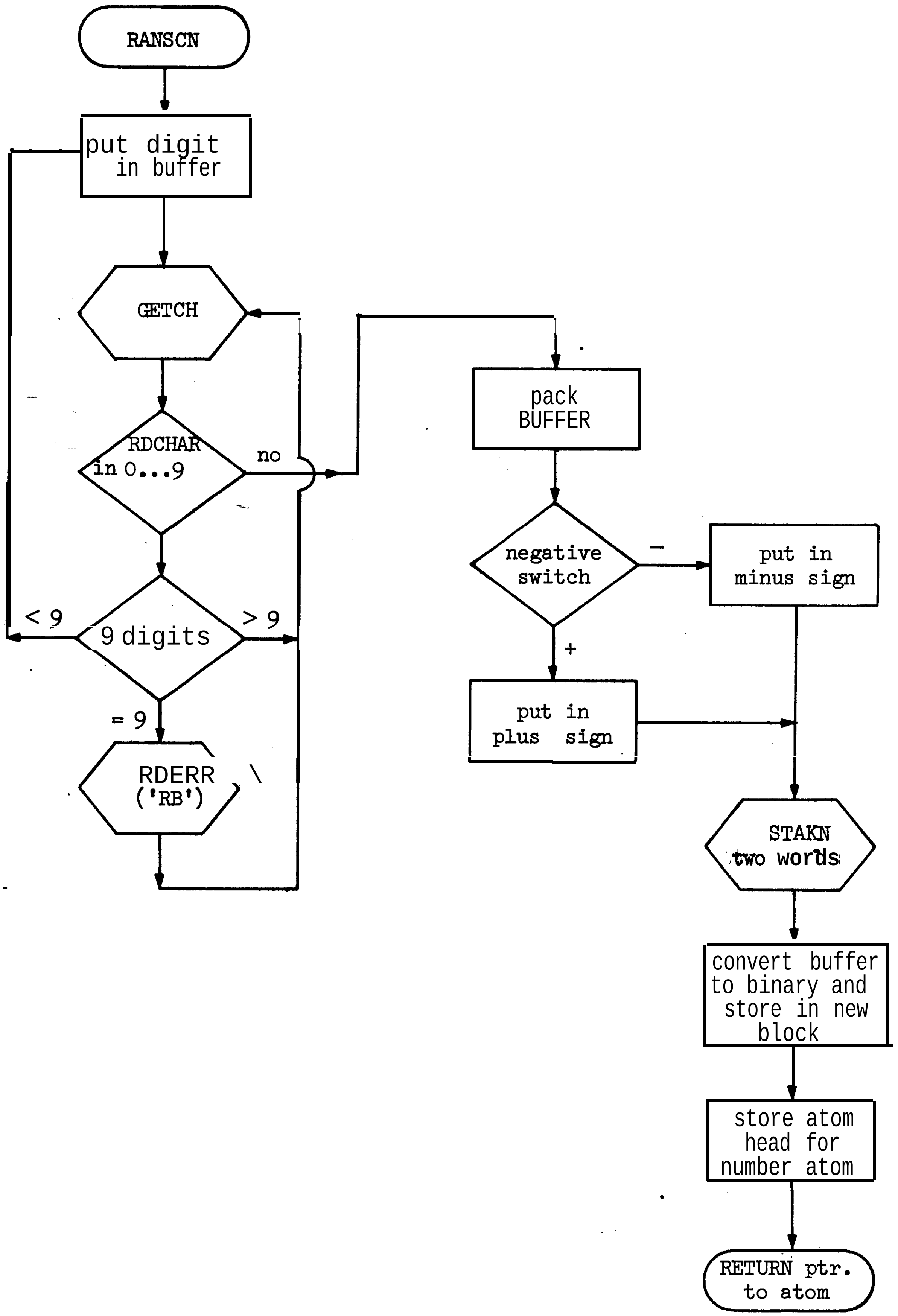


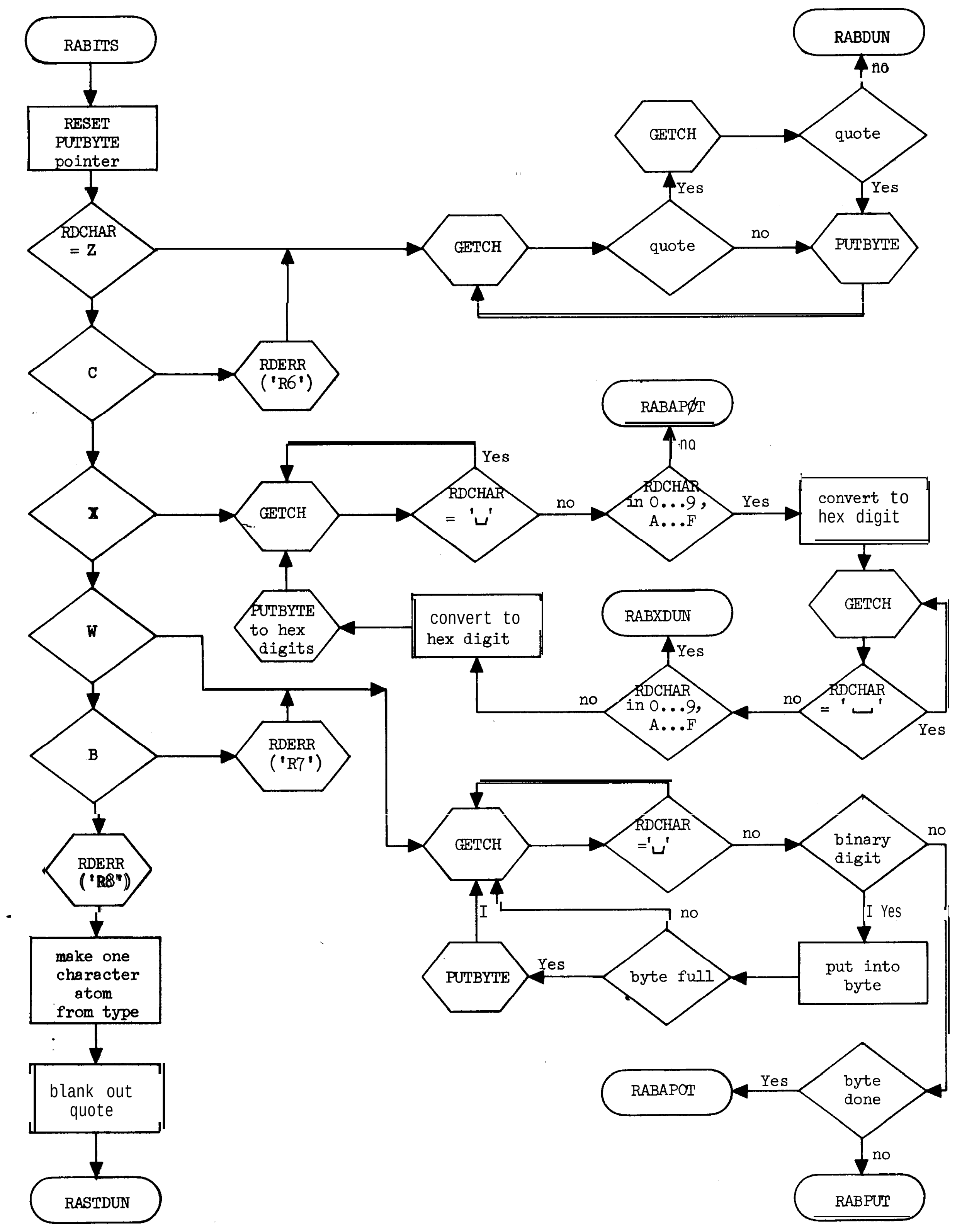

GETCH
A
end of card
Yes
no
QUOMO
Off
dash in 72
no
Yes
1st card
Yes
no
RDERR R8
C
C
blank in 72
no
1st card
Yes
Yes
RDERR RA
incr char ptr.
B
next char = '⌴'
Yes
followed by slash
Yes
QUOMO
Off
C
no
On
no
use OS to get next card
next char = blank
QUOMO
Off
RDCHAR = blank
on
no
Yes
no
A
set up ptrs to card
find class of next char
print line empty
Yes
no
class = 255
Yes
make char blank
print current line
RDCHAR ← next char
RDCLASS ← class
print card
RETURN
B

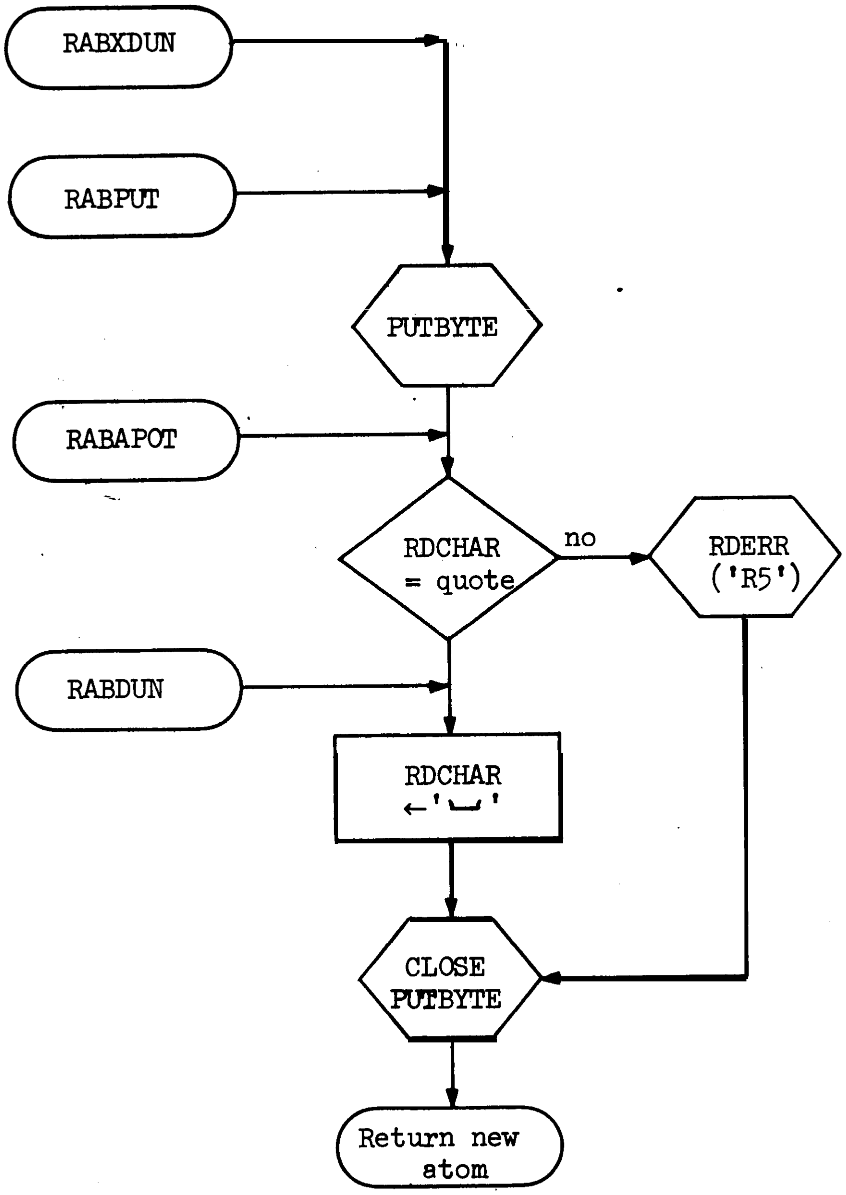

RDSE
GETCH
Yes
RDCHAR = '␣'
RDCHAR atomstarter
Yes
RDAT
RETURN
RDCHAR = '('
Yes
RDLIST
RETURN
RDCHAR = '<'
Yes
# parens created ← 0
RDLIST
RDERR ('RC')
save value to return
RDCHAR = '>'
no
RDERR ('RO')
Yes
RDERRCNT ('R2')
Yes
parens created >1
RDSE
RDCHAR ← ␣
RDCHAR = '>'
no
# parens created ← 0
RDERR ('RI')
return value of RDLISTJ


RDLIST
GETCH
RDCHAR ='␣'
Yes
RDERR ('RC')
no
RDCHAR = (< or atom start
no
)
no
>
Yes
add 1 to no. parens created
Yes
Yes
RDSE
RETURN NIL
current-list ←(value of RDSE . NIL)
RDCHAR ='␣'
Yes
GETCH
A
RDCHAR = '.'
no
(< or atomstart
Yes
RDSE
place value of RDSE as last element of current list
RDCHAR ← '␣'
Yes
no
RDCHAR =)
Return current list
Yes
RDCHAR ← '␣'
no
RDSE
is next non-blank) or >
Yes
store value of RDSE as final RST of current list
RDCHAR = >
Yes
add 1 to no. parens created
no
no
RDERR ('R3')
RDERR ('RC')
A


RDAT
RDCHAR in 0-9?
Yes
RANSCN
no
A...Z? or @
Yes
no
RDCHAR= –
Yes
set negative switch
No
ABEND
1
open PUTBYTE
RDCHAR= '@'?
no
Yes
GETCH
PUTBYTE
GETCH
RDCHAR= quote
Yes
RABITS
No
RASTDUN
RDCHAR = 0...9, A...Z,@
no
close PUTBYTE
Yes
RDCHAR= '@'?
no
Yes
GETCH
PUTBYTE
GETCH
GETOBJ
found
Yes
return allocated storage
RETURN atom


RANSCN
put digit in buffer
GETCH
RDCHAR in 0...9
no
< 9
9 digits
> 9
= 9
RDERR ('RB')
pack BUFFER
negative switch
−
put in minus sign
+
put in plus sign
STAKN two words
convert buffer to binary and store in new block
store atom head for number atom
RETURN ptr. to atom


RABITS
RESET PUTBYTE pointer
RDCHAR = Z
C
X
W
B
RDERR ('R6')
RDERR ('R7')
RDERR ('R8')
make one character atom from type
blank out quote
RASTDUN
GETCH
quote
no
PUTBYTE
Yes
RABDUN
GETCH
RDCHAR = '␣'
RDCHAR in 0...9, A...F
RABAPØT
convert to hex digit
PUTBYTE to hex digits
RABXDUN
RDCHAR = '␣'
binary digit
put into byte
byte full
byte done
RABAPOT
RABPUT


RABXDUN
RABPUT
PUTBYTE
RABAPOT
RDCHAR = quote
no
RDERR ('R5')
RABDUN
RDCHAR ←'␣'
CLOSE PUTBYTE
Return new atom

Appendix D. EVAL and the Stutter Interpreter

To facilitate experimentation with Swym, an interpreter for the evaluation of functions was provided. These functions are written in a language called Stutter, similar to LISP 1.5, but without PROG.

The interpreter is essentially the routine MAIN. When Swym is loaded for a Stutter run, MAIN is given control. MAIN can be described by:

```
main ( ) = begin
A: print (eval (read( )));
   terpri ( );
   goto A
end
```

(But note that Stutter does not currently have goto or assignment statements.) Thus, the interpreter repeatedly reads an expression, evaluates it, and prints the value. MAIN as implemented in assembly language also prints numbers between reading the expression and printing the value. The first is the time to read the expression, the second is the time to evaluate that expression. Both times are hundredths of a second. READ is described in Appendix C. PRINT and TERPRI are described in Appendix F. EVAL is described below. The routine ERROR exits to the loop in MAIN, so that interpretation can continue with the next expression. Succeeding sections of this appendix describe Stutter function definition, Stutter variable binding, and the individual internal interpreter routines.

D.1 Defining Functions to the Interpreter

There are four varieties of functions in Stutter, just as in LISP 1.5: SUBR, FSUBR, EXPR, FEXPR. SUBR's are machine language routines, executed by the machine. EXPR's are s-expressions executed interpretively by EVAL. The arguments for SUBR's and EXPR's are EVALuated before the function is called. FSUBR's and FEXPR's are the same as SUBR's and EXPR's, except their arguments are not EVALuated. Instead, a list of the unevaluated arguments is passed as the single argument to an FSUBR or an FEXPR.

Functions are stored on the property lists of symbol atoms. The indicatorused is the type of function. The value is either a pointer to a piece of code (SUBR's and FSUBR's) or a pointer at an s-expression (EXPR's and FEXPR's). These values can be stored, referenced, or modified using PUTPROP, GET, and REMPROP. To save property list searching time and storage space, a function definition for a symbol atom is stored in that atom's value cell. See the discussion of BINDERY in section D.3.

The format for an EXPR or FEXPR s-expression is different than that for Lisp 1.5. The expression should be a list of the form,

$$(vl\ \underline{exp_1}\ \underline{exp_2}\ \underline{exp_3}\ \ \ldots.\underline{exp_n}\ \underline{.at})$$

where:

vl is a list of variables. These are bound to the arguments of the function as discussed in the next section.

$\underline{exp_i}$ is an expression

each exp_i is evaluated until the atom $\underline{at}$ at the end is reached. Normally $\underline{n}$ is 1 and $\underline{at}$ is NIL so that a function definition looks like ($\underline{vl}$ $\underline{exp}$)rresponding to the LISP 1.5: (LAMBDA $\underline{vl}$ $\underline{exp}$)]

$\underline{at}$ this is the atom at the end of the list of expressions. If $\underline{at}$ is NIL, the value of $\underline{exp_n}$ is returned. Otherwise, the EVAL value of $\underline{at}$ is returned.

Two problems with a common solution exist in Stutter and in many implementations of LISP. First, a pointer at a piece of code -- the value of a SUBR property -- is not distinguished from a pointer at an s-expression. This leads to either errors or special handling in routines that accept arbitrary list structure as input, eg. PRINT. The second problem is the impossibility of compiling a function stored under a special indicator. Suppose the atoms of some class have, as one property, the indicator PROCESS whose value is a functions If the value is an s-expression, this code applies the appropriate-function to one such atom,

```
--. ((GET X (QUOTE PROCESS)) X)
```

This works because EVAL assumes that the FST will EVALuate to a function. But the only way code can be executed is to be stored under the indicator SUBR or FSUBR. The solution to both these problems is to create a third atom type: the code atom. Such an atom would indicate the location of the code and its length. It might contain garbage collection information such as relocatability and a list of pointers referenced by the routine. The atom might also contain information about whether the arguments should be evaluated.

D.2 Stutter Variable Binding

Two kinds of variable binding are used in Stutter. SUBR's and FSUBR's receive their arguments in registers A1, A2, . . . A6. Thus no SUBR may have more than six arguments. (FSUBR's always have exactly one argument.) Assembled routines may generally use the registers and the stack as temporary storage, as long as they obey the restrictions of Appendices I and A.2. The value of a SUBR or FSUBR is returned in A1.

EXPR's and FEXPR's are lists whose first element must be a list of symbol atoms (called vl, variable list, above). There must be exactly as many atoms in the list as arguments in the function call. The arguments of the function are stored in the value cells of the listed symbol atoms. The previous contents of the value cells are stored in a stack-block type 1 as described in Appendix A.2. When EVAL is called with a single symbol atom as its argument, the value returned is the value in that symbol's value cell. Thus, sub-expressions are EVALuated using the appropriate values for symbol atoms.

Using the value cell mechanism there is no simple method of establishing any particular environment that existed at some higher level (for example, that existed'whenafunction was passed as an argument). That would be dynamic variable binding. Stutter variable bindings are static; that is, every variable has its most recent binding time-wise, regardless of when a function was passed as an argument. This affects free variables of passed functions and their sub-functions.

D.3 Stutter Interpreter Internal Routines

Six routines are basic to the Stutter interpreter: MAIN, EVAL, EVLIS, EVGET, BINDERY, UNBIND. They are all assembly language routines. With the exception of EVAL, they are not available to the Stutter programmer.

MAIN.

This routine is the central loop of the interpreter. It was described above.

EVAL.

This routine has one argument, an s-expression. The expression is evaluated in terms of the current environment (bindings of variables). A complete description of the action of EVAL is in figure D.1. EVAL, like all Stutter functions, returns its value in register A1. In D.1, symbolp(a) is a predicate true when a is a symbol atom. The other functions are described further on in this appendix. WNBND points at a special atom. It is the contents of the value cell of any unbound atom (if there is no function definition in the value cell.) EVAL signals an error when an unbound atom is EVALuated. EVAL should also test for the value cell containing a function definition and signal the same error. Currently, though, this latter test is not made. EVAL handles correctly the evaluation of an atom whose value is non-relocatable, i.e., a number. The value is converted into a numeric type 1 atom. This makes possible communication between the interpreter and fast arithmetic functions using the value cell simply to hold a number.

When the fst of EVAL's argument is non-atomic and evaluates to a non-atomic expression, that expression is treated as though it were an FEXPR. That is, its arguments are not evaluated. However, the variable list for that expression must have as many atoms as EVAL's argument has rst's because of the way the call on BINDERY is reached. This permits the expression to have some control over the evaluation of its arguments. The most serious problem is the inconsistency of this feature with the rest of the language.

EVLIS.

This routine has one argument, a list of s-expressions. Its value is a list of the EVAL values of those s-expressions. EVLIS simply applies EVAL to each member of its argument list and creates a list of the values. The length of the list is computed and a compact list of that length is allocated. Successive values are stored in that list.

It is now realized that using free storage to return the value of EVLIS is just as flagrantly wasteful of space as an a-list would have been. The appropriate correction is to have EVLIS place values on the stack. They would then be taken off the stack by BINDERY. Since BINDERY must put information on the stack, the best solution is the combination of EVLIS and BINDERY into a single function. This function would create a BINDERY type stack block and store the new values of the atoms in it. When all arguments were EVALuated, the values would be swapped between the stack and the value cells of the atoms. Note that the call of EVLIS at the label EVSUBR in EVAL must be replaced with code, probably in-line, that stores new values in the stack and then places them in the registers.

EVGET.

This function gets the function definition of a symbol atom from that atom's value cell or property list. This is a non-standard function in that its-argument is passed on the stack. The value is returned in Al. EVGET also stores the previous contents of Al on the stack to avoid repeating that store in several places in EVAL. EVGET first checks the CELVAL bit in the atom head. If that bit is off, the contents of the value cell are the function definition for the atom. If CELVAL is on, EVGET finds out (by indexing VFPROPS with the CELFNC bits) the type of function definition: SUBR, FSUBR, EXPR, or FEXPR.

GET is called to find the function definition on the property list.

BINDERY.

This function has two arguments; a list of values, and a list of symbol atoms. The result is to store each value in the value cell of the corresponding atom. When EVAL subsequently evaluates one of these atoms, it retrieves the new value. The old values of the atoms are stored in a plex on the stack (stack plex type 1 -- ses Appendix A.2). This stack plex must later be popped off the stack by a call on UNBIND.

Information is left on the stack after BINDERY exits. This leads to the stringent requirement that BINDERY may not itself use temporary storage on the stack, nor may the calling' routine. BINDERY does all its computation in the general registers. When EVAL calls BINDERY, a pointer to EVAL's argument is in register A3. BINDERY must not affect this register.

Because BINDERY cannot call functions, it cannot bind a symbol atom having a function definition in the value cell. The function definition would have to be put on the property list, which would require storage allocation and possibly garbage collection. Consequently, BINDERY causes error BI when a yalue cell contains a function definition. The simplest solution to this problem is to not store function definitions in the value cell. This would increase property list searching time, but would save a great deal of messy bit pushing. A second solution would be to always store function definitions on the property list and to store them in the value cell until the atom is bound to some value.

UNBIND.

This function pops off the stack a plex stored on the stack by BINDERY.

Note that UNBIND must be called when the BINDERY plex is at the top of the stack, or disaster will occur. UNBIND may not use any storage on the stack, nor may it affect register Al.

Figure D.1

```
eval (a) = begin list x, y;
     if atom (a) then
          if symbolp (a) then
               if cell (a) = VUNBND then error (El)
               else return (cell (a))
          else return (a)
     else if ¬ atom (fst (a)) then begin
          x: = eval (fst (a));
          if ¬ atom (x) then begin
               comment assume x is s-expression for an FEXPR w/ multiple arguments;
               y: = rst (a); goto EVENBD;
          end
     end else x: = fst (a);
x := get (x, { SUBR, FSUBR, EXPR, or FEXPR depending on bits in atom head]);
goto {EVSUBR, EVFSUBR, EVEXPR, or EVFEXPR depending on bits in atom head];

EVSUBR:   y : = evlis (rst (a));
          {place elements of y into registers Al to A6};
          return ({execute routine pointed at by x});
EVFSUBR:  {put rst (a) into register Al};
          return ([execute routine pointed at by x});
EVEXPR: y := evlis (rst (a));
```

```
EVENND:  bindery (y, fst (x)); x := rst (x);
EVELP:   if atom (x) then begin
              x := eval(x); unbind ( ); return (x)
         end;
         Y := fst (x) ; x := rst (x);
         if null (x) then begin
              x := eval (y); unbind( ); return (x)
         end;
         eval(y);
         got0 EVELP;
EVFEXPR: bindery(list (rst (a)), fst (x));
         x := rst (x);
         got0 EVELP
end eval
```

Appendix E. Swym Garbage Collector

One of the important goals of Swym was the development of a list compacting garbage collector. This appendix explains that collector in great detail. Section III.2 contains a simple version of the collector explaining the basic concept. The first section of this Appendix describes the heart of the collector in a higher level language. The second section describes the internal garbage collector routines (i.e., those not available to the STUTTER program). The last section describes those portions. of CSSWYM used by the garbage collector.

E.1. The Complete Garbage Collector Algorithm

The simple garbage collector in III.2 is inadequate for many common list structures: circular lists, several lists with the same rst, a structure which is an element of more than one list, and-more pathological cases., The implemented garbage collector handles all possible cases with marking bits and a fixup table.

Two marking bits are associated with each list word, Each pass sets a marking bit to indicate it has visited a given word. The first pass sets bit m1, the second sets m2. Special action must be taken when a marked word is encountered, because that word is already being processed at some other level of recursion. A word with m2 set always contains the address of the corresponding word in the new core image.

Several functions set and test the marking bits:

MARK1 (w)	The word pointed at by w is marked with m1.
MARK12 (w)	The word pointed at by w is marked with both m1 and m2.
UNMARK1 (w)	m1 is turned off in the word pointed at by w.
M1 (w)	This predicate is true if m1 is on in the word pointed at by w.
M2 (w)	This predicate is true if m2 is on in the word pointed at by w.

Conceptually, each of these functions tests its argument to see if it points at an atom and adjusts the addressing appropriately. In practice it is known a priori whether the argument is an atom, and a bit macro (see B.5) is coded instead of a function call.

In circular structures, a word points at some structure already being collected at some higher level of recursion (m1 is set, but not m2). That word cannot be written correctly to the new core image because its contents are not determined. In most reasonable applications, the number of such circularities is well below one percent of the number of pointers. Nonetheless, some provision must be made to handle this case; in Swym, the garbage collector uses a fixup table. When the correct new contents of a word cannot be determined, a word of zeros is written to the new core and an entry is made in the fixup table. Each entry is two pointers. The first points at the word of zeros in the new core; the second points at the word in old core which will eventually contain the correct address to substitute for the word of zeros. After COLLECT is finished, the second pointer of each fixup entry is replaced by the contents of the word it points at. Then, after the new core image has been read in, the fixups are applied; i.e., the second word of the entry is 'or'ed into the location indicated by the first word of the entry. (The 'or'ing permits the word of zeros to have the rst bit on if required. The fixup procedure thus works for both fst and rst fixups.)

One additional function must be defined to describe the complete garbage collector (others are defined in 111.2):

FIXUP (p, c) The word c (either zero or rstbit) is GCPUT to the new core. An entry is made in the fixup table consisting of the address returned by GCPUT and the pointer p.

The function ATCOL defined in section III.2 must be extended. When ATCOL is entered, the ml is set in the plexhead. After collecting the atom, both marking bits are set. Since COLLECT may be called for some sub-structure of an atom, provision is made for a pointer at an atom with ml and not m2 (a fixup entry is generated).

The complete garbage collector is given in Figure E.1. The argument x must be a pointer at list structure with neither marking bit on. COLLECT has no value, but the new-core address of the list corresponding to x is stored in place of the pointer to fst(x). A demonstration that this algorithm creates a correct representation of its argument is given in Appendix L. The UNMARK1(r) and the boolean variable m are related. The former indicates the need for a fixup in the rst direction; the latter detects this need in the second pass. In Figure E.1, the marking bits are assumed to be associated with each word, but not part of the word. This association could be by extra bits in the hardware or by a bit table in a separate area of memory. The former requires hardware modification, while the latter requires six percent more memory. In the implemented system, the marking bits are in the list words themselves, as shown in Figure 2. Figure E.1 must be modified for these bit assignments by turning off the marking bits in the arguments to GCPUT and replacing

```
                    t := rst(r)
```

with

```
                    if M1(r+4) then t := r+4 else t := rst(r).
```

Figure E.2 illustrates effect of COLLECT on a complex structure.

Figure E.1

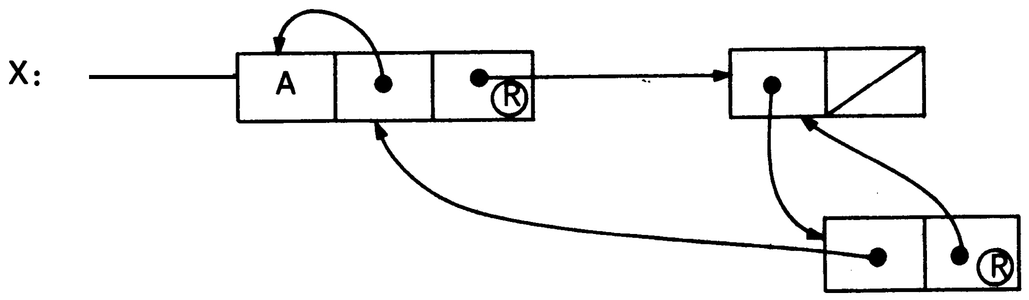
Swym Garbage Collection Algorithm

```
COLLECT (x) = begin list r, t; Boolean m;
   rstbit        := x'00000001';

   r := x;
 chkloop:  comment loop to collect each fst;
   t := fst (r);  MARK1 (r);
   if atom (t) then ATCOL (t) else if M1 (t) then COLLECT (t);
-comment test for end of list or reached marked word;
   t := rst (r);
   if atom (t) then ATCOL (t)
   else if M2 (t) then
   else if M1 (t) then UNMARK1 (r)
   else begin r := t;  goto chkloop end;

   r := x;
 wrloop:  comment loop to write out each new fst;
   m := M1 (r); t := fst (r);
   rplf (r, if atom (t) then
                 if M2 (t) then GCPUT (HD (t)) else FIXUP (t, 0)
             else if M2 (t) then GCPUT (fst (t)) else FIXUP (t, 0));
  I MARK12 (r);
 comment test for end of second pass;
   t := rst (r);
   if atom (t) then
      if M2 (t) then GCPUT (HD (t) v rstbit)
      else FIXUP (t, rstbit)
   else if M2 (t) then GCPUT (fst (t) v rstbit)
   else if m then begin r := t;  goto wrloop end
```

Figure E.1 Continued

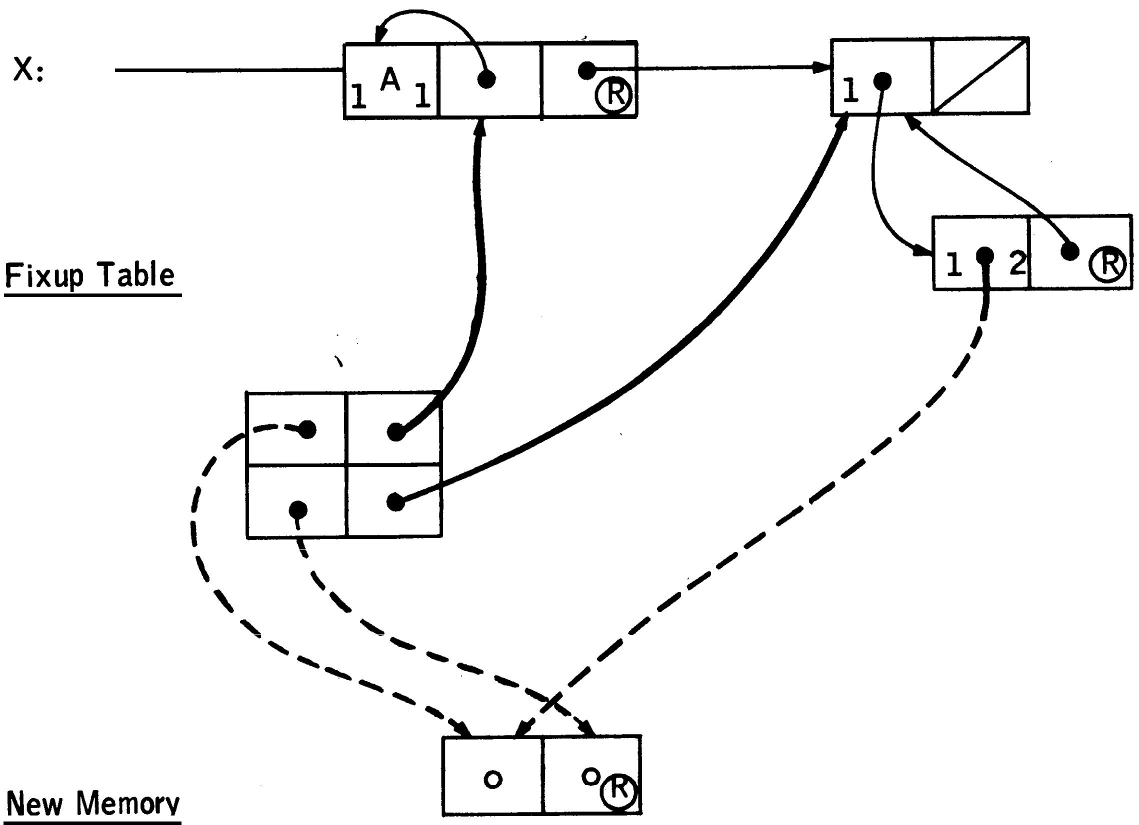
```
   else FIXUP (t, rstbit)
end collect
```

Figure E.2



At wrloop on the highest level:

Old Memory

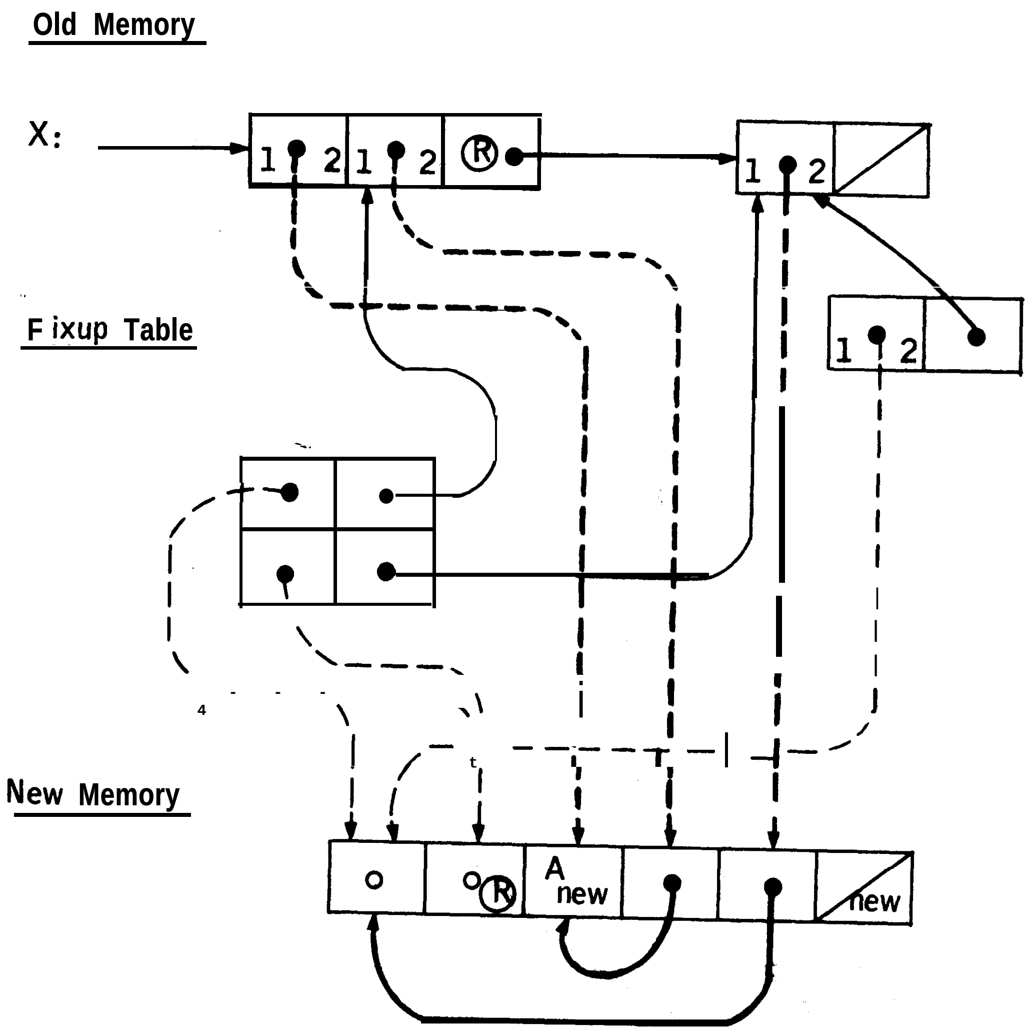


Fixup Table

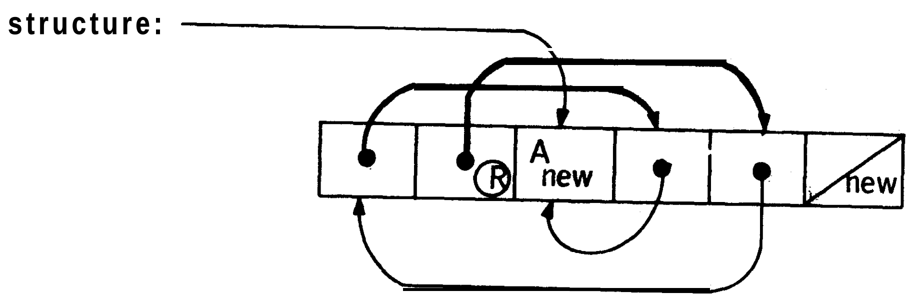
New Memory

Figure E.2 (Con't)

At exit from COLLECT:



Final structure after-reading new core image and applying fixups:



E.2 Garbage Collector Internal Routines

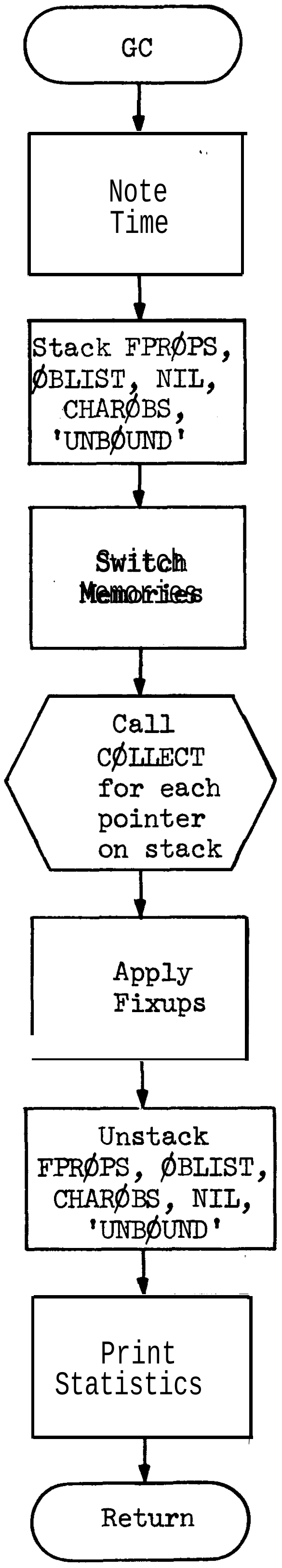
The interface between all other routines and the garbage collector is the routine CC. It receives control when TAK2 or some other routine detects insufficient memory, or it may be called explicitly from a Stutter program. GC controls the garbage collection process and prints statistics. CC, ATCOL, COLX, and COLLECT are called with the standard CAL macro. CHOKE, GCABEND, and GCPUT are routines with special calling sequences.

Routines written to garbage collect newly created atom types must be made part of the routine ATCOL. The description of that routine includes information on inserting new atom collection routines. But all the information in section E.3 should be understood before coding special atom collection routines.

Gc This is the executive portion of the garbage collector. Its major functions are outlined in Figure E.3. Pointers at OBLIST, CHAROBS, NIL, FPROPS, and *UNBOUND* are put on the stack so the corresponding information will be garbage collected. Since the OBLIST points at all symbol atoms, both they and their property lists will be collected.

The current implementation does not use temporary storage for garbage collection; instead, the data structures are moved between two areas of memeory. The 'switch memories' action in Figure E.3 is merely the swapping of pointers so GCPUT will store the new structures into the currently non-active free-storage area. In an implementation using temporary storage, the temporary data set would have to be initialized. Similarly, the step 'apply fixups' would have to be preceded by 'read in new core image'.

Figure E.3


GC
Note Time
Stack FPRØPS, ØBLIST, NIL, CHARØBS, 'UNBØUND'
Switch Memories
Call CØLLECT for each pointer on stack
Apply Fixups
Unstack FPRØPS, ØBLIST, CHARØBS, NIL, 'UNBØUND'
Print Statistics
Return

The following statistics are printed, all on a single line:

length of active pdl (stack)

number of bytes of active free storage

time at start of garbage collection (100 ths/sec)

time at end of garbage collection (100 ths/sec) (times are since last starting the READ in the MAIN loop)

total time for garbage collection (100 ths/sec)

CØLLECT. This routine has been described in detail in section E.l. The argument (in Al) to CØLLECT is a pointer at an unmarked list. CØLLECT has no result, but the fst of the argument points at the representation of that list in the new core.

ATCØL. This routine garbage collects one atom and writes a representation of that atom to the new core image. The argument (in Al) must be a pointer at an unmarked atom. The result is that the head of the atom is replaced by the new-core address of that atom. The main routine of ATCØL simply abstracts the type field from the atom head and branches to the appropriate routine for that atom type. Currently, there are routines for symbol atoms and bit string atoms. Adding a new routine is done by putting the address of the routine into the branch table (ATCBTBL). If more than eight atom types are implemented, the table can be extended by increasing the number of bits masked from the type field. The individual processing routines should branch to ATCXIT after completely collecting the atom. The individual routines are responsible for replacing the atom head with the new core address of the atom.

ATCO. This is the part of ATCØL for collecting symbol (type 0) atoms. For such atoms, the atom head and the atom cell must immediately precede the property list. To achieve this, the routine processes the property list with a loop similar to the first loop in collect. Thus all pointers in the property list are marked with ml and all elements of the list are collected. Then ATCO collects the contents of the atom cell (if they are relocatable). Finally, ATCO writes the atom head and the new atom cell to the new core; then it transfers to the WRLØØP portion of CØLLECT to finish writing out the property list.

CØLX. The-argument to CØLLECT must not be marked and must not be an atom. The argument to CØLX may be marked or unmarked, atomic or not, But if marked, the structure must have both bits on. If its argument is unmarked, CØLX calls CØLLECT or ATCØL as required. The result of CØLX is a pointer at the new core representation of CØLX's argument. CØLX's can be used by atom collection routines if it is certain that its argument will never satisfy

$(m1(A) \wedge \neg\, m2(A))$.

CHØKE. If, following a garbage collection, insufficient free storage is available, then this routine should be entered. It is in the CSSWYM control section and can be entered simply with

```
        B CHOKE
```

or

```
        BC nn,CHOKE
```

CHOKE simply ABEND's with the user completion code 20.

GCABEND. If the garbage collector detects an error in the data structure construction, it ABEND's immediately to avoid propagating errors. A call on GCABEND is

```
BAL L,GCABEND
```

This routine constructs a completion code based on the displacement of the BAL from the beginning of the current routine. The contents of register 1 are stored in register L, and the ABEND is issued. The current completion codes and their significance are listed in Appendix J.

GCPUT. This routine is called by the GCPUT macro (section B.8). It is called by that macro with either

```
BAL  L,GCPUT
```

or

```
BAL  L,GCPUTFUL.
```

This routine must be changed if SWYM is to use temporary storage during garbage collection.(Note: The comments about #M1M2 in the next section).

ATC1. This portion of ATCØL collects bit string atoms. Since such atoms contain no relocatable information, ATC1 simply writes a new atom head and copies the string into the new core. The subtypes of type 1 atoms are designed so that the garbage collector need not distinguish among them. The length field always indicates a length in bytes and the garbage collector always transfers the integral number of words necessary to transfer all the bytes.

E.3 Information stored in CSSWYM

MEMUSE, MEMNXT. These two words contain the addresses of the two memories used alternately as free storage. On entry to CC, the two fields are swapped and the new contents of MEMUSE are the initial destination for words stored by GCPUT.

MEMSIZ. This word contains the number to be added to MEMUSE to compute the new FEND.

FEND. This word contains the address of the next to last word to be stored into by TAK2. When this word or the succeeding word is stored, TAK2 calls GC. FEND is also used by PBØPEN, PUTBYTE, and STAKN to check for the end of the free storage area.

GCTIME. GC saves the TTIME time on entry and uses it to compute the total garbage collection time before exitting. This total is printed in the garbage collector statistics line.

GCABAD. This word is used by GCABEND to create a completion code for ABEND. Because the high order bit is on, ABEND calls for a dump.

#M1M2. This word is used by GCPUT to put the Ml and M2 bits on the address word it returns. #M1M2 must be in CSSWYM because B may have different values when GCPUT is called.

Appendix F. Stutter Functions

This appendix details all functions available to the Stutter programmer. They are represented in initial free storage by atoms with the property SUBR or FSUBR. For each routine there is a description of the inputs, the value of the function, and the internal code involved. Three routines are described in more detail in separate appendices: GC, EVAL, and READ.

Internally, a Stutter function cannot be distinguished from a Swym system function. Specifically, all Stutter functions can be called internally with the standard CAL macro. The name of the function is the same to the CAL macro as to the Stutter program. (Note that a few functions - like RST and FST - are also available as macros. Although they can be called with CAL, it is clearer and faster to use the macro form.) Arguments to these functions are passed in registers Al, A2,... A6. The value is returned in register Al. Any excess arguments are ignored; they may or may not remain after execution of the function.

The routines are organized in five groups: basic, input, output, Stutter and utility. This index tells where to find each routine:

Routine	Group	Type	# of Args.	Control Section
ATOM	basic	SUBR	1	CSSUBS
BELL	utility	SUBR	1	CS2250
COND	Stutter	FSUBR		CSEVAL
EJECT	output	SUBR	0	CSPRINT
EQ	basic	SUBR	2	CSSUBS

Routine	Group	Type	# of Args.	Control Section
ERROR	utility	SUBR	1	CSSUBS
EVAL	Stutter	SUBR	1	CSEVAL
EXPLODE	output	SUBR	1	CSEVAL
FST	basic	SUBR	1	CSSUBS
GC	utility	SUBR	0	CSGC
GET	Stutter	SUBR	2	CSEVAL
GETOBJ	input	SUBR	1	CSREAD
IVCCH	input	SUBR	0	CSREAD
IVQMØ	input	SUBR	0	CSREAD
LIST	basic	FSUBR		CSEVAL
MAKSTRNG	input	SUBR	1	CSREAD
NULL	basic	SUBR	1	CSSUBS
PRINT	output	SUBR	1	CSPRINT
PRIN1	output	SUBR	1	CSPRINT
PUTPROP	Stutter	SUBR	3	CSEVAL
QUOTE	Stutter	FSUBR		CSEVAL
READ	input	SUBR	0	CSREAD
READCH	input	SUBR	0	CSREAD
REMPROP	Stutter	SUBR	2	CSEVAL
RST	basic	SUBR	1	CSSUBS
SASSOC	Stutter	SUBR	2	CSEVAL
STIVCCH	input	SUBR	1	CSREAD
STIVQMØ	input	SUBR	1	CSREAD
TAK2	basic	SUBR	2	CSSUBS
TERPRI	output	SUBR	0	CSPRINT

F.1 Basic Routines

RST, FST, TAK2, ATØM, NULL, EQ, LIST

The routines in this group are the lowest level functions for the manipulation of lists.

(RST x). Returns the ReST of the list x, which must not be atomic. Atomic x results in a specification interrupt4

(FST x). Returns the FirST element of the list x, which must not be atomic. Atomic x results in a specification interrupt.

(TAK2 x, y). If y is a list, returns a list whose FST is x and whose RST is y. If y is atomic (other than NIL), TAK2 returns a generalized list, that is, a list whose R...RST is not NIL. In either case, TAK2 is well defined. This function takes two words from the free storage block and thus incurs part of the expense of the next garbage collection. Beware when CAL'ing TAK2 from an assembled routine. Because the garbage collector might be called, all registers must be saved, and all pointers must be identifiable as such.

(EQ x, y). Predicate. If x and y are atomic, returns T if they are the same atom, and NIL if they are not. If x or y is not atomic, returns T if x and y both point at the same location. EQ is always defined.

(ATØM x). Predicate. Returns T if x is an atom and NIL otherwise.

(NULL x). Predicate. Returns T if x is the atom NIL. If x is any other atom or is non-atomic, NULL returns NIL.

..... x_1, x_2 . . . x_n). Returns a list whose elements are $x_1, x_2, \ldots x_n$. Unlike other basic functions, LIST accepts any number of arguments. Note in particular that (LIST) is valid and returns NIL. LIST is implemented so that if given n (> 1) arguments it will use ntl words from the free storage block. Thus list is more efficient than successive TAK2's.

F.2 Input Routines

READ, READCH, IVCCH, STIVCCH, IVQMØ, STIVQMØ, MAKSTRNG, GETØBJ

The Stutter input routines are well developed since they were a necessary adjunct to testing the system. Two modes are provided: READ reads an entire expression. It is also used by the main interpretative loop, so an understanding of it is an understanding of the input syntax for Stutter. A single character input mode is also provided to permit the writing of more general input. The internal read routines are described in Appendix C.

The read routines make use of a device, borrowed from CØGENT, called an "internal variable". This is a variable whose value affects the system and which can be set or reset by special subroutine calls. Each internal variable is represented by a three character mnemonic; two routines are associated with each internal variable. If the mnemonic is xxx, the routines are (IVxxx) and (STIVxxx a). The first routine returns the current value of the variable and the second assigns the value of a to the variable. If the variable is a switch, it will have the value T or NIL and can be set by STIVxxx. The argument NIL sets the switch off and any other argument sets the switch on.

(READ). One expression is READ from a card or cards and returned as the value of READ. This routine is described in detail in Appendix C.

(READCH). READs the next CHaracter from the input card and returns a pointer to an atom with that character as its print name. All printable characters and ¢, !, $\begin{bmatrix}0\\2\\8\end{bmatrix}$ already exist as objects in the system. Any other character is translated by READCH into blank. EQ may be used to compare characters because they are uniquely represented. Characters are read using the same conventions of card layout, that is, columns 1 to either 71 or the first underbar-slash. Also, if the current character is a blank, READCH will return the next non-blank character. These conventions may be altered by turning on the quote mode with (IVQMØ).

(IVCCH) (STIVCCH x). To store one character in the case that an expression read by READ is an atom and the following character is a left parenthesis, an internal variable called 'Current CHaracter' is defined. Its value can be SeT to any character by STIVCCH. An error is signalled if the argument is not an atom with a one character printname. The 'current character' can be accessed by evaluating (NCCH).

The relationship between REAL, READCH, and IVCCH is most easily explained in terms of a 'scan pointer' and a character variable called the 'current character'. The scan pointer moves along the input text having due regard for card boundaries and the '_/' convention. The character pointed at by the scan pointer is called the scanned character. After READing an atom, the scan pointer points at the character following the atom (usually blank) and the current character contains the scanned character. After READing a list, the scan pointer points at the final right parenthesis and current character contains a blank. IVCCH does not affect the scan pointer and returns the current character. The first character read by READ is the current character. Succeeding characters would be the values of successive READCH'es. READCH can best be described as a call on GETCH, as flow charted in Appendix C.4. An approximation to READCH can be given by:

```
Loop:  move scan pointer to next character;
       if (current character is blank A
                 quote mode is off A
                 scanned character is blank) then go to loop;
       current character := scanned character;
       return (scanned character).
```

(IVQMØ) (STIVQMØ x). If Quote MØde is on, then each character on each card is passed in turn as the value of READCH. This provides a means of avoiding the normal underbar-slash and de-blanking conventions. Unfortunately, in this mode there must be a dash in column 72 (or quote mode must be set off just before column 71 is scanned). Calling READ always sets quote mode off.

(MAKSTRNG x). x must be a list whose elements are all symbol atoms with one character print names. The characters are collected together and the value of MAKSTRNG is a character STRiNG atom MAKed of the print names of those atoms. $\lceil \text{length}(x)/4 \rceil + 1$ words are taken from the free storage block.

(GETØBJ x). x must be a character string atom such as is returned by MAKSTRNG. The value returned by GETØBJ is an atom with the indicated print name. GETØBJ searches the OBLIST for an atom with the proper print name. If such an atom is found, it is returned; otherwise an atom is created. If an atom is created, three words are used from the free storage block.

F.3 Output Routines

PRINT, PRIN1, TERPRI, EJECT, EXPLØDE

The routines in this group provide for printing expressions and controlling the printer. A routine is also provided to abstract from a symbol atom a list of the characters in its printname. A print line is 132 characters; no access to the carriage control character is provided other than that supplied by TERPRI and EJECT.

(PRINT x). The expression x is PRINTed, and then the printer is spaced to a new line. Lines will be as full as possible without printing an atom name on two lines. This means that isolated left parentheses will appear on the right. The value of (PRINT x) is x. Internally, PRINT simply calls PRIN1 and TERPRI.

(PRIN1 x). Identical to PRINT except PRIN1 returns NIL and does not space the line printer after printing. The first character of a succeeding PRINT or PRIN1 will immediately follow the last character of a given PRIN1.

(TERPRI). TERminate the PRInt line. The line printer is advanced the the next line. (TERPRI x) returns x.

(EJECT). The line printer is EJECTed to the next page. The next PRINT or PRIN1 will put characters beginning at the upper lefthand corner of the next page.

(EXPLØDE x). x must be a type 0 atom (symbol). EXPLØDE returns a list whose elements are the character atoms corresponding to the print name of x. Thus (GETOBJ(MAKSTRNG(EXPLØDE x))) returns x if x was on the OBLIST, otherwise a new atom with the same print name.

Fields in CSSWYM used by Output Routines:

PRPT. Pointer to location to store next character to be printed. Intitialized by TERPRI and incremented by PUTCH.

PRPEND. Address of character just beyond last character in print line. PUTCH calls TERPRI if PRPT reaches PRPEND. Intitialized by TERPRI.

PRLNG. This constant is the length of the print line. Normally 132, it can be changed for different buffer lengths or a wider right margin.

PRATBAD. Used by PRIN1 to print the message '?TYPx' for atoms with type $x \in \{2, 3, 4, 5, 6, 7\}$. (That is, for atom types for which no print routine has been defined).

F.4 STUTTER Routines

COND, EVAL, GET, PUTPRØP, REMPRØP, QUØTE, SASS@

(COND $\ell_1, \ell_2, \ldots \ell_n$). This FSUBR CØNDitionally evaluates an expression. Each sublist must be a list-of two expressions. The first expression in each successive sublist is EVALuated until one is found that is not NIL. The second expression of the selected sublist is EVALuated and returned as the value of CØND. If all first expressions are NIL, error CN is signaled.

(EVAL x). EVALuates and returns the value of the s-expression x. Complete details of EVAL are in Appendix D.

GET, PUTPRØP, REMPRØP. Symbol atoms have an associated list called a property list. On this list the different 'properties' of the atom are stored, each under different names, called 'indicators.' The indicators must be symbol atoms. The properties may be any s-expression. In the initial free storage, only the properties for SUBR and FSUBR indicators occur. Function definitions can be stored under EXPR and FEXPR. Other properties and corresponding indicators can be defined at the Stutter programmers'convenience. The only restriction is that the above three functions are the only ones allowed to access the property list. This is because PUTPRØP and REMPRØP replace element pointers with rst pointers in some case.

(GET a i). This SUBR has two arguments: an atom and an indicator. It searches the property list of the atom for the indicator and returns the corresponding property value. If the indicator is not found, GET returns NIL.

(PUTPRØP a p i). This SUBR has three arguments: an atom, a value, and an indicator. The value is stored under the indicator on the property list of the atom. If the indicator existed on the property list, the pointer at the old value is replaced with a pointer at the new value. Otherwise, the indicator and value are placed at the front of the property list. Currently, the value of PUTPRØP should not be used. It should be changed to return the atom.

(REMPRØP a i). The arguments of this SUBR are an atom and an indicator. The indicator and the corresponding value are removed from the property list of the atom. REMPRØP returns the atom. Currently, REMPRØP ignores (does not delete) function definitions stored in the value cell.

(QUØTE x). This function is an FSUBR. Its arguments are passed as an unEVALuated list to the quote routine. If the list has one element, QUØTE assumes that the normal LISP 1.5 QUØTE was desired. If the list has more than one element, QUØTE simply returns the list. Both (QUØTE A B) and (QUØTE (A B)) return the value (A B).

(SASSØC x pl). This SUBR expects an expression (usually an atom) and a list of dotted pairs as arguments, The list is searched for a pair whose FST is EQ to the expression. The value of SASSØC is RST of the selected pair. If the expression is not found, the value of SASSØC is the atom at the end of the list of pairs, Usually, this atom is NIL, but this is up to the creator of the list of dotted pairs.

F.5 Utility Routines

BELL, ERRØR, CC

All these routines are SUBR's.

(BELL x). The argument must be a number. BELL rings the bell on the 2250 twice. The interval between the rings is specified by the argument, in hundredths of seconds (200 represents delay of 2 seconds). To use this routine, a DD card must be provided assigning SWYMSCOP to a 2250. The value of BELL is NIL. (Until registers B and L are assigned other than 14 and 15, BELL causes an abnormal termination,) (ERRØR x). This routine prints its argument and exits to the top level of the 'Stutter interpreter. The stack is not unwound, so variables retain the values they had at the time of the error.

(GC). A call on GC causes a garbage collection. The value of GC is NIL. It may be advantageous to call GC at times, because garbage collection is much less expensive when the amount of active 'storage is low. GC is described in detail in Appendix E.

Appendix G. Miscellaneous Swym Routines

The routines in this section are available within Swym but not to Stutter programs. Unless otherwise stated, a routine is called with CAL, but most have non-standard calling sequences: either they pass numbers rather than pointers or they are not called with CAL. Such non-standard routines are justifiable in limited contexts to avoid using free storage and to speed processing.

STIME, TTIME. These routines provide access to the ØS task timer. STIME Starts the TIMEr. It has no argument, but returns the value of any argument supplied. (i.e., STIME does not modify Al.) TTIME reports the elapsed Task TIME (in hundredths of a second) since the last execution of STIME. The result of TTIME is left in register Al. (Not a pointer to the result, the result itself.)

STAKN. This routine allocates a plex. The argument in Al is the number of bytes to be allocated; it must be a multiple of four. The value of STAKN is a pointer to the newly allocated plex. The calling routine must store a valid plexhead in the newly allocated plex. The name "STAKN" has nothing to do with the stack. It refers to a System function to TAKe N bytes from free storage. Note that STAKN can cause garbage collection: all pointers which are to be garbage collected must be in the stack when STAKN is called.

There is currently a major bug in STAKN. When the garbage collector is called, one of the pointers on the stack is to the new plex. But it is not an atom pointer nor is there a plexhead in the plex. There is no indication to the garbage collector of the type and extent of the allocated plex. The best correction is to have STAKN call the garbage collector before allocating the storage. The argument to STAKN would be made odd and saved on the stack.

NLENGTH. The single argument to this routine is a list (or atom) in Al. The result of NLENGTH is the number of elements in the argument. The number, rather than a pointer, is left in Al. The length of an atom is zero.

PUTSTR. PUTSTR PUTs a character STRing atom on the current output line. If its argument is not a character string atom, PUTSTR calls ERRØR. If the string is too long to fit on the current line and short enough to fit on a full line, PUTSTR calls TERPRI to terminate the current line. PUTSTR uses PUTCH (in CSSWYM) to transmit characters one at a time to the print line.

INIT, FINISH. INIT is the INITialization routine. It is entered from ØS, saves the registers, and initializes the registers for swym. It also opens data sets, sets the memory control pointers and calls STIME to start the timer. INIT exits to MAIN, the Stutter interpreter loop. Control is returned to ØS by FINISH. When the end of the input file is recognized, EØDAD in CARDRDR sends control to NOCARDS, which transfers control to FINISH.

FINISH prints some information for debugging, and abnormally terminates. When debugging is complete, FINISH will close all data sets and terminate normally.

SWERROR. This routine prints ERRØR messages for SWYM routines. Its argument is two characters in the low order two bytes of register L. SWERRØR is called by a simple branch. It changes the two characters to a character string atom, and calls ERRØR with that atom as its argument. SWERRØR is designed so that changing it to ABEND rather than call ERRØR will preserve all registers as they were at the time of the error. It is also possible to get very useful results if ERRØR prints all registers.

TRUE, FALSE. These two routines are called with a simple branch. They set Al to T and NIL, respectively, and execute a return. These routines save a little code in predicates like NULL and ATOM. These can exit by branching to TRUE or FALSE, thus avoiding two load instructions and the code for return (RET).

PUTCH. This routine PUTs one CHaracter into the current print line. The character must be in the low order byte of register A4. PUTCH is called with the instruction

```
BAL   L,PUTCH
```

This avoids several instructions for each character output. If the current character fills the output line, PUTCH calls TERPRI to print the line. PUTCH modifies only register TT.

Appendix H. Swym - Stutter Initial Free Storage

When Swym is loaded there are three classes of structure in the free storage area: character objects;' function names, and special structures. Each of these is described in a separate section below. The cards used to create the initial free storage are shown in Figure H.l.

H.1 Character Objects

As indicated in Appendix C (Read Routines), there are 64 character objects in SWYM. Each input character is converted into one of these 64 objects. These objects include A-S, V-Z, 0-9, +, 1, $,¬,/,?, :, #, ", ¢, !, 0-2-8, *, =, _, <, >, @, -, ·, ;, ,),, 1),, and ','. These character objects are assembled with the macros CHAR and QCHAR. For various reasons, other means are used to assemble the character objects for T, blank, apostrophe, and ampersand.

H.2 Subroutine Objects

All subroutines available to Stutter programs must be represented in initial free storage. There is one atom for each subroutine described in Appendix F. Subroutine atoms are assembled with the SUBR and FSUBR macros.

H.3 Special Structures

NIL,T. These two atoms are used by Stutter to represent the Boolean values <u>false</u> and <u>true</u>. Each has a predefined value equal to. itself. Thus, (EVAL(QUOTE NIL)) is NIL; but one can also say (EVAL NIL) and get NIL.

ØBLIST. The predefined value of this atom is a list of all symbol atoms active at any given time. This list is a list of 64 sublists. An atom is placed on a sublist chosen by hash coding the atom's print name. This speeds up the read routine search to find an existing instance of an input atom (in GETØBJ). The hashing function is

((length of pname) + 2*(last character) + 3*(first character) + 13*(third character)) mod 64 ,

where the characters are represented in EBCDIC. If the third character is absent, blank is used. This function seems to distribute the atoms fairly well, although there is a slight preference for bin 32.

The value of ØBLIST is treated as though it were an array. That is, the proper sublist is accessed by address arithmetic rather than successive RST operations. There is the danger that the garbage collector could convert this list into two or more lists connected by RST pointers. To avoid this, no variable should ever point at a portion of the object list.

CHARØBS. The predefine8 value of this atom is the list of all character objects. This list has 256 elements, one for each possible EBCDIC byte pattern. All illegal characters point at the character object for blank. Like ØBLIST, the character object list is referenced (by READCH and IVCCH) as though it were an array. Again, no variable may point at a portion of the character object list.

SUBR, FSUBR, EXPR, FEXPR. These atoms represent properties which can be PUTPRØP and which the system must know about. Specifically, each represents some form of function definition. To use an atom as a function, EVGET looks for one of these indicators on the property list and uses the corresponding value as the function definition. See further description in Appendix D.2.

FPRØPS. This is a structure:

```
((SUBR . 1) (FSUBR . 2)
 (EXPR . 3) (FEXPR . 4))
```

EVAL uses this structure at various points to associate a bit pattern with one of the indicators for a function definition. If an atom has a function definition, the appropriate bit pattern will be in the CELFNC field of the plexhead. This structure cannot be accessed by Stutter programs.

'UNBØUND'. This is simply a character string atom. It is the value of any atom that has not been assigned a value by one of

initial value
variable binding
function definition.

If 'UNBØUND' is the value of an atom, EVAL signals error El and terminates processing of the current s-expression.

Figure H.1

```
UNBOUND    STRAT     C'UNBOUND'
BLANK      QCHAR     ' '
NIL        VALUE     NIL,NIL
TRUTH      VALUE     T,TRUTH
OBLIST     VALUE     OBLIST,OLST
CHAROBS    VALUE     CHAROBS,COBS
SUBR       MATOM     SUBR
FSUBR      MATOM     FSUBR
EXPR       MATOM     EXPR
FEXPR      MATOM     FEXPR
           SUBR      FST,RST,TAK2,GC
           SUBR      ATOM,EQ,NULL,PRINT,PRIN1,TERPRI
           SUBR      READ
           SUBR      ERROR,STIVCCH,IVCCH,READCH,STIVQMO,IVQMO
           SUBR      GETOBJ,MAKSTRNG,EJECT
           SUBR      EVAL,SASSOC,EXPLODE,GET,PUTPROP,REMPROP
           FSUBR     COND,QUOTE,LIST
           SUBR      BELL
           CHAR      A,B,C,D,E,F,G,H,I ,J ,K,L,M,N,O,P Q,R,S,T,U,V,W,X,Y,Z
           CHAR      0,1,2,3,4,5,6,7,8,9
           CHAR      +,|,$,¬,/,%,:,#,",¢,!,o-2-8,*,=,_,<,>,@,-,.,;
           QCHAR     [illegible]
```

Appendix I. Swym Register Assignments

All the general registers are assigned names under Swym. About half are available for general use, while the remainder have specific uses. Although the register currently assigned to each name is listed, these assignments must be changed to better cooperate with ØS.

Register	Name	Use
0	N	Contains a pointer to the atom NIL.
1-6	A1-A6	Arguments to SUBR's; Stutter routines return results in A1; otherwise available for general use. Always six consecutive registers.
7	C4	Must always contain F'4'.
9	S	Permanent base register for addressing system data, transfer vectors, and a few basic routines.
10, 11	T, TT	An even-odd pair of temporary registers. TT is used by ATØM and PUTCH.
12	F	Free storage pointer - next word to be allocated.
13	P	PUSH down list pointer - last word which was allocated. See Appendix B.4.
14	B	Base for all routines
15	L	Linkage, holds return address on entry to a routine.

The user may alter A1-A6, T and TT with impunity. The following rules must be observed:

1. No register contents are garbage collected. If something must be collected, it must be in the stack. The garbage collector destroys all temporary registers.
2. A calling routine is responsible for saving any registers which might be destroyed by a called routine.

Appendix J. Swym - Stutter Output and Error Messages

There are four classes of output:

1) Normal

2) Read Error

3) Computation Error

4) ABEND - Abnormal Terminations

Each of these will be discussed in turn.

J.1 Normal output

Normally, the Swym system running Stutter reads an s-expression, evaluates it, and prints the value. All cards read are printed beginning in column 24 of the print line. After reading, the time since the start of processing this s-expression is printed (in 100ths/sec.). Next appear any lines PRINTed during EVAL. After EVAL, the total time since starting to read the s-expression is printed (in 100ths/sec.). Finally, the value of the expression is printed, followed by a blank line. At any time, the garbage collector may be called. It will produce a line of output as described in appendix E.

5.2 Read Errors

While reading cards, certain syntax errors are indicated. In all cases the read routine proceeds in some manner, usually by ignoring the error. The read error message includes a pointer ('<') beneath the next character to be scanned. Usually the character in error is immediately to the left.

Error Code	Routine	Error	Action
R0	RDSE	missing right super-paren -'>';	start skipping s-expressions
R1	RDSE	end of skipping chars for R0;	reading continues
R2	READ and RDSE	missing right parens ')'- inside super-parens;	right parens created; number is printed at the far right
R3	RDLIST	extra dot between list elements;	ignored
R4	RDSE	R0 occurred while skipping for earlier R0;	skips for inner R0 then back to skipping for outer R0
R5	RDAT	igl char in X'...', W'...', or B'...';	invents quote before the error character this may confuse the scanner
R6	RDAT	C'...' but should use Z'...';	Z'...' assumed
R7	RDAT	B'...' but should use W'...';	W'...' assumed
R8	RDAT	x' appears where $x \notin \{W, X, Z, C, B\}$;	quote ignored, atom with print name x is produced; beware, the scanner may become confused.
R8	GETCH	inside quotes but no '-' in 72;	stays in quote mode
RA	GETCH	non-blank in 72 outside quotes	ignored
RB	RDAT	too many digits (9) in integer;	this and all after ignored
RC	RDSE and RDLIST	igl char at start of s-expr; igl char after '<' or '('; igl char between list elements;	ignored

READ ERRORS

J.3 Computation Errors ,

These errors terminate evaluation of the current s-expression. Variables are not rebound; this means that global variables may not have their correct value and also that list structure may be saved unnecessarily. Swym continues after these errors by evaluating the next input s-expression.

Error Code	Routine	Error
BI	BINDERY	trying to bind atom with function definition in cell
CN	CØND	no predicate was true
Ex	EXPLODE	argument not symbol atom (type 0)
El	EVAL	arg was unbound atom atom at front of s-expr was not symbol (type 0) atom at front of s-expr had no function definition atom at front of s-expr had illegal function definition type (system error) more than six arguments to a SUBR more than one formal argument in FEXPR definition
Ml	MAKSTRNG	argument was not a list of atoms each having a one character print name
PP	PUTPRØP	first argument not a symbol atom (type 0)
Pl	PUTSTR	argument not character string atom (type 1) (system error)
RI	STIVCCH	argument's print name not one character
RJ	GETØBJ	argument not a character string atom (type 1)

J.4 ABEND - Abnormal Terminations

These errors are always fatal and produce a dump if a //SYSUDUMP DD card has been included. Most are concerned with errors in the garbage collector and indicate that the data structure was illegal. Further computation on an erroneous data structure can produce nothing useful.

Completion Code	Routine	Error
System 0C6	FST,RST	Fst or rst taken of an atom
User 7.	FINISH	During debugging, normal termination
20	PBØPEN,PUTBYTE	Insufficient memory remaining after garbage collection
20	CØLLECT	Argument already marked with ml
28	ATCØL	Illegal atom type
2E	CØLX	Atom A ¬ml ∧ m2
3E	CØLX	Atom A ml A ¬m2
6E	CØLX	¬atom A m2 A ¬ml
7E	CØLX	¬atom A ¬m2 A ml
7E	GC	stack pointed at an ml A ¬m2 or ¬ml ∧ m2 word
118	CØLLECT	in second pass, found atom ∧ ¬ml ∧ ¬m2
122,126	Gc	invalid stack block type
15A	CØLLECT	in second pass, found ¬atom ∧ ¬ml ∧ ¬m2
1A8	CØLLECT	in second pass, found rst: atom A ¬ml A ¬m2

Appendix K. Proposed Instructions for the IBM/360

The instructions proposed in this appendix are intended to give the flavor of possible additions to the 360 instruction set. A completely different machine design might be preferrable, but would mean reprogramming on the scale accompanying introduction of the 360. Additions to the instruction set would not obsolete any existing programs, except in that they could be written more compactly in the proposed extended instruction set. The instructions are proposed in terms of the 360 because to a large extent they then also apply to most traditionally designed computers. Thus, although these instructions might make radical changes in program design (more modularity), the basic design of computers need change very little.

Four sets of proposals are included below:

Loads and Stores

Associated-Bit Instructions

Stack Instruction

Subroutine Linkage

The last two are interdependent, but otherwise these instruction sets could be added individually.

Proposed Loads and Stores

These instructions are intended to remove some of the more annoying limitations of the 360. They have been proposed many times, especially in [Wrth 68].

LHL	(RX) Load Halfword Logical
	The halfword at D_1 (X_1,B_1) replaces the low order

16 bits of register R_1. The upper 16 bits of R_1 are unaffected.

STHA (RX) Store Halfword Arithmetic

If bits 1-16 of R_1 do not all match the sign bit, this instruction causes a fixed point overflow. Otherwise, the low order 16 bits are stored in the halfword addressed by $D_1(X_1, B_1)$.

LI (AI, SI) (RX) Load (Add, Subtract) Immediate

A thirty-two bit quantity is computed from D_1 plus the contents of registers X_1 and B_1, treated as signed numbers. The resulting quantity is loaded (added, subtracted) to register R_1. AI and SI may cause fixed point overflow.

LIR (AIR, SIR) (RR) Load (Add,Subtract) Immediate Register Field

These instructions are similar to LI (AI, SI) except that the quantity loaded, added, or substracted is the R_2 field of the instruction (<u>not</u> the contents of that register).

LIN (STIN) (RX) Load (Store) Indirect

The $D_1(X_1,B_1)$ field refers to a word in memory. The contents of this word are used as the address from which to load or to which to store the contents of R_1.

<u>Proposed Associated-Bit Instructions</u>

There are many uses in higher level languages for non-numeric bits associated with the words of memory. This proposal describes one set of instructions for manipulating these bits. It is assumed that one bit is associated with every byte of memory, but that the most common use will be to use all four bits for each word. Four bits are also associated with each general register. Any instruction not specified below does not alter the bits in memory or in a general register. This means that a floating point field, for example, remains marked as such as long as only floating operations are used on that field.

MVB	(SS) Move Bits The bits associated with the L + 1 words starting at $D_2(B_2)$ are moved to the bits for the L + 1 words starting at $D_1(B_1)$. The operation proceeds from left to right by word. Both addresses must be on word boundaries. $0 \leq L \leq 255$.
MVSB	(SS) Move Single Bits The bits associated with the L + 1 bytes starting at $D_2(B_2)$ are moved to the bits for L + 1 bytes starting at $D_2(B_2)$. The operation proceeds from left to right. $0 \leq L \leq 255$.
TMB,NIB,OIB, XIB,MVIB	(SI) These instructions correspond to the normal instruction without the 'B' suffix. The difference is that the four low order bits of the mask correspond to the four bits associated with the addressed word. The address must be on a word boundary.

GBR	(RR) Get Bits from Register The four low order bits of R1 are replaced by the bits associated with R2. Bits 24-27 of R1 are zeroed; other bits are unchanged.
PBR	(RR) Put Bits from Register The bits associated with R1 are replaced by the four low order bits of R2.
PIB	(RR) Put Immediate Bits The bits associated with R1 are replaced by the contents of the R2 field.
LB	(RX) Load Bits The four low order bits of register R1 are replaced by the bits associated with the word at $D_1(X_1,B_1)$. The next four low order bits (24-27) are replaced by zero. The rest of the register is unchanged. $D_1(X_1,B_1)$ must specify a word boundary.
STB	(RX) Store Bits The four low order bits of R_1 replace the bits associated with the word at $D_1(X_1,B_1)$. The latter must specify a word boundary.
PB	(SS) Pack Bits The $D_1(B_1)$ field specifies the beginning of a field of L + 1 bytes. The low order four bits of each of these bytes is set from the bits associated with the corresponding word in the

$D_2(B_2)$ field. The latter is L + 1 words long. The high order four bits of each byte are zeroed. $D_2(B_2)$ must be on a word boundary. $0 \leq L \leq 255$.

UPB (SS) Unpack Bits

$D_2(B_2)$ specifies the start of a field of L + 1 bytes. $D_1(B_1)$ specifies the start of a field of L + 1 words. UPB reverses the process of PB by setting the bits associated with the words from the low order four bits of the corresponding byte. $D_1(B_1)$ must specify a word boundary. $0 \leq L \leq 255$.

TSB (RX) Test Single Bit

The low order bit of the condition code is set from the bit associated with the byte at $D_1(B_1)$. The high order bit is set from the bit associated with the other byte in the half-word of which $D_1(B_1)$ is part. If $D_1(B_1)$ is even, the high order bit is set from the bit associated with $D_1(B_1) + 1$. If odd, then $D_1(B_1) - 1$.

TRTB (SS) Translate and Test Bits

The four bits associated with the word at $D_1(B_1)$ et sequens are used to index into the table at $D_2(B_2)$. The table need have only 16 entries. Termination and condition code setting are as for the instruction TRT.

L	(RX) Load This instruction is identical to the normal load instruction, except that the bits associated with the target register are set from the bits associated with the word in memory.
LR, LNR, LPR, LTR	(RR) The bits of the target register are set from the bits of the source register.
LM, STM	(RX) The bits of the target are set from the source.

Proposed Stack Instructions

The problem with using a stack on the 360 is that code must be generated to test for the ends of the stack. These instructions manipulate the stack and test for the beginning and end. In all cases, the R_1 field indicates a register containing a stack pointer. This register always points to the latest word added to the stack. The register is decremented for each entry, so all recent entries can be addressed relative to the stack pointer. The $D_1(B_1)$ field of the instructions is assumed to be the address of a two word Stack Control Block. The first word of the block is the address of the first entry in the stack, the second word is the address of the last allowable entry in the stack. This control block is used to check for the ends of the stack. Stack instructions can generate two new interruption types; stack overflow and stack underflow.

QR (RX) Queue Register on Stack

The contents of R_1 are decremented by four and compared against the contents of the word addressed by $D_1(B_1)$. If less-than, then a stack overflow interrupt is generated. Otherwise, the contents of the R_2 are stored at the location indicated by the revised contents of R_1.

QMI (RX) Queue Multiple Immediate

The R_2 field is multiplied by four and subtracted from R_1. The result is compared against the contents of the word addressed by $D_1(B_1)$. If less-than, a stack overflow interrupt is generated.

UQR (RX) Unqueue Word from Stack

The contents of R_1 are compared against the contents of the word at $D_1(B_1)+4$ if greater-than or equal, then a stack underflow interrupt is generated. Otherwise, the contents of R_2 are replaced by the word addressed by R_1. Finally, R_1 is incremented by four.

UQMI (RX) Unqueue Multiple Immediate

The R_2 field is multiplied by four and added to R_1. The result is compared against the contents of the word at $D_1(B_1) + 4$. If greater-than, a stack underflow interrupt is generated.

QDR, QER, UQDR, UQER (RX) Queue Double Floating Register

Queue Short Floating Register

Unqueue Double Floating Register

Unqueue Short Floating Register

These are analogous to QR and UQR except that they use the floating registers. Also, QDR and UQDR modify the R_1 register by eight rather than four.

Proposed Subroutine Instructions

CAL (SS) Call a Subroutine

The R_1 and $D_1(B_1)$ fields refer to a stack. These fields are used to QR the program counter. The R_2 register is loaded with the word indicated by $D_2(B_2)$. The program counter is loaded with the same word so that execution begins at the address in R_2.

RET (SS) Return from a Subroutine

The R_1 and $D_1(B_1)$ fields refer to a stack. UQR is executed from this stack and the top element is loaded into the program counter and into R_2. The displacement D_2 and the contents of B_2 are added to the program counter.

Appendix L. Demonstration of the Correctness of the Swym Garbage Collection Algorithm

The Swym garbage collector is reasonably complex since the central routine, COLLECT, involves two loops and recursion. The potential user deserves some reassurance that COLLECT will not mysteriously modify his data. The problems of minor errors in garbage collectors are severe because the collector is called when storage is exhausted, and this depends on the data in the problem at hand. This appendix attempts to demonstrate the correctness of the COLLECT algorithm. But it is important to note that this demonstration proves nothing about the actual Swym system garbage collector. There are three reasons:

1) This is a demonstration of an algorithm. The program itself may or may not correspond to the algorithm. There is many a slip 'twixt conception and core; errors can occur in coding, keypunching, assembly, or during execution, when some other part of the system may modify COLLECT.

2) It is necessary for this proof.to make numerous assumptions about the effect of subsidiary functions. These are subject to the problems mentioned in (1). They are also subject to that fact that they are specified only in English, a not always precise language.

3) The proof itself is primarily in English. A gain in precision could be achieved by translating the proof into the predicate calculus; but even though more readers might be reassured, the number of readers would decline drastically.

Despite all the above, the demonstration of the correctness of the COLLECT algorithm is at least an interesting problem. Because of the involuteness and the fact that a given call depends on the correctness of higher level invocations as well as lower level invocations, the major problem is avoiding a circular proof.

Most of the functions used in COLLECT are defined elsewhere. The following are assumed as primitives: fst, rst, atom, rplf, and HD. The five operations on marking bits - ML, M2, MARK1, MARK12, and UNMARK1 - are all assumed to use two bit tables to associate two bits with each word. This is contrary to the implementation, but simplifies the demonstration somewhat. (A final note will show how to remove this restriction.) The properties of four functions must be presented in detail: ATCØL, GCPUT, FIXUP and CØLLECT. The properties of the first three will be assumed while the properties of CØLLECT are to be demonstrated. The relevant properties are listed in Figure L.4.

The CØLLECT algorithm in figure L.l has extra labels for reference during this appendix; otherwise, it is the same algorithm as given in appendix E. A flow chart is in Figure L.2, for those who read flow charts. The labels in L.l and L.2 will be used to refer to the relevant statement without specific reference to the figure. Several other types of references are made to items identified with a capital letter followed by one or more digits. This table summarizes the capital letters and the location of more information.

A	ATCOL property	See Figure L.4
C	COLLECT property	
F	FIXUP property	
G	GCPUT property	
L	a figure in this appendix	
M	marking bit	See Appendix E
S	statement label in Figure L.l	

The argument to CØLLECT is a list. CØLLECT processes as much of that list as can be represented in new core as a single sequence of consecutive words, where only the last is a rst pointer. This Dart of a list is called a list segment. Sometimes it is the entire list, -ending with a rst pointer at an atom. But if some rst of the list is already collected, the list segment must end with a rst pointer to the existing representation of that rst. For convenience, the pointers pointing at the elements of the list segment will be called fst pointers.

Each invocation of CØLLECT writes a list segment on the temporary file. After all structures are collected, this file is read in to replace list storage. It represents the same list structures as the old contents, providing that all pointers into list storage are modified to point to the new locations of the structures. The old contents of list storage are referred to as old core. The new contents, though stored temporarily on the file, are referred to as new core. For every pointer into old core, there is an equivalent pointer into new core. As CØLLECT processes a list segment, say x, it replaces fst (x) in old core with a pointer to the equivalent of x in new core. For example, the fst of the list (A B C) is replaced with a pointer to the same list in new core (not with a pointer to A in new core). This replacement is done with the rplf in S34. Later the pointer to the new core equivalent is accessed with the fst in S3422 or S422. These three statements are not operations on list structure in the sense normally understood by 'fst', but they are implementation independent in that they only require that fst return the value stored with rplf.

CØLLECT contains two loops: the first is all statements numbered S1x and S2x; the second is all statements S3x and S4x. S11 and S31 initialize the loops by setting r to a rst of the list (the list itself being considered the

0^{th} <u>rst</u>). Then the S1x and S3x statements process an element of the **list.** The S2x and S4x statements check the next successive <u>rst</u> and either **loop** back, or process the <u>rst</u> and terminate. Below, the first loop will be referred to as pass one and the second loop as pass two. This is because each makes one pass over the list segment.

Understanding CØLLECT requires knowledge of the state of the list segment, <u>x</u>, at S31. There are three cases:

1. Each pointer in the list points at a word with at least Ml. Each pointer has its own Ml. bit on and M2 bit off. The end of the list is signalled by a <u>rst</u> pointing at an atom.
2. Same as case 1, except that the final <u>rst</u> points at a word marked with both Ml and M2.
3. This case is like case 1, except that the final <u>rst</u> is a word that is marked with Ml and not M2. In addition, the element pointer to the last element has neither marking bit.

Pictorially these cases can be represented as in diagram L.3.

To illustrate the predicate calculus approach to this demonstration of correctness, here is the predicate that a list segment satisfies:

$$(\exists n)(L1 \wedge L2 \wedge L3)$$

where

$$L1 = \bigwedge_{i=1}^{n-1} (\neg M2(R(i)) \wedge M1(R(i))) \wedge \neg M2(R(n))$$

$$L2 = \bigwedge_{i=1}^{n} M1(\underline{fst}(R(i)))$$

$$\begin{aligned} L3 = {} & (M1(R(n)) \wedge (\overbrace{\underline{atom}(R(n+1))}^{\{case} \vee \overbrace{M2(R(n+1))}^{\{case\ a:}))) \\ & \left.\begin{aligned} & \vee(\neg M1(R(n)) \wedge M1(R(n+1)) \wedge R(n+1) \neq R(n)) \\ & \vee(R(n+1) = R(n) \wedge \neg M1(R(n))) \end{aligned}\right\} \quad \{case\ 3\} \end{aligned}$$

where

$R(i) = \underline{\mathbf{rst}}^{i}(x)$

$\underline{\mathbf{rst}}^{i}(p)$ = if i=0 <u>then</u> p else rst($\underline{\text{rst}}^{i-1}$(P))

x = argument to CØLLECT

The demonstration of the correctness of CØLLECT requires 3 steps. The first step is to show that CØLLECT terminates. This can be shown with minimal recourse to CØLLECT's properties. Secondly, assuming that CØLLECT is correct for all recursive invocations, CØLLECT is shown to have properties C1-C10. Finally, it is shown that the new core image is equivalent to the old core, and thus that CØLLECT is correct.

The first two steps are sufficient to show that CØLLECT writes out a list segment. For if CØLLECT terminated, at some level of recursion it did not call itself and thus did not depend on its own properties. The fact that CØLLECT also depends on the correctness of higher levels of recursion is dealt with in the third step.

Certain of the properties in L.4 are assumptions about the arguments to the relevant function. These are included for ease of reference, but they must be demonstrated each time the function is called. There are a few <u>global assumptions</u>:

1) At the time CØLLECT is first called, for a given garbage collection, there are no marking bits set; all words w satisfy $\neg M1(w) \wedge \neg M2(w)$.

2) When CØLLECT is called by the garbage collector or ATCØL, its argument satisfies C0.

3) No pointer in memory points at a word with the <u>rst</u> bit on.

Lemma 1. CO is always satisfied.

By the second global assumption above, CO is satisfied when CØLLECT is called externally. When CØLLECT is called at S142, its argument is neither an atom nor marked M1 because of the tests in S14. Thus to violate CO, t in S142 must be $\neg$M1(t) $\wedge$ M2(t). But by the first global assumption above this word was not so marked at the beginning of garbage collection. Consequently, it must have been created by earlier or concurrent calls on CØLLECT. These calls must have included execution of S35 to turn on the M2 bit and a subsequent call on S223 to turn off the M1 bit that is also set at S35. (S35 is the only statement turning on M2 and S223 is the only statement turning off M1). But by the test before S222, S223 cannot be executed for a word with the M2 bit. Consequently, a word satisfying $\neg$M1(t) $\wedge$ M2(t) cannot exist. Thus S142 cannot violate CO, and the lemma is proven.

Lemma 2. At S12, r is unmarked and non-atomic.

This is true on entry to CØLLECT, by Lemma 1. Thereafter, the lemma is true by the tests in S22, which terminate pass one if the next r would be atomic or marked.

Lemma 3. S223 unmarks the last word marked at S13; a word previously unmarked.

No statements modifying r occur between S223 and S13 (assuming the Algol interpretation of variable binding). The second assertion follows from lemma 2.

Lemma 4. M2(w) $\supset$ M1(w).

This is initially true since it is assumed that there are no M2 bits set. Thereafter, it remains true since M2 can only be set by S35 and that statement also sets M1. The M1 cannot be unmarked by S223 as shown in the proof of lemma 1.

I. CØLLECT Terminates

Lemma 5. Each call on CØLLECT sets at least one previously zero M1 bit.

By lemma 2, the argument to CØLLECT, x, is not marked with M1. It is so marked by S13. If S223 is not the path chosen through S22, then x remains marked with MI. If S223 is executed while r = x, then x is unmarked, but is marked again at S35. In either case, x remains marked with M1 by lemma 3 and A.5.

Lemma 6. The recursion in S142 is always to a finite depth and therefore terminates.

By lemma 2, a previously unmarked word is marked at S13. But there are a finite number of words in memory (otherwise the garbage collector would not be called and its correctness would not matter). By the test before S142, CØLLECT does not recur if what would be its argument is already marked. Since every time CØLLECT is called there are fewer words not marked with M1, CØLLECT cannot recur indefinitely.

Lemma 7. The loop in pass one terminates.

At S2242 the loop returns to chkloop, that is, S12. But then S13 marks a previously unmarked word (by lemma 2). Since at each execution of S13 there are fewer words unmarked with M1, the loop terminates. Note that if S223 unmarks a word, the loop is terminating since S2242 will not be executed.

Lemma 8. The loop in pass two terminates.

By lemma 1, x is not marked with M2 after S31. But that x is marked with M2 after S35. The loop terminates at S422 if t is marked with M2, but r is assigned the value of t in S4231, just before looping back. Therefore S35 again marks a word previously unmarked with M2. Since there are a finite number of words not marked with M2, the loop must terminate at S422, if not sooner.

Theorem 1. COLLECT terminates.

Assuming that all subsidiary functions terminate, the theorem follows from lemmas 6, 7 and 8.

II. Collect has properties C1-C10.

In this section the inductive assumption is made that all subsidiary calls of CØLLECT satisfy C0-C10 if they terminate.

Lemma 9. Pass one has properties C1-C4.

The words constituting the list segment are those pointed at by successive values of r. S13 sets the M1 bit in that word, thus satisfying C2. C1 is satisfied by S14:

If t (= fst(r)) is atomic then A1 is satisfied for S141 and t is marked M1 by property A2 or A4.

If t is marked with M2, then it is also marked with M1 by lemma 4.

If t is marked with M1, there are two possible cases: t has been marked by a higher level invocation of CØLLECT, or t is a word in the list segment. In either case, t is indeed marked with M1, satisfying C1.

If t is unmarked, then it is marked with M1 since the lower level CØLLECT is assumed to satisfy C2.

S22 tests for termination of the list segment. If S221 is executed, then the list segment is an instance of case 1 in L.3. If S222 is executed, then this is an instance of case 2. If S223 is executed, then this is an instance of case 3, and the M1 bit in e_n is indeed set off, satisfying C4. If S224 is executed, then at least one more element pointer is to be included in the list segment. Each time through S224, all prior element pointers of the list segment satisfy C1 and C2, as shown above. The first pass eventually does terminate, by lemma 7, and can only terminate by one of the paths through S22 discussed above; thus C3 and C4 are satisfied.

Lemma 10. Pass 2 satisfies C5-C8.

The proof is by induction on n, the length of the list segment isolated in pass 1. Suppose n = 1. About half of the possibilities for this case are illustrated in L.5.

C5: one word is written for the one fst pointer in the list segment by S342.

C6: the address of the written word replaces the fst pointer in the list segment (statement S34).

C7: the word in the old core list segment is marked with Ml and M2 by S35.

C8: since n = 1, S42 writes a rst pointer in one of its branches, depending on which case of list segment has occured.

Case 1. The rst is an atom. In this case a pointer with the rst bit is written in S4211 or S4212.

Case 2. The rst is marked with M2. A pointer with the rst bit is written by S422.

Case 3. Note that m is false because there is no Ml bit with the last fst pointer (by C4). Thus S424 is executed and a word is written that will eventually contain a pointer and a rst bit.

Suppose n > 1. In this case, C5,C6, and C7 are satisfied for the first fst pointer by the same argument used for n = 1. By the structure of a list segment, rst (r) is neither atomic, nor marked with M2. Furthermore, m is true, because the Ml bit is always on for all fst pointers in the list segment other than the last. Consequently, S423 is executed and control returns to S32 with r pointing at the rst of the original list segment. But rst of a list segment of length greater than 1 is a shorter list segment, so the induction is satisfied. Thus the lemma is demonstrated.

Lemma 11. (C9)

CØLLECT does not modify any word marked with M1 by any other routine or other invocation of CØLLECT.

There are seven statements in CØLLECT that modify marking bits or words in old core: S13, S141, S142, S221, S223, S34, and S35. The lemma will be demonstrated for each in turn.

S13 (MARK1(r)) By lemma 1, this word was previously unmarked.

S141 and S221 (ATCØL(t)) By the tests preceeding these statements, A1 is satisfied. Hence, ATCØL satisfies A5 and A2, modifying no word previously marked with M1.

S142 (CØLLECT(t)) t is neither atomic nor marked by lemma 4 and the tests in S14. Thus C0 is satisfied and by assumption the lower level invocation of CØLLECT is correct. Therefore S142 satisfies C9 because the lower level CØLLECT does.

S223 (UNMARK1(r)) By lemmas 2 and 3, this statement unmarks a word that was unmarked prior to S13.

S34 (rplf (r; . ..)) As shown in the demonstration of lemma 10, r is part of the list segment and it was marked with M1 by pass one of the current invocation of CØLLECT.

S35 (MARK12(r)) Similarly to S34.

Lemma 12 (C10)

Any word marked M1 either contains or will contain the address of the equivalent word in new core.

When the equivalent address is placed in the word by S34, the word is marked M1 (and M2) by S35. By C9 and A5, this word is not thereafter modified by any other routine. If M2 is off, then M1 was set by S13. But by C5 and

C6 the address of the new core equivalent will be placed in this word.

Theorem 2. COLLECT has properties C1-C10.

Lemmas 9, 10, 11, and 12 were demonstrated with the assumption that all lower level calls of CØLLECT were correct. But if the recursive call terminates, then at some level CØLLECT did not call itself. Thus at this level correctness can be demonstrated without reference to lower level calls of CØLLECT. Consequently, this lowest level is correct. The correctness of the outermost level can be proven by induction on the depth of recursion. But by Theorem 1, CØLLECT terminates. Consequently, by Lemmas 9,10,11, and 12, CØLLECT has properties C1-C10.

III. The New Core Image is Isomorphic to the Old

The isomorphism to be demonstrated will be written $x \cong y$ and defined by

$$x \cong y = (\underline{\text{if}}\ \text{atom}\ (x)\ \underline{\text{then}}\ \text{atom}\ (y) \wedge x = y$$
$$\underline{\text{else}}\ \underline{\text{fst}}\ (x) \cong \underline{\text{fst}}(y) \wedge \underline{\text{rst}}\ (x) \cong \underline{\text{rst}}\ (y))$$

where $x = y$ is the isomorphism induced by ATCØL. If x is a word in old core marked with M1 and M2, then by C6 that word contains the address of the equivalent word in new core. This equivalent word is denoted by x'. It is necessary to demonstrate that after garbage collection (but before reading the new core) $(\forall x)\ (M1(x)) \supset (M2(x) \wedge x \cong x')$. The proof will be by induction on n the length of the list segment in new core. This length is the number of words from x' (including x') to the next word in memory with'a $\underline{\text{rst}}$ bit.

Lemma 14. $M2(\underline{x}) \supset \underline{\text{if}}\ \text{atom}\ (x)\ \underline{\text{then}}\ HD(x) = x'\ \underline{\text{else}}\ \underline{\text{fst}}(x) = x'$ and the value of x is not modified, nor is the M2 removed, by CØLLECT or any subsidiary function.

By A4,C6, and C7, $\underline{x}'$ is written into $\underline{x}$ at the same time that $\underline{x}$ is marked with M2. By lemma 4, $M2(\underline{x}) \supset M1(\underline{x})$; but if $M1(\underline{x})$ then $\underline{x}$ is not modified as guaranteed by A5 and C9.

Lemma, 15. S342 has the effect of GCPUT $(\underline{t}')$, where $\underline{t} = \underline{\text{fst}}(\underline{r})$.

Note that by definition FIXUP executes GCPUT; so every branch of S342 executes GCPUT exactly once. By A4, S34211 does GCPUT $(\underline{t}')$ if $\underline{t}$ is an atom marked M2. By C6 and C7, S3422 does GCPUT $(\underline{t}')$ if $\underline{t}$ is non-atomic and marked with M2. S34212 does GCPUT. (0) but establishes a fixup so that the zero will be replaced by the contents of $\underline{t}$ after CØLLECT. But by A3 and A4, $\underline{t}$ will contain t'. Similarly S3423 does GCPUT (0) and establishes a fixup. By C1, t is marked with M1 (and not M2 because of test before S3422); but by C10 that word will contain the address of its new core equivalent. Thus in each branch of S342 either t' is written or a fixup is generated so that the written word will contain t'.

Lemma 16. S4211, S4212, S422, and S424 have the effect of GCPUT (t' v rstbit) where t = rst (r).

S4211: By A4, HD(t) contains the address t'.

S4212: By A3 and A4, HD(t) will contain the address t'. Since the fixup processing routine or's the fixup into the word in new core, rstbit remains in the word.

S422: By C6, fst (t) is t'.

S424: Since m is false, this must be a case 3 list segment. (The only case having ¬M1(R(i)).) But in this case, by the test before S223, the rst (r) is marked with M1 and by C10 will contain t'. Consequently, the fixup process will create a correct rst pointer to t'.

Theorem 3.

After CØLLECT, any word, x, marked M2 is also marked M1 and contains a pointer to the equivalent word, x', in new core satisfying $x \cong x'$.

If x is an atom, then CØLLECT called ATCØL if it processed x. By A4, x' is atomic and x = x'. If x is not atomic, then by the properties of pass two, x' is not atomic. The proof that $x \cong x'$ is by induction on n, the- number of pointers from x' (and counting x') to the next word with a rst bit. Note that x' was marked by S35 and x' was written by S342 which never puts in a rst bit.

n = 1.

fst ≅ fst(x'). By lemma 15, x' was effectively written with GCPUT(t') where t' is the address of the equivalent of t and t = fst(x).

rst(x) ≅ rst(x'). Since n = 1, the word following x' has a rst bit and thus contains the pointer at rst(x'). But any word with a rst bit must have been written with S42. By lemma 16, any word written with S42 was effectively written with GCPUT((rst(r))' v rstbit). But r was not modified between S23 and S42 so r indicated the same x whose fst was written out in S342. Thus rst(x) ≅ rst(x') because the latter was created from the former.

n ≥ 1.

fst ≅ fst(x'). By the same argument as the case above.

rst(x) ≅ rst(x'). Since n > 1, the word following x' has no rst bit and rst(x') is a pointer to that following word, that is, a pointer to the list segment of length n-1 starting at that following word. After x' was written, S423 was executed (otherwise the following word would have a rst bit). So S32 et sequens were executed with r pointing to rst(x), creating a list segment of length n-1. By the induction, the shorter list segment is equivalent to rst(x). Consequently rst(x) ≅ rst(x').

Thus in all cases, CØLLECT creates a correct representation of its argument.

Note on the Implementation

The actual implementation of CØLLECT uses the M1 and M2 bits in the word itself as shown in figure I.2. The problem for the above demonstration is that the M2 bit is the same as the rst bit. Two changes are made in the algorithm: the arguments to all functions aremasked to remove possible marking bits and t := rst(r) is changed to

t := if M1(r+4) then r+4 else rst(r).

This note will show that the proof can be modified to take these changes into account and that the modified rst function is valid.

The proof of Lemma 1 depends on global assumption 1 that no marking bits exist before the first entry to CØLLECT (for a given garbage collection). But since there can be rst bits, global assumption1 does not hold. Instead, it must be changed to:

At the time CØLLECT is first called for a given garbage collection, there are no marking bits set in any fst pointers.

Thereafter, all discussion of marking bits must be qualified by reference to fst pointers only. But we have:

Lemma 0. COLLECT never sets Ml in a word with the rst bit.

Global assumption 3 states that no pointer into list storage, no fst pointer, and no rst pointer points at a word with the rst bit on. But the variables x, r, and t only acquire values from these three sources. Thus x, r, and t never point at a word with the rst bit on. But Ml is only set by S13 and S35 where the argument is r. Consequently the lemma is true.

Because of lemma 0, the modified global assumption 1 is valid. Furthermore, the extension to the _rst_ operation is justified; if the word following a given word has ML, it cannot be a _rst_ pointer and the pointer to r+4 is what _rst_ would return anyway.

Figure L.1

```
COLLECT (x) = begin list x,r,t; Boolean m;
    word rstbit := x'00000001';

S11: r := x;
   chkloop:
S12: t := fst (r);
S13:    MARK1 (r);
S14:    if atom (t) then
           S141:    ATCOL (t)
        else if ¬M1 (t) then
           S142:   --COLLECT (t);

S21: t := rst (r);
S22:    if atom (t) then
           S221:    ATCOL (t)
        else if M2 (t) then
           S222:
        else if M1 (t) then
           S223:    UNMARK1 (r);
        else
           S224:    begin
                      S2241: r := t;
                      S2242:    goto chkloop
                    end;
S31: r := x;
S32:    wrloop: m := M1 (r);
S33: t := fst (r);
```

```
S34:    rplf (S341:   r;
            S342:   if atom (t) then
                      S3421:   if M2 (t) then
                                 S34211:   GCPUT (HD (t))
                               else
                                 S34212:   FIXUP (t; 0)
                    else if M2 (t) then
                      S3422:   GCPUT (fst (t))
                    else
                      S3423:   FIXUP (t; 0));
S35:    MARK12(r);

S41: t := rst (r);
S42:    if atom (t) then
            S421:   if M2 (t) then
                      S4211:   GCPUT (HD (t) V rstbit)
                    else
                      S4212:   FIXUP (t; rstbit)
        else if M2 (t) then
            S422:   GCPUT (fst (t) V rstbit)
        else if m then
            S423:   begin
                      S4231: r := t;
                      S4232:   goto l o o p
                    end
        else
            S424:   FIXUP (t; rstbit)
end COLLECT
```

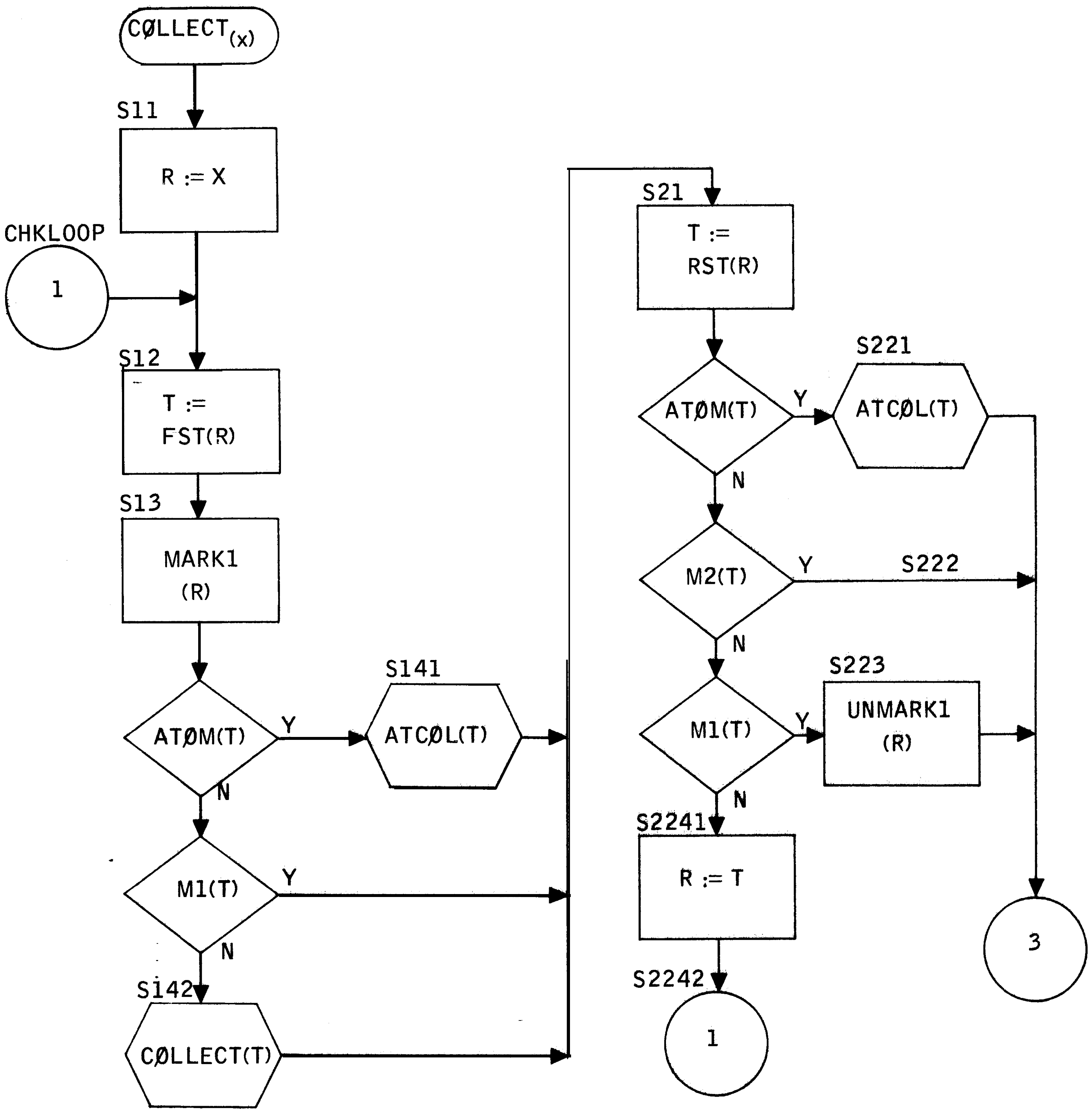


Figure L.2
=low Chart of CØLLECT

Figure L. 2 (Cont)

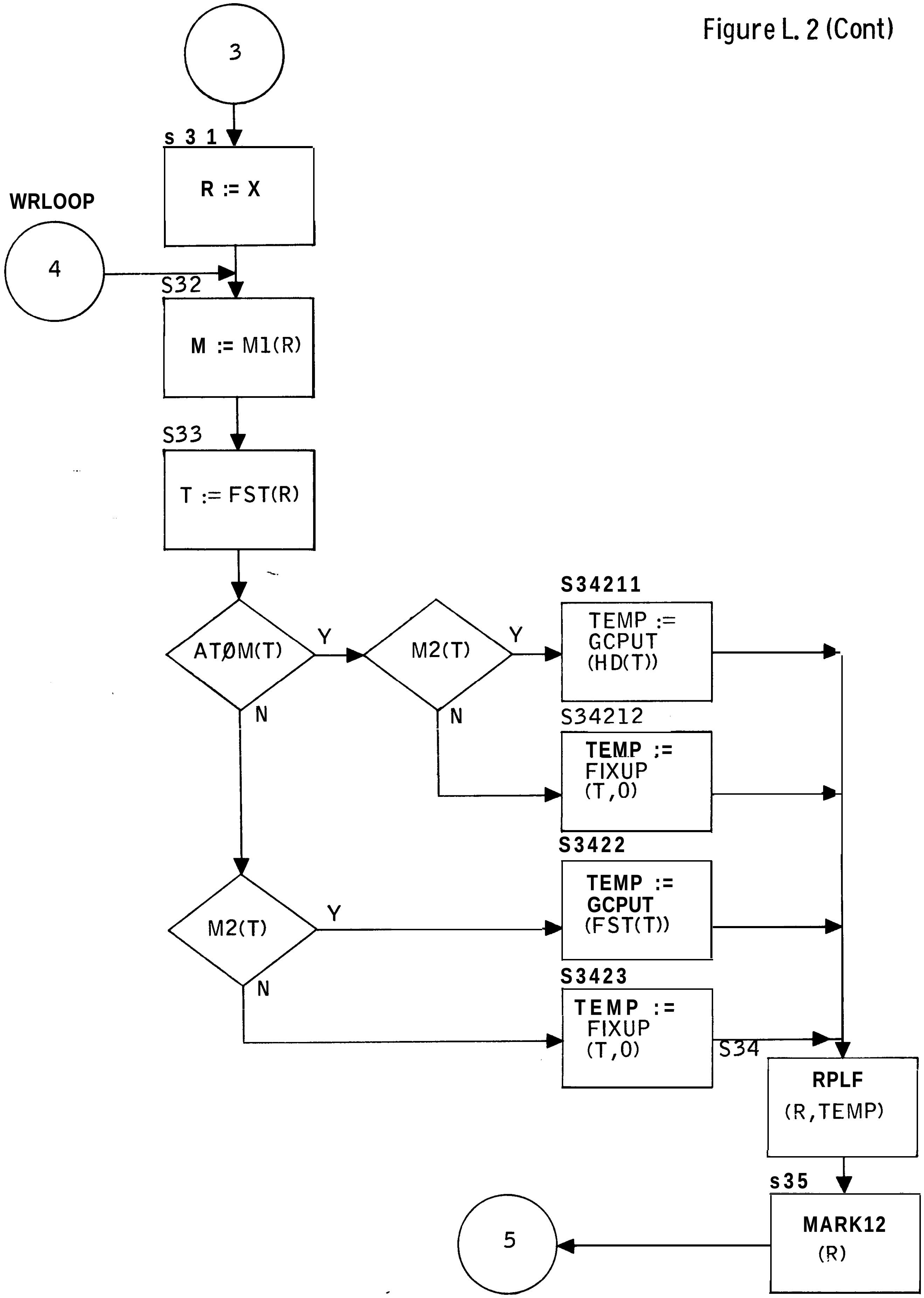


Figure L. 2 (Cont)

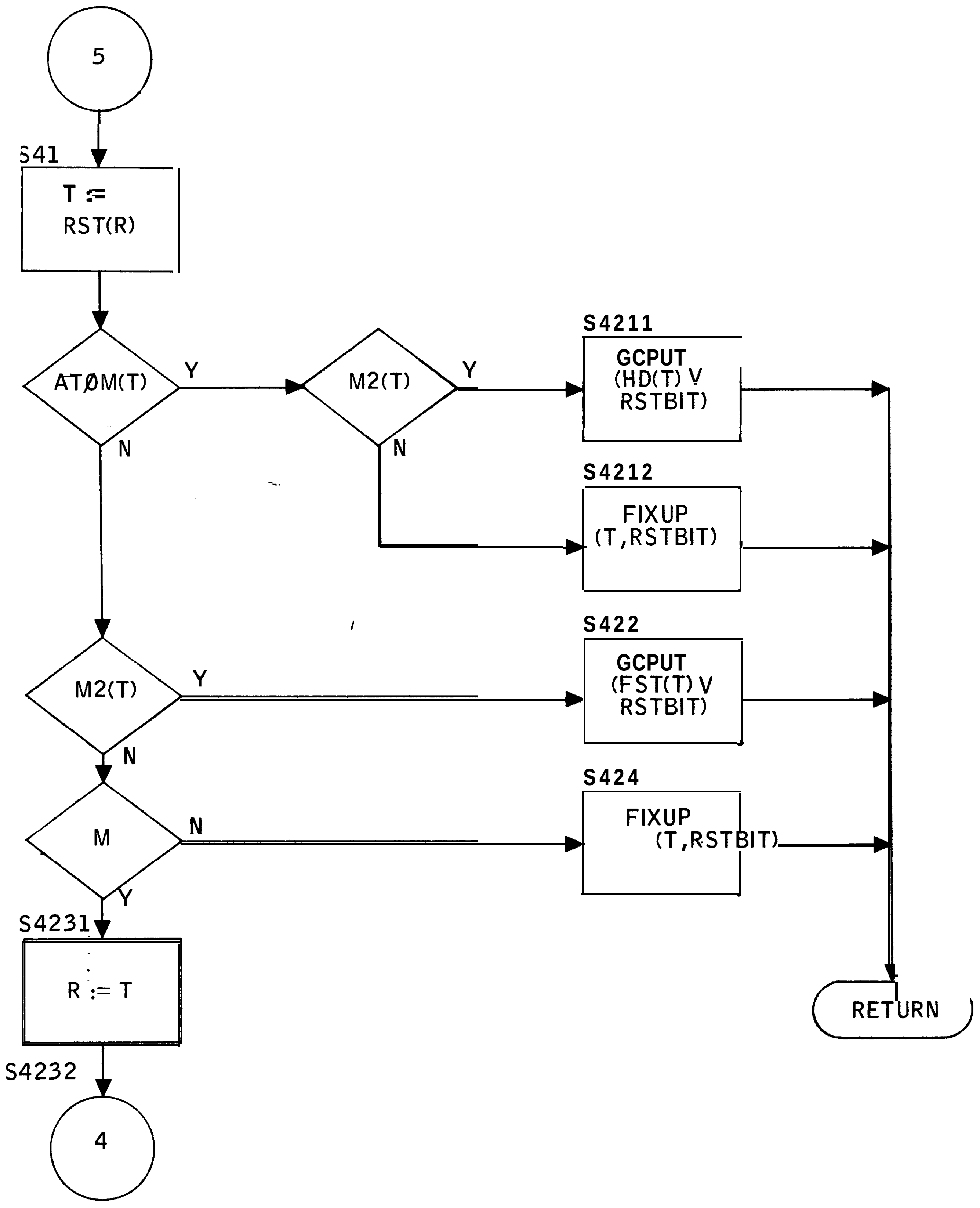
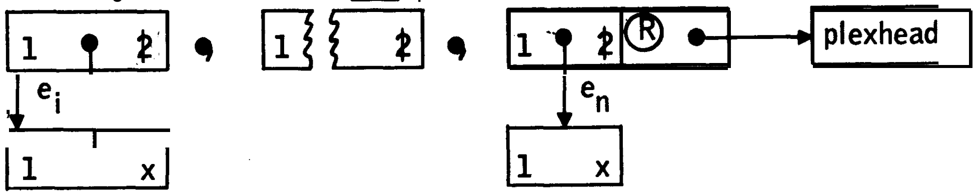
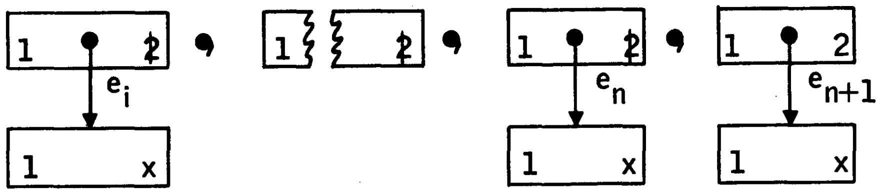


Figure L. 3
Cases of 'List Segment'

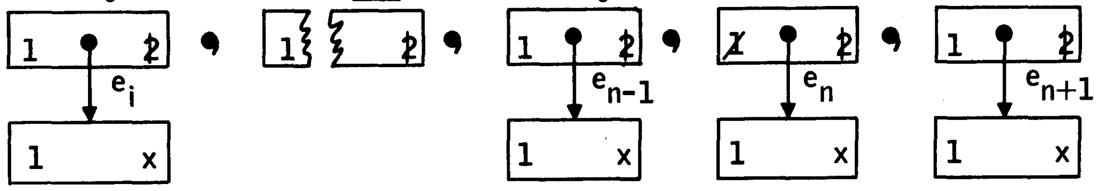
Case 1: List segment ends with rst pointer at atom



Case 2: List segment ends with rst that has already been collected



Case 3: List segment ends with rst that is being collected



Notation:		
	❟	indicates rst (either adjacent or rst pointer)
	e_i	is a pointer at an element of a list segment
	$i \geq 1$	
	1 (2)	indicates M1 (M2) set
	$\cancel{1}$ ($\cancel{2}$)	indicates M1 (M2) is zero
	x	indicates indeterminate M2

Figure L. 4

Properties of ATCØL

Assumption:

Al. The argument must be a pointer at an atom.

Properties:

A2. If the atomhead is already marked with Ml, then ATCOL returns; otherwise

A3. On entry, the atomhead is marked with Ml.

A4. On exit, the atomhead is replaced with a pointer to the equivalent atom in new core and the atomhead is marked with Ml-and M2.

A5. No word marked Ml before entry to ATCOL is modified; marked, or unmarked.

NOTE: ATCØL may call CØLLECT to collect a substructure of the atom. If that substructure points back to the atom, CØLLECT will find an atom that is Ml but not M2. This case is handled at S34212 and S4212.

Properties of GCPUT

Assumption:

G1. The argument may be any word, with or without the rst bit.

Properties:

G2. GCPUT stores its argument in the next location in the new core.

G3. The value is the assigned new core address.

Properties of FIXUP

Assumptions:

F1. First argument is a pointer at a word in old core.

F2. Second argument is either zero or zero with the rst bit.

Properties:

F3. The second argument is GCPUT.

F4. An entry is made in the fixup table consisting of the first argument and the value of GCPUT.

F5. After processing the fixup table, the GCPUT word will point to the equivalent of the first argument.

Processing the fixup table takes two steps:

(1) After CØLLECT, the first argument (to FIXUP) will be M1 and M2 by C10; it is replaced in the fixup table by its contents, which point to its new core equivalent (by lemma 14).

(2) After loading the new core, the word pointed at by the second item in each fixup is replaced by the first item.

Properties of CØLLECT

Assumption:

C0	$\neg M1(\underline{x}) \wedge \neg M2(\underline{x}) \wedge \neg \underline{atom}(\underline{x})$

Pass 1 isolates a list segment.

C1	After pass 1, each successive <u>fst</u> is marked with at least M1.
C2	The M1 bit for each word constituting the list segment is set on.
C3	Pass 1 terminates when it reaches a word that is an atom, is M2, or is M1.
C4	In the last case of C3, the M1 bit in the last word of the list segment is set off.

Pass 2 writes it out and remembers its location(s).

C5	Writes to new core one word for each word marked in C1.
C6	Places in each word marked in C1 the address of the new core equivalent word.
C7	Marks each word marked in C1 with M1 and M2.
C8	Writes to new core a <u>rst</u> pointer to the <u>rst</u> of the list segment.

Miscellaneous:

C9	CØLLECT does not modify any word marked with M1 by any other routine or by any other invocation of CØLLECT.
C10	Any word marked M1 either contains or will contain the address of the equivalent word in new core.

Instances of Case I with n=1

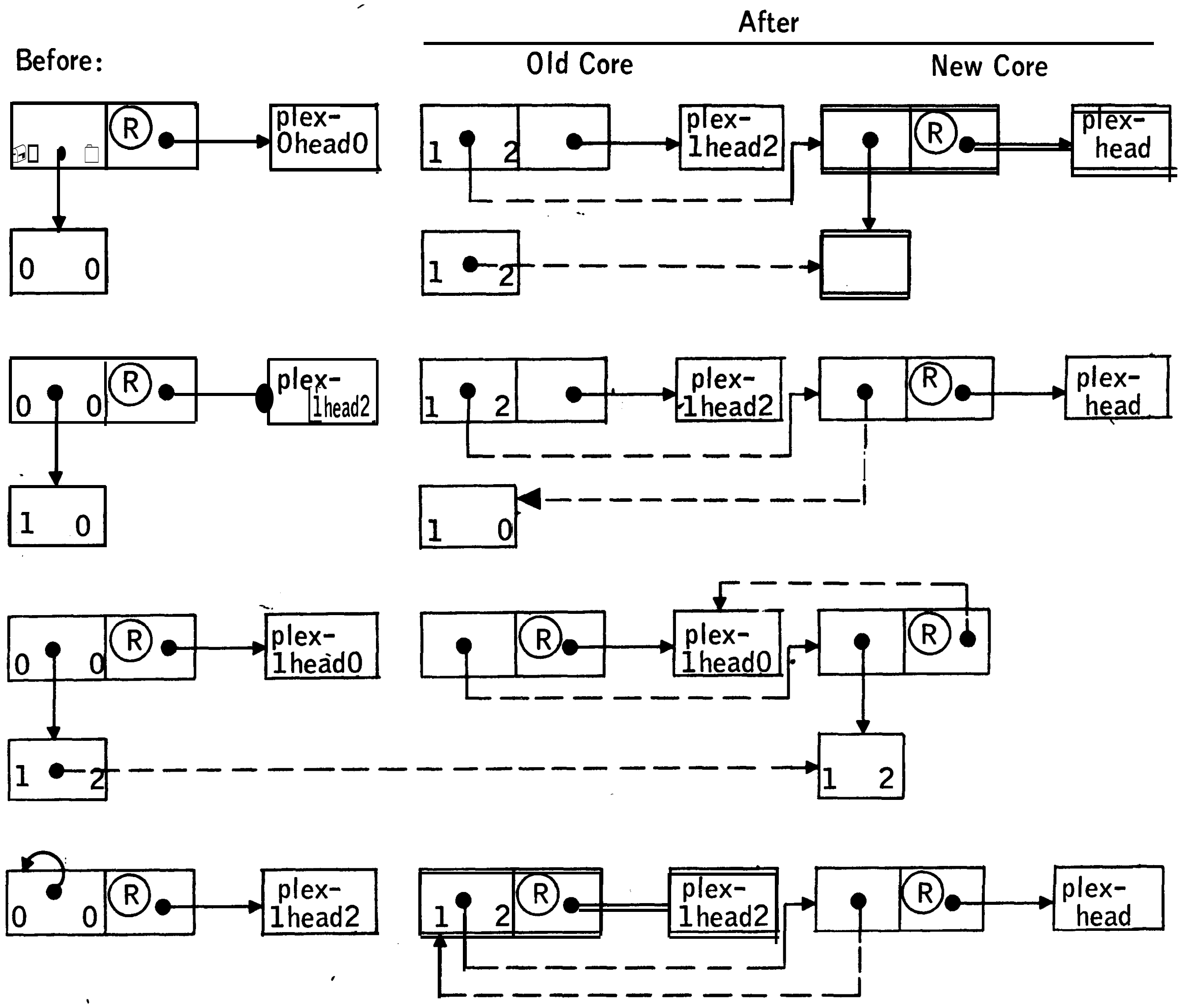


Figure L. 5
Collection of List Segments with n=1

Note:

A dashed line from old core to new core represents a pointer to the location a word will occupy when it is read in.

A dashed line from new core to old core represents an entry in the fixup table. The new core word will eventually point to the equivalent of the old core word.

Instances of Case I I with n=1

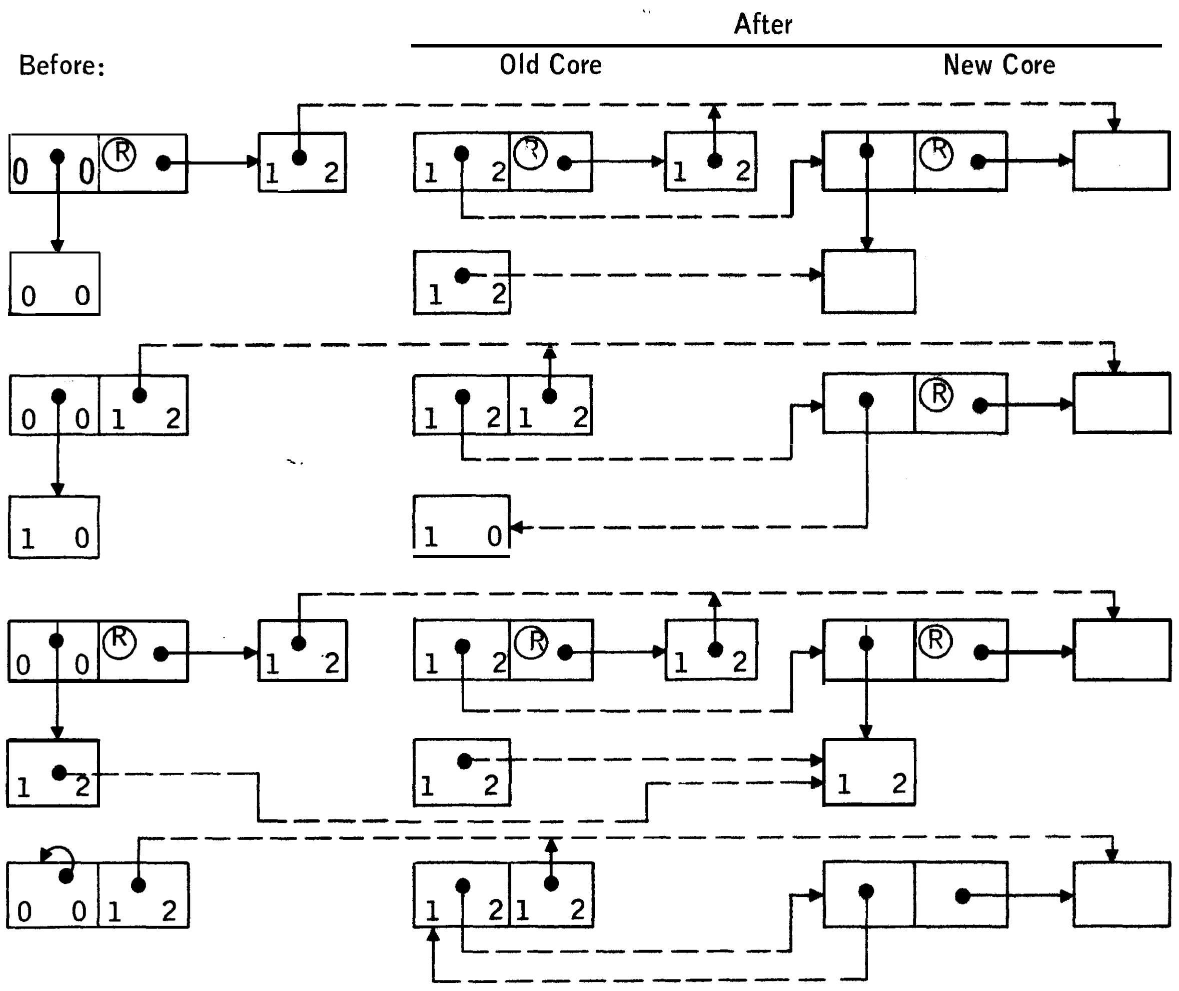


Figure L. 5
Collection of List Segments with n=1 (Cont)

Instances of Case III with n=1

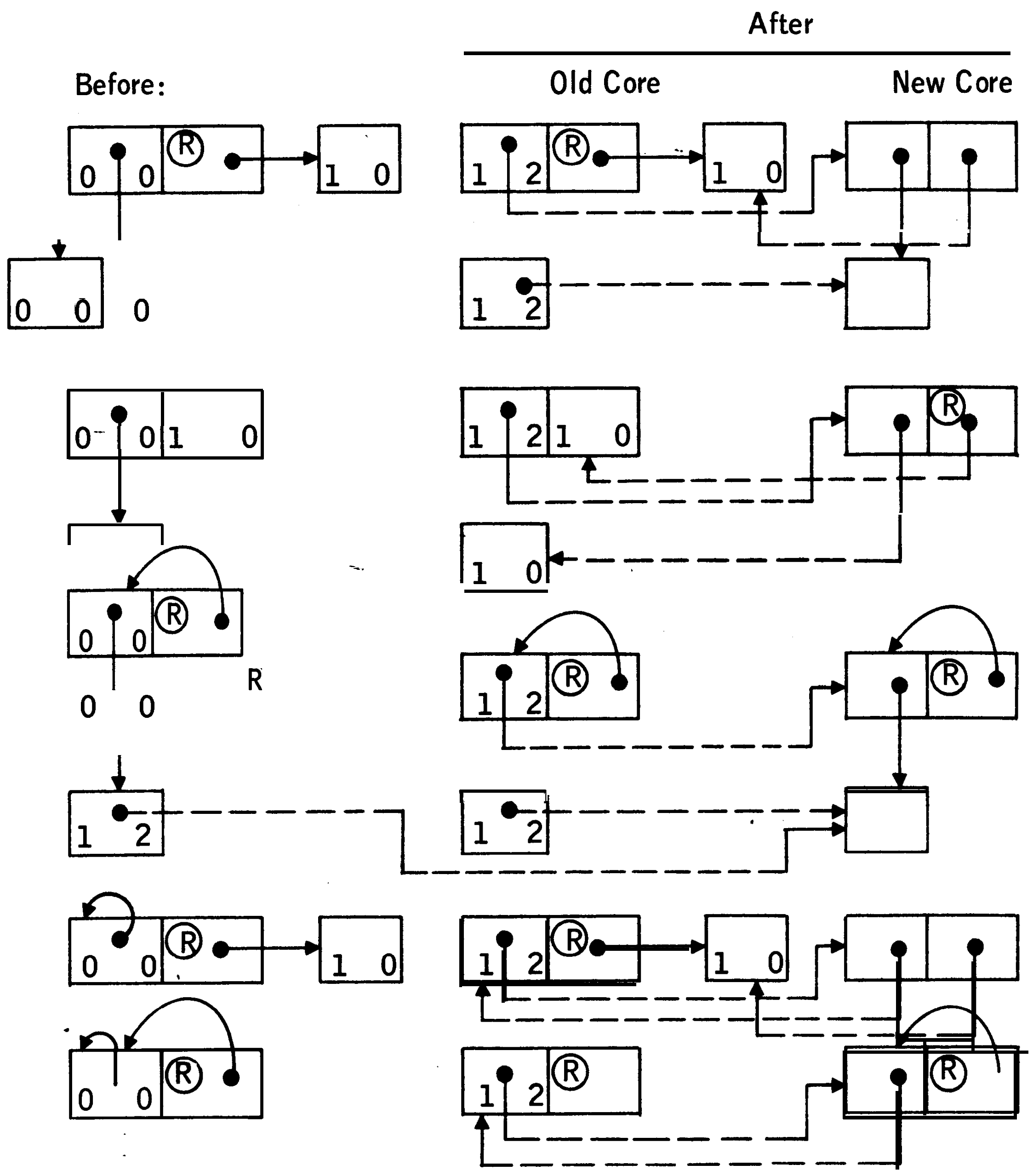


Figure L. 5

Collection of List Segments with n=1 (Cont)

Appendix M. Description of Control Section CSSWYM

The control section CSSWYM is always addressable via register S. It's contents serve a variety of needs: globalvariables for system routines, transfer vectors for routine linkage, register definitions. CSSWYM is non-reentrant. A DSECT describing its contents must be assembled with any Swym control section; the required code is described in Appendix N.

The following are included in CSSWYM:

1) Register Definitions. These names are equated to specific registers: N, Al, A2, A3,A4, A5,A6,C4, S, T, TT, F, P, B, IJ. See Appendix I.

2) AT EQU 6. Pointers at atoms point AT bytes in front of the atom. References to atoms should use this identifier to emphasize that the operand is an atom and in case the offset amount must be changed. (Many routines presently ignore this rule.)

3) Bit Definitions. The macro BITTBLMK is called to set up a table used by BIT (to find the bit mask for the bit-within-the-byte). Bits defined in CSSWYM are;

Ml, M2	The garbage collector marking bits. (These definitions should be moved to CSGC.)
CELREL	This bit is on in an atom head to indicate that the value cell contains a pointer at list structure. If off, the cell contains a number.
CELVAL	If on, the cell contains a value definition (possibly the special value UNDEFINED). If off, the cell contains a function definition.

CELFNC	This is a byte mask definition defining the function definition bits in the atom head. If any of these bits is on, the atom has a function definition.

4) SWYM EQU *

```
        USING SWYM,S
```

This establishes addressability for the information in CSSWYM. Note that no program may modify the contents of register S. (The contents are established by the routine CSINIT.)

5) Temporary Storage Areas.

SWYMSAVE	Used as save area when calling OS routines.
SYSFOO	Five word area to save registers 13, 14, 15, 0, 1 while calling OS.
DUBWORK	A double word work area.
TIME	Used by STIME and TTIME to compute processing time.
NUMAT, NUMATVAL	A number can be printed by storing it in NUMATVAL, then passing a pointer to NUMAT to PRINT or PRIN1.

6) Pointers at List Structure.

These pointers point at list structure referenced by the system. The values are updated by the garbage collector.

VCHAROBS	Points at CHAROBS, the list of all character objects; i.e., atoms with one character print names.
VOBLIST	Points at the OBLIST.
ST	Points at the atom T.
VFPROPS	Points at FPROPS for EVGET.

VUNBND   Points at the special atom 'UNBOUND' for EVAL.

For further information on these structures, see Appendix H.

7) Work Areas for Specific Routines

See the indicated appendix for further information on these variables:

. Memory control - Appendix E.4

MEMUSE, MEMNXT, MEMSIZ, FEND

Garbage Collector - Appendix E.4

GCTIME, GCABAD, #M1M2

Print - Appendix F.3

PRPT, PRPEND, PRLNG, PRATBAD

Read - Appendix C

RDCOL, RDEND, RDLNG, PBHD, ATAMT, RDSUPCTR, RDERMS, RDERNØ, RDERLØC, RDERCT, RDCLASS, RDCHAR, RDSTAT

8) Data control blocks.

There are two DCB's, one for output - PRINTER, and one for input - CARDRDR. In the copied code, these are not assembled, but space is reserved. They are assembled when CSSWYM is assembled by itself as a CSECT.

9) Transfer vectors.

These contain the address constants used to address routines by the CAL macro. The field labeled #xxx contains the address of the routine xxx. The transfer vectors are created with the TVMAK macro. One special transfer vector is included: #PO contains the address of the stack. This is-used by ERROR to restore the stack pointer (register P).

10) Always addressable routines.

See the indicated appendix for a description of these routines.

Appendix	Routine
G	FALSE, TRUE, PUTCH, SWERROR
E.3	CHOKE
B.1	RSTA1, RSTA2, RSTA3, RSTT, RSTTT

Appendix N. Adding Routines to SWYM-Stutter

Assembled routines, compiled routines, and interpreted routines can be added to the SWYM System with a minimum of difficulty. This appendix treats each of these types in turn.

N.1. Adding Assembled Routines

Routines designed to run under SWYM can be assembled in either an existing SWYM control section or a new control section. In either case, the assembly must include CSSWYM as a dummy control section so the routines can communicate with SWYM. The following code must begin any SWYM assembly:

```
         TITLE  'title of control section'
CSSWYM   DSECT
         PRINT OFF
        'COPY  SWYM
         PRINT ON
*        COPY  SWYM
csectnm CSECT
```

The code for CSSWYM is copied from the SWYM macro library. Each routine must obey the linkage conventions indicated in Appendix K. It must begin (physically and logically) with the SUB macro. It must end (logically) by executing the RET macro. If the routine is to be referenced by routines in other control sections, an entry must be made in the transfer vector table in CSSWYM. To avoid reassembling all control sections, the entry should be made at the end of the table and the card,

```
DS nnA(0) (currently nn = 20)
```

should have nn reduced by 1. In this way, the transfer vector table stays the same length. If the routine is not referenced by routines outside its control section, it is sufficient to include a TVMAK card for the routine at the end of the control section. The TVMAK card must be addressable when the routine itself is executed (register B points at the SUB macro).

If a routine is to be referenced from Stutter interpreted functions, there must be an atom for it in free storage. This atom can be created by coding either

```
        SUBR   new routine name
or      FSUBR new routine name.
```

Both generate an atom with the given indicator and a pointer at the new routine. The new routine name must be the same as the label on the SUB macro beginning the routine.

N.2. Compiling Functions for Swym

Although there is no STUTTER compiler, Swym has provision for including compilers. Three major problems must be faced: storage for the compiled code, linkage between routines, and variable binding.

There is no Swym binary program space. The plan is that compilers will store code in a new plex type. This 'code plex' will have a section for reentrant address-independent code, a section for relocatable pointers, and possibly a section for non-reentrant, address-independent data. The garbage collection routine for this plex type should move these plexes to a semi-permanent area to avoid relocating them every time the garbage collector is called.

The address of a routine may appear in two different places - the transfer vector table and the property list of the name of the routine (under either the SUBR or FSUBR indicator). To call another code routine, a compiled routine must load its address from the transfer vector table using code such as is generated by the CAL macro. The compiler can find the appropriate transfer vector entry because the contents are the same as the address stored on the property list of the called routine's name. The compiler must also store the address of a compiled routine in both the transfer vector table and on the property list of the name of the routine. This address must be the address of the code. If the code is stored in a *code plex', the plexhead is presumably stored immediately in front of the code. A special bit in the plexheadof the name of the routine must tell the garbage collector that the value of the SUBR or FSUBR property addresses a code plex. If thatplex is relocated, the address of the code must be changed in both places where it is stored.

The interpreter passes arguments to SUBR's and FSUBR's in registers Al to A6. Compiled functions may not have more than six arguments and may expect them in those registers. The result must be returned in register Al. If a compiled routine needs more working space than A1-A6, T, and TT, then it must store information on top of the stack with the equivalent of PUSH and POP.

N.3. Defining Routines To Be Interpreted

A routine to be interpreted must be stored as an s-expression with the format given in Appendix D. This expression must be the value of the indicator EXPR or FEXPR stored on the property list of the name of the routine. The basic function PUTPROP may be used for storing such expressions:

```
(PUTPROP
        (QUOTE   routine name)
        (QUOTE   s-expression)
        (QUOTE   EXPR)
)
```

A DEFINE function can be defined to simplify the process. The version in figure N.1. accepts a list of function definitions of this form:

```
(name vlexp, ... expm)
```

where name is the atom where the rest of the expression is to be stored under the indicator EXPR.

```
< PUTPROP
       (QUOTE DEFINE)
       (QUOTE ((A) (DEF1 A)))
       (QUOTE FEXPR)
>
< PUTPROP
       (QUOTE DEF2)
       (QUOTE ((A) < PUTPROP
               (FST A)
               (RST A)
               (QUOTE EXPR) >))
       (QUOTE EXPR)
>
(DEF2 (QUOTE
       (DEF1 (A) < COND
               ((NULL A) NIL)
               (T (TAK2 (DEF2 (FST A)) (DEF1 (RST A))))
       >)
))
```

Figure N.1

Appendix O. SWYM Control Sections

The assembly of SWYM-Stutter is divided into ten control sections or CSECT's. When a routine in one CSECT is modified, it is only necessary to reassemble that CSECT. Thus, total assembly time is reduced. All other CSECT's use information in CSSWYM. For this reason, CSSWYM is assembled as a DSECT along with each other control section. The assembly code to do this is in Appendix N. This appendix lists the CSECTS and sketches the contents of each.

The only non-reentrant control sections are CSSWYM, CSPDL, and CSFREEST. There must be separate copies of these for each user of Swym. The other control sections may be shared by all jobs in the 360 memory.

CSINIT Contains inititlization code for running any programs (not just Stutter) under Swym. CSINIT establishes register contents, opens the card and print data sets, and starts the timer. Eventually, initialization will include reading PARM information and setting up the stack and free storage areas according to parameters. CSINIT is not needed after initialization.

CSSWYM Contains global information for Swym system routines. Complete details are in Appendix M.

CSSUBS Basic subroutines for the Swym data structure; such as: FST, RST, and TAK2.

CSGC Garbage collector. See Appendix E.

CSFREEST Free storage. See Appendix H. (CSSWYM is not assembled with CSFREEST.)

CSMAIN Main loop for Stutter. Calls READ, EVAL and PRINT in turn as described in Appendix D. CSMAIN also contains FINISH which is entered when the input is exhausted. By replacing CSMAIN, Swym can be used as the basis for other interpreters.

CSREAD Read routines. See Appendix C.

CSPRINT Print routines, See Appendix F.3.

CSEVAL Stutter interpreter and functions useful to interpreted functions. The routines in CSEVAL are among those described in Appendix F.

CS2250 Experimental routine to interface to the 2250. Currently, the only function is to ring the 2250's bell.

MNEMONIC INDEX

All major Swym mnemonics are listed in this index. With each mnemonic is listed its class and the location of its definitions in the Appendices and the program code. A brief comment describes the function of the mnemonic. Four differently sorted indices are included: mnemonic, class, appendix, and control section. The last three are primarily for review purposes.

There are five columns:

1) MNEMONIC - The indexed mnemonic.

2) CLASS - The ten classes are:

a) MACRO	Swym macro
b) SUBR	routines available to Stutter programs. These
c) FSUBR	routines may also be entered with CAL.
d) CAL	routine callable only from assembled programs
e) CSECT	control section
f) REG	name equated to a register
g) SWYM	name defined in CSSWYM
h) FIELD	name equated to a bit or field definition
i) STRUCT	a structure in initial free storage
j) MISC	miscellaneous. Mostly routines with non-standard calling sequences.

3) APP - Appendix containing definition of mnemonic.

4) CSECT - Control section in which the mnemonic is defined.

5) COMMENTS - A brief description of the mnemonic.

MNEMONIC	CLASS	APP	CSECT	COMMENTS
AND	MACRO	6.7	MACLIB	COMBINE TWO PREDS
AT	MISC	M	CSSUYM	EQUATED TO ATOM OFFSET(6)
ATAMT	SWYM	C	CSSWYM	ATOM OFFSET (6)
ATCOL	CAL	E.3	CSGC	COLLECTS AN ATOM
ATCO	MISC	E.3	CSGC	PART OF ATCOL FOR TYPE 3 ATOMS
ATC1	MISC	E.3	C SGC	PART OF ATCOL FOR TYPE 1 ATOMS
ATOM	MACRO	8.1	MACL IB	? IS ARG AN ATOM
ATOM	SUBR	F.1	C SSUBS	STUTTER ROUTINE FOR-IS ARG ATOM?
A I	REG	I	CSSWYM	ARGUMENT REGISTER & RESULT REGISTER
A2	REG	I	CSSWYM	ARGUMENT REGISTER
A3	REG	I	CSSWYM	ARGUMENT REGISTER
A4	REG	I	CSSUYM	ARGUMENT REGISTER
A5	REG	I	CSSUYM	ARGUMENT REGISTER
A6	REG	I	CSSWYM	ARGUMENT REGISTER
B	REG	I	CSSUYM	BASE REG FOR ALL ROUTNS
BCMAC	MACRO	B.7	MACL IB	MAKE A BR CONDITION INSTRUCTION
BELL	SUBR	F.5	CS2250	RINGS BELL ON 2250
BINDERY	CAL	0.3	CSEVAL	BIND ARG ATOMS TO THEIR VALUES
BIT	MACRO	B.5	MACL IB	IDENTIFY MNEMONIC WITH BIT IN WORD
BITTBLMK	MACRO	B.5	MACL IB	MAKE A TABLE FOR 'BIT'MACRO
CAL	MACRO	B.6	MACLIB	SUBROUTINE CALL
C ARDRDR	SWYM	M	CSSWYM	DCB FOR READING CARDS
CELFNC	FIELD	M	C SSUYM	ATOM HEAD-FUNC DEF TYPE BITS
CELL	MACRO	B.2	MACLIB	LOADS ATOM CELL INTO REG
CELREL	FIELD	M	CSSUYM	ATOM HEAD-CELL IS RELOCATABLE
C ELVAL	FIELD	M	C SSUYM	ATOM HEAD-CELL HAS VALUE (NOT FNC)
CHAR	MACRO	8.3	YACLIB	CREATES A CHAR OBJECT ATOM
CHAROBS	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL CHARS
CHOKE	MISC	E.3	c SGC	BRANCH TO IF STORE EXHAUSTED,ABEND
CHTBL	MACRO	8.8	MACLIR	MAKE A CHARACTER TABLE (FOR TR)
COLLECT	CAL	E.3	CSGC	CREATES IMAGE OF ARG IN NEW CORE
COLX	CAL	E.3	C SGC	CHECKS AND COLLECTS ONE POINTER
COND	FSUBR	F.4	CSEVAL	CONDITIONAL EXPRESSION EVALUATED
C SEVAL	CSECT	D	CSEVAL	INTERPRETER AN0 RELATED ROUTINES
CSFREEST	CSECT	H	CSFREEST	FREE STORAGE,INCL INITIAL STRUCTS
CSGC	CSECT	E	C SGC	GARBAGE COLLECTOR
CSINIT	CSECT	0	CSINIT	INITIALIZATION
c SSWYM	CSECT	M	C SSUYM	GLOBAL INFORMATION FOK SWYM RTNS
CSMA IN	C SECT	0	CSYAIN	MAIN STUTTER LOOP
CSPDL	c SECT	0	CSPDL	STACK
CSPRINT	CSECT	0	CSPRINT	PRINT ROUTINES
CSREAD	CSECT	C	C SREAD	READ ROUT I NES
CSSUBS	CSECT	0	c SSUBS	BASIC SUBROUTINES
CS2250	CSECT	0	CS2250	2250 EXPERIMANTAL INTERFACE
C4	REG	I	CSSWYM	ODD REGISTER CONTAINING F'4'
DUBWORK	SWYM	M	CSSWYM	DOUBLE WORD WORK AREA
EJECT	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT PAGE
ELSE	MACRO	B. 7	HACLIB	COND - END TRUE; START FALSE PART
END IF	MACRO	B.7	MACLIB	COND - END FALSE; END CONDITIONAL
EQ	MACRO	8.1	MACL IB	? ARG1 = ARG2(TESTS TWO POINTERS)
EQ	SUBR	F.1	CSSUBS	STUTTER RTN FOR-ARGL = ARG2?

MNEMONIC	CLASS	APP	CSECT	COMMENTS
ERROR	SUBR	F.5	C SSUBS	WRITES MESSAGE AND GOES TO TOP LVL
EVAL	SUBR	0.3	CSEVAL	STUTTER INTRPRTR EXPRSN EVALUATOR
EVCH	MACRO	8.3	MACLIB	GETS 4RITH VAL OF EBCDIC BITS
EVGET	CAL	0.3	C SEVAL	GET FUNCTION DEFINITION OF ATOM
EVLIS	CAL	0.3	C SE VAL	EVALUATE LIST OF EXPRESS IONS
EXPLOOE	SUBR	F.3	CSEVAL	CONVERTS ATOM TO LIST CHARS IN PNAM
EXPR	STRUC	0.2	C SFREEST	INDICATOR FOR S-EXPR FUNCTIONS
F	PEG	I	CSSWYM	FREE STORAGE POINTER
FALSE	MISC	G	CSSWYM	L A1,NIL; RET; (BRANCH TO IT)
FEND	SWYM	E.4	CSSWYM	POINTS AT END OF FREE SOTR
FEXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXP SPECIAL FNCTS
FINDBIT	MACRO	B.5	MACLIB	FIND BIT MNEMONIC FOR BYTE-IN-WORD
FINISH	MISC	G	CSYAIN	CLOSE FILES AND EXIT
FIXUP	MACRO	B.8	MACL IB	GC-MAKE ENTRY IN FIXUP TABLE
FPROPS	S TRUC	H	C SFREEST	STRUCTURE: ((SUBR.1) (FSUBR ....
FST	MACRO	B.I	MACLIB	FIRST ELEMENT OF LIST
FST	SUBR	F.I	C SSUBS	STUTTER RTN FOR -1ST ELEM OF LIST
FSUBR	MACRO	8.3	MACLIB	CREATES AN ATOM WITH FSLJBR PROP
FSUBR	STRUC	0.2	CSFREEST	INOICATOR FOR ASSEMBLED SPECIAL FNC
GC	SUBR	E.3	C SGC	CONTROLS GARBAGE COLLECT ION
GCABAD	SWYM	E.4	CSSWYM	GC ABENDS FOR BAD DATA STRUCTURE
GC ABEND	MISC	E.3	C SGC	BAL TO IF DATA STRUCTURE ERR, ABEND
GCPUT	YACRO	B.8	YACL I B	GC-PUT WORD TO NEW CORE
GCPUT	MISC	E.3	C SGC	BAL'ED TO BY GCPUT MACRO
GCTIME	SWYM	E.4	C SSWYM	GC COMPUTES ITS TIME
GET	SUBR	F.4	C SEVAL	FINDS PROPERTY OF AN ATOM
GETCH	CAL	C	CSREAD	GET A CHARACTER
GETNAME	MACRO	8.2	MACL IB	LOAOS PTR AT PNAME CHR STR ATM
GETNUM	MACRO	8.2	MACL IB	GET VALUE OF NUM CHAR STR ATOM
GETOBJ	SUBR	F.2	CSREAD	FINDS SYMBOL FOR CHAR STRING ARG
GOTO	MACRO	8.7	YACLIB	BRANCH
HASH	MACRO	8.3	MACLIB	HASH CODE AN IDENT FOR OBLIST
HEAO	MACRO	8.2	MACLIB	LOADS HEAD OF ATOM
IF	MACRO	8.7	MACLIB	COND - START PREDICATE
INIT	MISC	G	CSINIT	SET UP SWYM REGS AND OPEN FILES
INST4	MACRO	8.8	YACLIB	ASSEMBLE INSTRUCTION WO/ ALIGN ERR
INVERTB	MACRO	B.5	MACLIB	CHANGE BIT
IVCCH	SUBR	F.2	C SRFAO	RETURNS NEXT INPUT CHAR
IVQMO	SUBR	F.2	CSREAO	RETURNS STATUS OF QUOTE MODE
L	REG	I	CSSWYM	LINKAGE REG /RETURN ADDRESS)
LIST	F SUBR	F.I	CSEVAL	MAKES A LIST Of THE ARG EXPRESSIONS
MAIN ·	MISC	D.I	CSMAIN	MAIN LOOP OF STUTTER INTERPRETER
MAKSTRNG	SUBR	F.2	CSREAD	MAKES CHR STR ATM FROM LIST OF CHRS
MATCM	MACRO	8.3	MACLIB	CREATES AN ATOM STRUC (IN CSFREEST)
MEMNXT	SWYM	E.4	C SSWYM	ALTERNATE FREE STOR
MEMSIZ	SWYM	E.4	CSSWYM	SIZE OF FREE STORAGE
MFMUSE	SWYM	E.4	CSSWYM	FREE STOR IN USE
Ml	FIELO	E.2	C SSWYM	GARB COL MARKING BIT
M2	FIELD	E.2	CSSWYM	GARB COL MARKING BIT

MNEHONIC	CLASS	APP	C SECT	COMMENTS
N	REG	I	CSSUYM	POINTS AT NIL
NIL	STRUC	H	CSFREEST,	ATOM WITH VALUE-NIL
NLENGTH	CAL	G	C SEVAL	GET LENGTH OF LIST
NOT	MACRO	0.7	MACLIB	NEGATE PREDICATE MACRO TEST
NULL	MACRO	B.1	MACLIB	? ARG = NIL
NULL	SUBR	F.I	CSSUBS	STUTTER RTN FUR - IS ARG = NIL?
NUMAT	SWYM	M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
NUMATVAL	SWYM	M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
OBLIST	STRUC	H	C SFREEST	ATOM WITH VALUE - LIST OF ALL ATOMS
ORX	MACRO	B.7	MACLIB	COMBINE TWO PREDS
P	REG	I	CSSWYM	STACK POINTER
PBCLOSE	CAL	C	CSREAD	FINISH CHAR STRING ATOM
PBHD	SWYM	C	CSSWYM	HOLDS ADRS OF At-HD DURING PUTBYTE
PBOPEN	CAL	C	CSREAD	START MAKING CHAR STRING ATOM
POP	MACRO	8.4	MACLIB	GETS TOP OFF STACK-REDUCES STACK
POPN	MACRO	B.4	MACLIB	REDUCES STACK N TIMES
PRATBAD	SWYM	F.3	CSSWYM	AREA FOR PRINGING '?TYPN'
PRINT	SUBR	F.3	CSPRINT	PRINTS ITS ARG AND GOES TO NFXT LIN
PRINTER	SWYM	M	CSSWYM	DCB FOR PRINTING
PRIN1	SUBR	F.3	CSPRINT	PRINTS ITS ARG
PRLNG	SWYM	F.3	CSSWYM	LENGTH OF PRINT LINE
PRPEND	SWYM	f.3	CSSWYM	WHERE TO PUT LAST PRINT CHAR
PRPT	SWYM	F.3	CSSWYM	WHERE TO PUT NXT PRINT CHAR
PUSH	MACRO	0.4	MACLIB	PUTS ARG ATOP STACK
PUTBYTE	CAL	C	CSREAD	PUT RYTE INTO CHAR STRING
PUTCH	MISC	G	CSSWYM	PUT CHARACTER IN PRINT LINE
PUTPROP	SUBR	F.4	CSEVAL	STORES PROPERTIES UN ATOMS PROP LST
PUTSTR	CAL	G	CSPRINT	PRINT A CHARACTER STRING ATOM
QCHAR	MACRO	8.3	MACLIB	CREATES A CHAR OBJ FOR '('')'',''
QUOTE	FSUBR	F.4	CSEVAL	RETURNS ITS ARG UNEVALUATED
RDAT	CAL	C	C SREAD	READ AN ATOM
RDCHAR	SWYM	C	CSSWYM	LAST CHAR READ
RDCLASS	SWYM	C	CSSWYM	CLASS OF LAST CHARACTER READ
RDCOL	SWYM	C	CSSWYM	LOC OF LAST WORD READ
RDEND	SWYM	C	CSSWYM	LOC OF LAST CHAR TO READ
RDERCNT	SWYH	C	CSSWYM	PRINT #PARENS CREATED BEFORE '>'
RDERLOC	SWYM	C	CSSWYM	SYNTAX ERROR CARD COLUMN INDICATION
RDERMS	SWYM	C	CSSWYM	READ SYNTAX ERROR MESSAGE AREA
RDERNO	SWYM	C	CSSWYM	SYNTAX ERROR NUMBER
RDERR	CAL	C	CSREAD	INDICATE INPUT SYNTAX ERROR
RDERRCNT	CAL	C	CSREAD	SYNTAX ERR-PARENS MADE BEFORE '>'
RDLIST	CAL	C	CSREAD	READ A LIST
RDLNG	SWYM	C	CSSWYM	NUMBER OF CHAR READ FROM EACH CARD
RDSE	CAL	C	CSREAD	READ AN S-EXPRESSION
RDSTAT	SWYM	C	CSSWYM	READ ROUTINES STATUS INFO BYTE
RD SUPCTR	SWYM	C	CSSWYM	COUNT #PARENS CREATED BEFORE '>'
READ	SUBR	F.2	C SREAD	READS ONE EXPRESSION FROM CARD
READCH	SUBR	f.2	CSREAD	READS ONE CHARACTER FROM CARD
REMPROP	SUBR	F.4	C SEVAL	REMOVES PROPERTIES FROM P-LIST
RESETB	MACRO	8.5	MACLIB	TURN OFF BIT

MNEMONIC	CLASS	APP	C SECT	COMMENTS
RET	MACRO	8.6	MACLIB	SUBROUTINE RETURN
RPLCEL	MACRO	8.2	MACLIB	REPLACES ATOM CELL
RPLF	MACRO	8.1	MACLIB	REPLACES FIRST PTR OF LIST
RPLHD	MACRO	B.2	MACLIB	REPLACES HEAD OF ATOM
RPLTOP	MACRO	a.4	MACLIB	REPLACE TOP ITEM ON STACK
RPLTOPN	MACRO	0.4	MACLIB	REPLACE NTH ITEM OF STACK
RST	MACRO	8.1	MACLIB	ALL BUT 1ST ELEMENT OF LIST
RST	SUBR	F.1	C SSUBS	STUTTER RTN FOR - REST OF LIST
RSTA1	MISC	a.1	C SSWYM	RST(A1). BAL'EDTO BY RST MACRO
RSTA2	HISC	a.1	CSSWYM	RST(A2). BAL'EDTO BY RST MACRO
R STA3	MISC	8.1	CSSWYM	RST(A3). BAL'EDTO BY RST MACRO
RSTMAK	MACRO	a.1	MACLIB	MAKE ROUTINES FOR 'RST' TO BAL TO
RSTT	MISC	a.1	CSSYYM	RST(T). BAL'EDTO BY HST MACRO
RSTTT	MISC	8.1	CSSUYM	RST(TT). BAL'EDTO BY RST MACRO
S	REG	I	CSSWYM	BASE REG FOR CSSWYM
SASSOC	SUBR	F.4	CSEVAL	FINDS ARC ON AN ASSOCIATION LIST
SETBIT	MACRO	B.5	MACLIB	TURN ON BIT
ST	SWYM	M	CSSUYM	POINTER AT T
STAKN	CAL	G	cssuas	GET FREE STORAGE BLOCK
STIME	CAL	G	cssuas	START TIMER
STIVCCH	SUBR	F.2	CSREAD	SETS CURRENT INPUT CHAR
STIVQMO	SUBR	F.2	CSREAD	SETS QUOTE MODE
STRAT	MACRO	8.3	MACLIB	CREATES STRING ATOM STRUC (FREEST)
SUB	MACRO	B.6	MACLIB	SUBROUTINE ENTRY
SUBR	MACRO	B.3	MACLIB	CREATES AN ATOM WITH SUBR PROPERTY
SUBR	STRUC	0.2	CSFREEST	INDICATOR FOR ASSEMBLED FUNCTIONS
SWEAR	MACRO	B.8	HACL I B	SYSTEM ERROR
SUERROR	MISC	G	CSSWYM	SYSTEM ERROR
SWYM	SUYM	M	CSSWYM	FIRST LOC IN CSSUYM
SWYMSAVE	SWYM	M	CSSWYM	SAVE AREA FOR CALLING OS
SYSFOO	SWYM	M	CSSWYM	SAVE AREA FOR SAVING OS LIMK REGS
T	STRUC	H	CSFREEST	ATOM WITH VALUE-T
T	REG	I	CSSWYM	TEMP (EVEN, NEXT TO TT)
TAIL	MACRO	a.2	MACLIB	LOADS PTR AT TAIL OF ATOM
TAK2	SUBR	F.1	CSSUBS	MAKES LIST W/FST ARG1 AND RST ARGZ
TERPRI	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT LINE
TEST6	MACRO	8.5	MACLIB	TEST BIT
THEN	MACRO	B.7	MACL I a	COND - END PRED: START TRUE PART
TIME	SWYM	M	CSSWYM	TIME SET'AT LAST STIME
TOP	MACRO	0.4	MACL I B	GETS TOP OF STACK-BUT LEAVES IT
TOPN	MACRO	a.4	MACLIB	GETS NTH ITEM ON STACK
TRUE	MISC	G	CSSWYM	L A1,T; RET; (BRANCH TO IT)
TT	REG	I	C SSWYM	TEMP (ODD, NEXT TO T)
TTIME	CAL	G	c SSUBS	HOW LONG SINCE LAST STIME
TVEND	SWYM	M	CSSWYM	LABEL OF LAST ENTRY IN TV TABLE
TVMAK	MACRO	6.6	MACLIB	MAKE A TRANSFER VECTOR FOR CAL
TVSTART	SWYM	M	CSSWYM	LABEL OF START OF TRANS VECT TABLE
UNBIND	CAL	0.3	CSEVAL	RESTORE OLD BINDINGS OF ARG ATOMS
UNBOUND	STRUC	H	C SFREEST	RECOGNIZED BY EVAL AS ERROR VALUE
VALUE	MACRO	a.3	MACLIB	CREATES AN ATOM WITH A VALUE
VCHAROBS	SWYM	M	CSSWYM	POINTER AT CHAR OBJECTS LIST

MNEMONIC	CLASS	APP	CSECT	COMYENTS
VFPROPS	SUYM	M	CSSWYM	POINTER AI FPROPS STRUCTURE
VOBLIST	SWYM	M	CSSWYM	POINTER AT ALL OBJECTS LIST
VUNBND	SWYM	M	CSSWYM	POINTER-AT SPECIAL 'UNBOUND'
XB	MACRO	B.6	MACLIB	TRANSFER INTO MIDDLE OF SUBROUTINE
#M1M2	SUYM	E.4	CSSWYM	USED BY GC TO 'OR' IN M1 & M2 BITS
#PO	SUYM	M	CSSWYM	ADRS OF BEGINNING OF STACK
#XXXX	SWYM	M	CSSWYM	TRANSFEP VECTOR, ADKS OF RTN XXXX

MNEMONIC	CLASS	APP	CSECT	COMMENTS
ATCOL	CAL	E.3	CSGC	COLLECTS AN ATOM
BINDERY	CAL	0.3	CSEVAL	BIND ARG ATOMS TU THEIR VALUES
COLLECT	CAL	E.3	CSGC	CREATES IMAGE OF ARG I N N E W CORE
COLX	CAL	E.3	CSGC	CHECKS AND COLLECTS ONE POINTER
EVGET	CAL	0.3	CSEVAL	GET FUNCTION DEFINITION OF ATOM
EVLIS	CAL	0.3	CSEVAL	EVALUATE LIST OF EXPRESS IONS
GETCH	CAL	C	CSREAD	GET A CHARACTER
NLENGTH	CAL	G	CSEVAL	GET LENGTH OF LIST
PBCLOSE	CAL	C	CSREAD	FINISH CHAR STRING ATOM
PBOPEN	CAL	C	CSREAD	START MAKING CHAR STRING ATOM
PUTBYTE	CAL	C	CSREAD	PUT BYTE INTO CHAR STRING
PUTSTR	CAL	G	CSPRINT	PRINT A CHARACTER STRING ATOM
RDAT	CAL	C	C SREAD	READ AN ATOM
RDERR	CAL	C	C SREAD	INDICATE INPUT SYNTAX ERROR
RDERRCNT	CAL	C	CSREAD	SYNTAX ERR-PARENS MADE BEFORE '>'
RDLIST	CAL	C	CSREAD	READ A LIST
ROSE	CAL	C	CSREAD	READ AN S-EXPRESSION
STAKN	CAL	G	C ssuas	GET FREE STORAGE BLOCK
STIME	CAL	G	C ssuas	START TIMER
TTIME	CAL	G	C SSUBS	HOW LONG SINCE LAST STIME
UNBIND	CAL	D.3	CSEVAL	RESTORE OLD BINDINGS OF ARG ATOMS
C SEVAL	CSECT	D	CSEVAL	INTERPRETER AND RELATED ROUTINES
C SFREEST	CSECT	H	CSFREEST	FREE STORAGE, INCL INITIAL STRUCTS
CSGC	CSECT	E	C SGC	GARBAGE COLLECTOK
CSINIT	CSECT	0	CSINIT	INITIALIZATION
CSMA IN	CSECT	0	CSMAIN	MAIN STUTTER LOOP
C SPDL	CSECT	0	CSPDL	STACK
CSPR INT	CSECT	0	CSPRINT	PRINT ROUTINES
CSREAD	CSECT	C	CSREAD	READ ROUTINES
c SSUBS	CSECT	CI	CSSUBS	BASIC SUBROUTINES
CSSWYM	CSECT	M	CSSUYM	GLOBAL INFORMATION FOR SUYM RTNS
CS2250	CSECT	0	CS2250	2250 EXPERIMANTAL INTERFACE
CELFNC	FIELD	M	CSSWYM	ATOM HEAD-FUNC DEF TYPE BITS
CELREL	FIELD	M	CSSWYM	ATOM HEAD-CELL IS RELOCATABLE
CELVAL	FIELD	M	CSSWYM	ATOM HEAD-CELL HAS VALUE(NOTFNC)
MI	FIELD	E.2	CSSWYM	GARB COL MARKING BIT
M2	FIELD	E.2	CSSUYM	GARB COL MARKING BIT
COND	FSUBR	F.4	CSEVAL	CONDITIONAL EXPRESSION EVALUATED
LIST	F SUBR	f.I	CSEVAL	MAKES A LIST OF THE ARC EXPRESSIONS
QUOTE	FSUBR	F.4	C SEVAL	RETURNS ITS ARG UNEVALUATED
AND	MACRO	B.7	MACLIB	COMBINE TWO PREDS
ATOM	MACRO	a.1	MACLIB	? IS ARG AN ATOM
BCMAC	MACRO	a.7	MACLIB	MAKE A BR CONDITION INSTRUCTION
BIT	MACRO	B.5	MACLIB	IDENTIFY MNEMONIC WITH BIT TN WORD
BITTBLMK	MACRO	5.5	MACL IB	MAKE A TABLE FOR 'BIT'MACRO
CAL	MACRO	8.6	MACLIB	SUBROUTINE CALL
CELL	MACRO	8.2	MACLIB	LOADS ATOM CELL INTO REG
CHAR	MACRO	6.3	MACLIB	CREATES 4 CHAR OBJECT ATOM
CHTBL	MACRO	5.8	MACLIB	MAKE A CHARACTER TABLE (FORTR)
ELSE	MACRO	8.7	MACLIB	COND - END TRUE; START FALSE PART
END IF	MACRO	8.7	MACLIB	COND - END FALSE; END CONDITIONAL

MNEMONIC	CLASS	APP	CSECT	COMMENTS
EQ	MACRO	5.1	MACLIB	? ARG1 = ARG2(TESTS TWO POINTERS)
EVCH	MACRO	5.3	HACLIB	GETS ARITH VAL OF EBCDIC BITS
FINDBIT	MACRO	B.5	MACLIB	FIND BIT MNEMONIC FOR BYTE-IN-WORD
FIXUP	MACRO	8.8	MACLIB	GC-MAKE ENTRY IN FIXUP TABLE
FST	MACRO	8.1	MACLIB	FIRST ELEMENT OF LIST
FSUBR	MACRO	0.3	MACLIB	CREATES AN ATOM WITH FSUBR PROP
GCPUT	MACRO	8.8	MACLIB	GC-PUT WORD TO NEW CORE
GETNAME	MACRO	8.2	MACLIB	LOADS PTR AT PNAME CHR STR ATM
GETNUM	MACRO	8.2	MACLIB	GET VALUE OF NUM CHAR STR ATOM
GOTO	MACRO	8.7	MACLIB	BRANCH
HASH	MACRO	8.3	MACLIB	HASH CODE AN IDENT FOR OBLIST
HEAD	MACRO	8.2	MACLIB	LOADS HEAD OF ATOM
IF	MACRO	B.7	MACLIB	COND - START PREDICATE
INST4	MACRO	5.8	MACLIB	ASSEMBLE INSTRUCTION WO/ ALIGN ERR
.INVERTB	MACRO	8.5	MACLIB	CHANGE BIT
MATOM	MACRO	5.3	MACLIB	CREATES AN ATOM STRUC (INCSFREEST)
NOT	MACRO	8.7	MACLIB	NEGATE PREDICATE MACRO TEST
NULL	MACRO	8.1	MACLIR	? ARG = NIL
ORX	MACRO	8.7	MACLIB	COMBINE TWO PREDS
POP	MACRO	8.4	MACLIB	GETS TQP OFF STACK-REDUCES STACK
POPN	MACRO	-8.4	MACLIB	REDUCFS STACK N TIMES
PUSH	MACRO	8.4	MACLIB	PUTS ARG ATOP STACK
QCHAR	MACRO	8.3	MACLIB	CREATES A CHAR OBJ FOR '(' ')' ','
RESETB	MACRO	5.5	MACLIB	TURN OFF BIT
RET	MACRO	6.6	MACLIB	SUBROUTINE RETURN
RPLCEL	MACRO	8.2	MACLIB	REPLACES ATOM CELL
RPLF	MACRO	8.1	MACLIB	REPLACES FIRST PTR OF LIST
RPLHD	MACRO	B.2	MACLIB	REPLACES HEAD OF ATOM
RPLTOP	MACRO	8.4	MACLIB	REPLACE TOP ITEM ON STACK
RPLTOPN	MACRO	8.4	MACLIB	REPLACE NTH ITEM OF STACK
RST	MACRO	8.1	MACLIB	ALL BUT 1ST ELEMENT OF LIST
RSTMAK	MACRO	B.1	MACLIB	MAKE ROUTINES FUR 'RST' TO BAL TO
SETBIT	MACRO	0.5	MACLIB	TURN ON BIT
STRAT	MACRO	8.3	MACLIB	CREATES STRING ATOM STRUC (FREEST)
SUB	MACRO	8.6	MACLIB	SUBROUTINE ENTRY
SUBR	MACRO	8.3	MACLIB	CREATES AN ATOM WITH SUBR PROPERTY
SWEAR	MACRO	8.8	MACLIB	SYSTEM ERROR
TAIL	MACRO	5.2	MACLIB	LOADS PTR AT TAIL OF ATOM
TESTB	MACRO	5.5	MACLIB	TEST BIT
THEN	MACRO	8.7	MACLIB	COND - END PRED; START TRUE PART
TOP	MACRO	8.4	MACLIB	GETS TOP OF STACK-BUT LEAVES IT
TOPN	MACRO	8.4	MACLIB	GETS NTH ITEH ON SJACK
TYMAK	MACRO	8.6	MACLIB	MAKE A TRANSFER VECTOR FOR CAL
VALUE	MACRO	8.3	MACLIB	CREATES AN ATOM WITH A VALUE
XB	MACRO	B.6	MACLIB	TRANSFER INTO MIDDLE OF SUBROUTINE
AT	MISC	M	CSSWYM	EQUATED TO ATOM OFFSET(6)
ATCO	MISC	E.3	CSGC	PART OF ATCOL FOR TYPE 0 ATOMS
ATC1	MISC	E.3	CSGC	PART OF ATCOL FOR TYPE 1 ATOMS
CHOKE	MISC	E.3	CSGC	RRANCH TO IF STORE EXHAUSTED,ABEND
FALSE	MISC	G	CSSUYM	LA1,NIL;RET: (BRANCH TO IT)
FINISH	MISC	G	CSMAIN	CLOSE FILES AND EXIT
GCABEND	MISC	E.3	CSGC	BAL TO IF DATA SJRUCJURE ERR, ABEND
GCPUT	MISC	E.3	CSGC	BAL'ED TO BY GCPUT MACRO
INIT	MISC	G	CSINIT	SET UP SWYM REGS AND OPEN FILES

MNEMONIC	CLASS	APP	CSECT	COMMENTS
MAIN	MISC	0.1	CSMAIN	MAIN LOOP Of STUTTER INTERPRETER
PUTCH	MISC	G	CSSWYM	PUT CHARACTER IN PRINT LINE
RSTA1	MISC	6.1	CSSWYM	RST(A1). BAL'EDTO BY RST MACRO
RSTA2	MISC	8.1	CSSUYM	RST(A2). BAL'EDTO BY RST MACRO
R STA3	MISC	R.1	CSSUYM	RST(A3). BAL'EDTO BY RST MACRO
RSTT	MISC	B.1	CSSWYM	RST(T). BAL'EDTO BY RST MACRO
RSTTT	MISC	B.1	CSSUYM	RST(TT). BAL'EDTO BY RST MACRO
SWERROR	MISC	G	CSSUYM	SYSTEM ERROR
TRUE	MISC	G	CSSUYM	L A1,T; RET; (BRANCH TO IT)
A I	REG	I	CSSWYM	ARGUMENT REGISTER & RESULT REGISTER
A2	REG	I	CSSWYM	ARGUMENT REGISTER
A3	REG	I	CSSUYM	ARGUMENT REGISTER
A4	REG	I	C SSUYM	ARGUMENT REGISTER
A5	REG	I	CSSUYM	ARGUMENT REGISTER
Ab	REG	I	CSSUYM	ARGUMENT REGISTER
B	REG	I	CSSUYM	RASE REG FOR ALL ROUTNS
c 4	REG	I	CSSUYM	ODD REGISTER CONTAINING F'4'
F	REG	I	CSSUYM	FREE STORAGE POINTER
L	REG	I	CSSUYM	LINKAGE REG (RETURN ADDRESS)
N	REG	I	C SSUYM	POINTS AT NIL
P	REG	I	CSSUYM	STACK POINTER
S	REG	I	CSSUYM	RASE REG FOR CSSUYM
T	REG	I	CSSUYM	TEMP(EVEN, NEXT TO TT)
TT	REG	I	C SSUYM	TEMP (ODD, NEXT TO T)
CHAROBS	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL CHARS
EXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXPR FUNCTIONS
FEXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXP SPECIAL FNCTS
FPROPS	STRUC	H	C SFREEST	STRUCTIJRE:((SUBR.1)(FSUBR....
FSUBR	STRUC	0.2	C SFREEST	INDICATOR FOR ASSEMBLED SPECIAL FNC
NIL	STRUC	H	CSFREEST	ATOM WITH VALUE-NIL
OBLI ST	STRUC	H	CSFREEST	ATOM UITH VALUE - LIST OF ALL ATOMS
SUBR	STQUC	0.2	CSFREEST	INDICATOR FOR ASSEMBLED FUNCTIONS
T	STRUC	H	CSFREEST	ATOM WITH VALUE-T
UNROUND	STRUC	H	CSFREEST	RECOGNIZED BY EVAL AS ERROR VALUE
ATOM	SURR	f.1	CSSUBS	STUTTER ROUTINE FOR-IS ARG ATOM?
BELL	SUBR	F.5	CS2250	RINGS BELL ON 2250
EJECT	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT PAGE
EQ	SUBR	F.I	CSSUBS	STUTTER RTN FOR-ARG1=ARG2?
ERROR	SUBR	F.5	CSSUBS	WRITES MESSAGE AND GOES TO TOP LVL
EVAL	SUBR	0.3	CSEVAL	STUTTER INTRPRTR EXPRSN EVALUATOR
EXPLODE	SUBR	F.3	CSEVAL	CONVERTS ATOM TO LIST CHARS IN PNAM
FST	SUBR	f.1	CSSUBS	STUTTER RTN FOR-1STELEM OF LIST
GC	SUBR	E.3	CSGC	CONTROLS GARBAGE COLLECTION
GET	SURR	F.4	CSFVAL	FINDS PROPERTY OF AN ATOM
GETOBJ	SUBR	f.2	CSREAD	FINDS SYMBOL FOR CHAR STRING ARG
IVCCH	SUBR	F.2	CSREAD	RETURNS NEXT INPUT CHAR
IVQMO	SUBR	F.2	CSQEAD	RETURNS STATUS OF QUOTE MODE
MAKSTRNG	SUBQ	F.2	CSREAD	MAKES CHR STR ATM FROM LIST OF CHRS
NULL	SUBR	F.I	CSSUBS	STUTTER RTN FOR - IS ARG = NIL?
PRINT	SUBR	F.3	CSPRINT	PRINTS ITS ARG AND GOES TO NEXT LIN
PRIN1	SUBR	F.3	CSPRINT	PRINTS ITS ARG
PUTPROP	SUBR	F.4	CSEVAL	STORES PROPERTIES ON ATOMS PROP LST

MNEMONIC	CLASS	APP	CSECT	COMMENTS
READ	SUBR	F.2	CSREAD	READS ONE EXPRESSION FROM CARD
R EADCH	SUBR	F.2	CSREAD	READS ONE CHARACTER FROM CARD
REHPROP	SUBR	F.4	CSEVAL	REMOVES PROPERTIES FROM P-LIST
RST	SUBR	F.1	CSSUBS	STUTTER RTN FOR - REST OF LIST
SASSOC	SUBR	F.4	CSEVAL	FINDS ARG ON AN ASSOCIATION LIST
STIVCCH	SUBR	F.2	CSREAD	SETS CURRENT INPUT CHAR
STIVQMO	SUBR	F.2	CSREAD	SETS OUOTE MODE
TAK2	SUBR	F.1	CSSUBS	MAKES LIST W/ FST ARG1 AND RST ARG2
TERPRI	SURR	F.3	CSPRINT	MOVES PRINTER TO NEXT LINE
ATAMT	SWYM	c	CSSWYM	ATOM OFFSET (6)
CARDRDR	SWYM	M	CSSWYM	DCB FOR READING CARDS
DUBUORK	SWYM	M	CSSUYM	DOUBLE WORD WORK AREA
FEND	SWYM	E.4	CSSUYM	POINTS AT END OF FREE SOTR
GCABAD	SWYM	E . 4	CSSWYM	CC ABENDS FOR BAD DATA STRUCTURE
GCTIME	SWYM	E.4	CSSUYM	GC COMPUTES ITS TIME
MEMNXT	SUYM	E . 4	CSSWYM	ALTERNATE FREE STOR
MEMSIZ	SWYM	E.4	CSSUYM	SIZE OF FREE STORAGE
MEMUSE	SUYM	E.4	CSSWYM	FREE STOR IN USE
NUMAT	SUYM	M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
NUMATVAL	SUYM	-M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
PBHD	SUYM	C	CSSWYM	HOLDS ADRS Of AT-HO DURING PUTBYTE
PRATBAD	SWYM	F . 3	CSSWYM	AREA FOR PRINGING '?TYPN'
PRINTER	SWYM	M	CSSWYM	DCB FOR PRINTING
PRLNG	SWYM	F.3	CSSUYM	LENGTH OF PRINT LINE
PRPEND	SUYM	F.3	CSSUYM	WHERE TO PUT LAST PRINT CHAR
PRPT	SWYM	F.3	CSSUYM	WHERE TO PUT NXT PRINT CHAR
RDCHAR	SWYM	C	CSSWYM	LAST CHAR READ
ROCLASS	SUYM	C	CSSWYM	CLASS OF LAST CHARACTER READ
RDCOL	SWYM	C	CSSWYM	LOC Of LAST WORD READ
RDEND	SUYM	C	CSSUYM	LOC OF LAST CHAR TO READ
RDERCNT	SWYM	C	CSSWYM	PRINT #PARENS CREATED BEFORE '>'
RDERLOC	SWYM	C	CSSWYM	SYNTAX ERROR CARD COLUMN INDICATION
RDERHS	SUYM	C	CSSUYM	READ SYNTAX ERROR MESSAGE AREA
RDERNO	SWYM	c	CSSWYM	SYNTAX ERROR NUMBER
RDLNG	SWYM	c	CSSWYM	NUMBER OF CHAR READ FROM EACH CARD
RDSTAT	SWYM	C	CSSUYM	READ ROUTINES STATUS INFO BYTE
ROSUPCTR	SWYM	C	CSSUYM	COUNT #PARENS CREATED BEFORE '>'
ST	SWYM	M	CSSWYM	POINTER AT T
SWYM	SUYM	M	CSSUYM	FIRST LOC IN CSSUYM
SUYMSAVE	SWYM	M	CSSWYM	SAVE AREA FOR CALLING OS
SYSFDO	SWYM	M	CSSWYM	SAVE AREA FOR SAVING OS LIMK REGS
TIME	SUYM	M	CSSUYM	TIME SET AT LAST STIME
TVEND	SUYM	M	CSSWYM	LABEL OF LAST ENTRY IN TV TABLE
TVSTART	SWYM	M	CSSWYM	LABEL OF START OF TRANS VECT TABLE
VCHAROBS	SUYM	M	CSSUYM	POINTER AT CHAR OBJECTS LIST
VFPROPS	SWYM	M	CSSWYM	POINTER AT FPROPS STRUCTURE
VOBLIST	SWYM	M	C SSUYM	POINTER AT ALL OBJECTS LIST
VUNBND	SWYM	M	CSSWYM	POINTER AT SPECIAL 'UNBOUND'
#M1M2	SUYM	E.4	CSSUYM	USED BY GC TO 'OR' IN MI & M2 BITS
#PO	SWYM	M	CSSWYM	ADRS OF BEGINNING OF STACK
#XXXX	SUYM	M	CSSUYM	TRANSFER VECTOR, AORS Of RTN XXXX

MNEMONIC	CLASS	APP	CSECT	COMMENTS
ATOM	MACRO	0.1	MACLIB	? IS ARG AN ATOM
EQ	MACRO	0.1	MACLIB	? ARG1 =ARG2(TESTS TWO POINTERS)
FST	MACRO	6.1	MACLIB	FIRST ELEMENT OF LIST
NULL	MACRO	B.1	MACLIB	? ARG = NIL
RPLF	MACRO	B.1	MACLIB	REPLACES FIRST PTR OF LIST
RST	MACRO	B.1	MACLIB	ALL BUT 1ST ELEMENT OF LIST
RSTA1	MISC	6.1	CSSUYM	RST(A1). BAL'EDTO BY RST MACRO
RSTA2	MISC	B.1	CSSWYM	RST(A2). BAL'EDTO BY RST MACRO
RSTA3	MISC	8.1	CSSUYM	RST(A3). BAL'EDTO BY RST MACRO
RSTMAK	MACRO	6.1	MACLIB	MAKE ROUTINES FOR 'RST' TO BAL TO
RSTT	MISC	0.1	CSSUYM	RST(T). BAL'EDTO BY RST MACRO
RSTTT	HISC	B.1	CSSWYM	RST(TT). BAL'EDTO BY RST MACRO
CELL	MACRO	B.2	MACLIB	LOADS ATOM CELL INTO REG
GETNAME	MACRO	0.2	YACLIB	LOADS PTR AT PNAME CHR STR ATM
GETNUM	MACRO	0.2	MACLIB	GET VALUE OF NUM CHAR STR ATOM
HEAD	MACRO	5.2	MACLIB	LOADS HEAD OF ATOM
RPLC EL	MACRO	8.2	MACLIB	REPLACES ATOM CELL
RPLHD	MACRO	8.2	MACLIB	REPLACES HEAD OF ATOM
TAIL	MACRO	8.2	MACLIB	LOADS PTR AT TAIL OF ATOM
CHAR	MACRO	0.3	MACLIB	CREATES A CHAR OBJECT ATOM
EVCH	MACRO	B.3	MACLIB	GETS ARITH VAL OF EBCDIC BITS
F SUBR	MACRO	0.3	MACLI B	CREATES AN ATOM WITH FSUBR PROP
HASH	MACRO	0.3	MACL IB	HASH CODE AN IDENT FOR OBLIST
YATOM	MACRO	0.3	MACLIB	CREATES AN ATOM STRUC (IN CSFREEST)
QCHAR	MACRO	0.3	MACLIB	CREATES A CHAR OBJ FOR '('')'','
STRAT	MACRO	8.3	MACLIB	CREATES STRING ATOM STRUC (FREEST)
SUBR	MACRO	8.3	MACLIB	CREATES AN ATOM WITH SUBR PROPERTY
VALUE	MACRO	8.3	MACLIB	CREATES AN ATOM WITH A VALUE
POP	MACRO	0.4	MACLIB	GETS TOP OFF STACK-REDUCES STACK
POPN	MACRO	0.4	MACLIB	REDUCES STACK N TIMES
PUSH	MACRO	B.4	MACLIB	PUTS ARG ATOP STACK
RPLTOP	MACRO	0.4	MACLIB	REPLACE TOP ITEM ON STACK
RPLTOPN	MACRO	8.4	MACLIB	REPLACE NTH ITEM OF STACK
TOP	MACRO	8.4	MACLI B	GETS TOP OF STACK-BUT LEAVES IT
TOPN	MACRO	8.4	MACLIB	GETS NTH ITEM ON STACK
BIT	MACRO	8.5	MACLIB	IDENTIFY MNEMONIC WITH BIT IN WORD
B ITTBLHK	MACRO	8.5	MACLIB	MAKE A TABLE FOR 'BIT'MACRO
FINDRIT	MACRO	8.5	MACL IB	FIND BIT MNEMONIC FOR BYTE-IN-WORD
I NVFR TB	MACRO	0.5	MACLIB	CHANGE BIT
RESETB	MACRO	8.5	MACLIB	TURN OFF BIT
SETBIT	MACRO	B.5	MACLIB	TURN ON BIT
TESTB	MACRO	B.5	MACLIB	TEST BIT
CAL	MACRO	B.6	MACLIB	SUBROUTINE CALL
RET	MACRO	8.4	MACLIB	SUBROUTINE RETURN
SUB	MACRO	B.6	MACLIB	SUBROUTINE ENTRY
TVMAK	MACRO	B.6	MACLIB	MAKE A TRANSFER VECTOR FOR CAL
XB	MACRO	0.6	MACLIB	TRANSFER INTO MIDDLE OF SUBROUTINE
AND	MACRO	8.7	MACLIB	COMBINE TWO PREOS
BCMAC	MACRO	B.7	MACLIB	MAKE A BR CONDITION INSTRUCTION
ELSE	MACRO	8.7	MACLIB	COND - END TRUE; START FALSE PART
END IF	MACRO	6.7	MACL IB	COND - END FALSE; END CONDITIONAL
GOTQ	MACRO	B.7	MACL IB	BRANCH
I F	MACRO	0.7	MACLIB	COND - START PREDICATE
NOT	MACRO	6.7	MACLI B	NEGATE PREDICATE MACRO TEST
ORX	MACRO	B.7	YACLIB	COMBINE TWO PREDS

MNEMONIC	CLASS	APP	CSECT	COMMENTS
THEN	MACRO	0.7	MACLIB	COND - END PRED; START TRUE PART
CHTBL	MACRO	0.8	MACLIB	MAKE A CHARACTER TABLE (FORTR)
FIXUP	MACRO	0.8	MACLIB	GC-MAKE ENTRY IN FIXUP TABLE
GCPUT	MACRO	0.8	MACLIB	GC-PUT WORD TO NEW CORE
INST4	MACRO	0.8	MACLIB	ASSEMBLE INSTRUCTION WO/ ALIGN ERR
SWEAR	MACRO	0.8	MACLIB	SYSTEM ERROR
ATAMT	SWYM	C	CSSWYM	ATOM OFFSET (6)
CSREAD	CSECT	C	CSREAD	READ ROUTINES
GETCH	CAL	C	CSREAD	GET A CHARACTER
PBCLOSE	CAL	C	CSREAD	FINISH CHAR STRING ATOM
PBHD	SUYM	C	CSSWYM	HOLDS ADRS OF AT-HD DURING PUTBYTE
PBOPEN	CAL	C	CSREAD	START MAKING CHAR STRING ATOM
PUTBYTE	CAL	C	CSREAO	PUT BYTE INTO CHAR STRING
RDAT	CAL	C	CSREAD	READ AN ATOM
RDCHAR	SWYM	C	CSSUYM	LAST CHAR READ
RDCLASS	SWYM	C	CSSUYM	CLASS OF LAST CHARACTER READ
RDCOL	SWYM	C	CSSUYM	LOC OF LAST WORD READ
ROEND	SWYM	C	CSSWYM	LOC OF LAST CHAR TO READ
RDERCNT	SWYM	C	CSSWYM	PRINT #PARENS CREATED BEFORE '>'
ROERLOC	SWYM	C	CSSWYM	SYNTAX ERROR CARD COLUMN INOICATION
RDERMS	SUYM	C	CSSUYM	READ SYNTAX ERROR MESSAGE AREA
RDERNO	SWYM	C	CSSUYM	SYNTAX ERROR NUMBER
RDERR	CAL	C	C SREAD	TNDICA'TE INPUT SYNTAX ERROR
RDERRCNT	CAL	C	CSREAD	SYNTAX ERR-PARENS MADE BEFORE '>'
RDLIST	CAL	C	CSREAD	READ A LIST
RDLNG	SWYM	C	CSSUYM	NUMBER OF CHAR READ FROM EACH CARD
RDSE	CAL	C	CSREAD	READ AN S-EXPRESSION
RDSTAT	SUYM	C	CSSUYM	READ ROUTINES STATUS INFO BYTE
RD SUPCTR	SUYM	C	CSSUYM	COUNT #PARENS CREATED BEFORE '>'
CSEVAL	CSECT	D	CSEVAL	INTERPRETER AND RELATED ROUTINES
MAIN	MISC	0.1	CSMAIN	MAIN LOOP OF STUTTER INTERPRFTER
EXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXPR FUNCTIONS
FEXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXP SPECIAL FNCTS
FSUBR	STRUC	0.2	CSFREEST	INDICATOR FOR ASSEMBLED SPECIAL FNC
SUBR	STRUC	0.2	C SFREEST	INDICATOR FOR ASSEMBLED FUNCTIONS
BINDERY	CAL	0.3	CSEVAL	BIND ARG ATOMS TO THEIR VALUFS
EVAL	SUBR	0.3	C SEVAL	STUTTER INTRPRTR EXPRSN EVALUATOR
EVGET	CAL.	0.3	CSEVAL	GET FUNCTION DEFINITION OF ATOM
EVLIS	CAL	0.3	CSEVAL	EVALUATE LIST OF EXPRESS IONS
UNBIND	CAL	0.3	CSEVAL	RESTORE OLD BINDINGS OF ARG ATOMS
CSGC	CSECT	E	CSGC	GARBAGE COLLECTOR
Ml	FIELD	E.2	CSSUYM	GARB COL MARKING BIT
M2	FIELD	E.2	C SSUYM	GARB COL MARKING BIT
ATCOL	CAL	E.3	CSGC	COLLECTS AN ATOM
AT-CO	MISC	E.3	CSGC	PART OF ATCOL FOR TYPE 0 ATOMS
ATC1	MISC	E.3	C SGC	PART OF ATCOL FOR TYPE 1 ATOMS
CHOKE	MISC	E.3	CSGC	BRANCH TO IF STORE EXHAUSTED, ABEND
COLLECT	CAL	E.3	CSGC	CREATES IMAGE OF ARG IN NEW CORE
COLX	CAL	E.3	C SGC	CHECKS AND COLLECTS ONE POINTER
GC	SUBR	E.3	CSGC	CONTROLS GARBAGE COLLECT ION
GCABEND	MISC	E.3	C SGC	BAL TO IF DATA STRUCTURE ERR, ABEND
GCPUT	MISC	E.3	CSGC	BAL'ED TO BY GCPUT MACRO

SUYM MNEMONICS SORTEO BY APPENDIX

MNEMONIC	CLASS	APP	CSECT	COMMENTS
F END	SWYM	E.4	CSSWYM	POINTS AT END OF FREE SOTR
GCARAD	SWYM	E.4	CSSUYM	GC ABENDS FOR BAD DATA STRUCTURE
GCTIME	SUYM	E.4	CSSUYM	GC COMPUTES ITS TIME
MEMNXT	SUYM	E.4	CSSWYM	ALTERNATE FREE STOR
MEHSIZ	SUYM	E.4	CSSWYM	SIZE OF FREE STORAGE
MEMUSE	SWYM	E.4	CSSUYM	FREE STOR IN USE
#M1M2	SUYM	E.4	CSSWYM	USED BY GC TO 'OR' IN MI & M2 BITS
ATOM	SURR	F.1	CSSUBS	STUTTER ROUTINE FOR-IS ARG ATOM?
EQ	SUBR	F.1	CSSUBS	STUTTER RTN FOR-ARG1 = ARG2?
FST	SUBR	F.1	CSSUBS	STUTTER RTN FOR -1ST ELEM OF LIST
LIST	F SUBR	F.1	CSEVAL	MAKES A LIST OF THE ARG EXPRESSIONS
NULL	SUBR	F.1	C SSUBS	STUTTER RTN FOR - IS ARG = NIL?
QST	SUBR	F.1	CSSIJBS	STUTTER RTN FOR - REST OF LIST
TAK2	SUBR	F.1	CSSUBS	YAKES LIST W/ FST ARG1 AN0 RST ARG2
GETOBJ	SUBR	F.2	CSREAD	FINDS SYMBOL FOR CHAR STRING ARG
IVCCH	SUBR	F.2	CSREAD	RETURNS NEXT INPUT CHAR
I VQMO	SUBR	F.2	CSREAD	RETURNS STATUS OF QUOTE MODE
MAKSTRNG	SUBR	F.2	CSREAD	MAKES CHR STR ATM FROM LIST OF CHRS
READ	SUBR	F.2	C SREAD	READS ONE EXPRESSION FROM CARD
QEADCH	SUBR	F.2	C SREAD	READS ONE CHARACTER FROM CARD
STIVCCH	SUBR	F.2	C SREAO	SETS CURRENT INPUT CHAR
STIVQMO	SUBR	F.2	CSREAD	SETS QUOTE MODE
EJECT	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT PAGE
EXPLODE	SUBR	F.3	CSEVAL	CONVERTS ATOM TO LIST CHARS IN PNAM
PRATBAD	SUYM	F.3	CSSUYM	AREA FOR PRINGING '?TYPN'
PRINT	SUBR	F.3	CSPRINT	PRINTS ITS ARG AND GOES TO NEXT LIN
PRIN1	SUBR	F.3	CSPRINT	PRINTS ITS ARG
PRLNG	SWYM	F.3	CSSWYM	LENGTH OF PRINT LINE
PRPEND	SWYM	f.3	CSSUYM	WHERE TO PUT LAST PRINT CHAR
PRPT	SWYM	F.3	CSSUYM	WHERE TO PUT NXT PRINT CHAR
TERPRI	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT LINE
COND	FSUBR	F.4	CSEVAL	CONDITIONAL EXPRESSION EVALUATED
GET	SUBR	F.4	CSEVAL	FINDS PROPERTY OF AN ATOM
PUTPROP	SUBR	F.4	CSEVAL	STORES PROPERTIES ON ATOMS PROP LST
QUOTE	FSUBR	F.4	CSEVAL	RETURNS ITS ARG UNEVALUATED
REMPROP	SUBR	F.4	C SEVAL	REMOVES PROPERTIES FROM P-LIST
SASSOC	SUBR	F.4	CSEVAL	FINDS ARG ON AN ASSOCIATION LIST
BELL	SUBR	F.5	CS2250	RINGS BELL ON 2250
ERROR	SUBR	F.5	CSSURS	WRITES MESSAGE AND GOES TO TOP LVL
FALSE	MI SC	G	CSSUYM	L A1,NIL; RET; (BRANCH TO IT)
FINISH	MISC	G	CSMAIN	CLOSE FILES AND EXIT
INIT	MISC	G	CSINIT	SET UP SWYM REGS AND OPEN FILES
NLENGTH	CAL	G	CSEVAL	GET LENGTH OF LIST
PUTCH	MISC	G	CSSUYM	PUT CHARACTER IN PRINT' LINE
PUTSTR	CAL	G	CSPRINT	PRINT A CHARACTER STRING ATOM
STAKN	CAL	G	CSSUBS	GET FREE STORAGE BLOCK
STIHE	CAL	G	C SSUBS	START TIMER
SUFRROR	MISC	G	CSSWYM	SYSTEM ERROR
TRUE	MISC	G	CSSWYM	L A1,T; RET; (BRANCH TOIT)
TTIME	CAL	G	C SSUBS	HOW LONG SINCE LAST STIME
CHAROBS	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL CHARS
CSFREEST	CSECT	H	CSFREEST	FREE STORAGE, INCL INITIAL STRUCTS

MNEMONIC	CLASS	APP	CSECT	COMMENTS
FPROPS	STRUC	H	CSFREEST	STRUCTURE: ((SUBR . 1)(FSUBR . . . .
NIL	STRUC	H	CSFREEST	ATOM WITH VALUE-NIL
OBLIST	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL ATOMS
T	STRUC	H	CSFREEST	ATOM WITH VALUE-T
UNBOUND	STRUC	H	CSFREEST	RECOGNIZED BY EVAL AS ERROR VALUE
A1	REG	I	CSSUYM	ARGUMENT REGISTER & RESULT REGISTER
A2	REG	I	CSSUYM	ARGUMENT REGISTER
A3	REG	I	CSSWYM	ARGUMENT REGISTER
A4	REG	J	CSSUYM	ARGUMENT REGISTER
A5	REG	I	CSSUYM	ARGUMENT REGISTER
A6	REG	I	CSSWYM	ARGUMENT REGISTER
B	REG	I	CSSUYM	BASE REG FOR ALL ROUTNS
C4	REG	I	CSSUYM	ODD REGISTER CONTAINING F'4'
F	REG	I	CSSUYM	FREE STORAGE POINTER
L	REG	I	CSSWYM	LINKAGE REG (RETURN ADDRESS)
N	REG	I	CSSUYM	POINTS AT NIL
P	REG	I	CSSUYM	STACK PDINTER
S	REG	I	CSSUYM	BASE REG FOR CSSWYM
T	REG	I	CSSWYM	TEMP (EVEN, NEXT TO TT)
TT	REG	I	CSSWYM	TEMP (ODD, NEXT TO T)
AT	MISC	M	CSSUYM	EQUATED TO ATOM OFFSET(6)
CARDRDR	SWYM	M	CSSUYM	DCB FOR READING CARDS
CELFNC	FIELD	M	CSSUYM	ATOM HEAD-FUNC DEF TYPE BITS
CELREL	FIELD	M	CSSUYM	ATOM HEAD-CELL IS RELOCATABLE
CELVAL	FIELD	M	CSSWYM	ATOM HEAD-CELL HAS VALUE(NOTFNC)
CSSUYM	CSECT	M	CSSUYM	GLOBAL INFORMATION FOR SWYM RTNS
DUBUORK	SWYM	M	CSSUYM	DOUBLE WORD WORK AREA
NUMAT	SUYM	M	CSSUYM	WORK AREA FOR PRINTING NUMBERS
NUMATVAL	SWYM	M	CSSUYM	WORK AREA FOR PRINTING NUMBERS
PRINTER	SWYM	M	CSSWYM	OCB FOR PRINTING
ST	SWYM	M	CSSUYM	POINTER AT T
SYYM	SWYM	M	CSSWYM	FIRST LOC IN CSSUYM
SUYMSAVE	SUYM	M	CSSUYM	SAVE AREA FOR CALLING OS
SYSFOO	SWYM	M	CSSWYM	SAVE AREA FOR SAVING OS LIMK REGS
TIME	SUYM	M	CSSUYM	TIME SET AT LAST STIME
TVEND	SWYM	M	CSSUYM	LABEL OF LAST ENTRY IN TV TABLE
TVSTART	SUYM	M	CSSUYM	LABEL OF START OF TRANS VECT TABLE
VCHAROBS	SWYM	M	CSSUYM	POINTER AT CHAR OBJECTS LIST
VFPROPS	SWYM	M	CSSWYM	POINTER AT FPROPS STRUCTURE
VOBLIST	SUYM	M	CSSWYM	POINTER AT ALL OBJECTS LIST
VUNBND	SWYM	M	CSSUYM	POINTER AT SPECIAL 'UNBOUND'
#PO	SUYM	M	CSSUYM	AORS OF BEGINNING OF STACK
#XXXX	SWYM	M	CSSWYM	TRANSFER VECTOR' ADRS OF RTN XXXX
CSINIT	CSECT	0	CSINIT	INITIALIZATION
CSMAIN	CSECT	0	CSMAIN	MAIN STUTTER LOOP
CSPDL	CSECT	0	CSPDL	STACK
CSPRINT	CSECT	0	CSPRINT	PRINT ROUTINES
CSSUBS	CSECT	0	CSSUBS	BASIC SUBROUTINES
CS2250	CSECT	0	CS2250	2250 EXPERIMANTAL INTERFACE

HNEHONIC	CLASS	APP	CSECT	COMMENTS
BINDERY	CAL	0.3	CSEVAL	BIND ARG ATOMS TO THEIR VALUES
COND	FSUBR	F.4	CSEVAL	CONDITIONAL EXPRESSION EVALUATED
C SEVAL	CSECT	D	C SEVAL	INTERPRETER AND RELATED ROUTINES
EVAL	SUBR	0.3	CSEVAL	STUTTER INTRPRTR EXPRSN EVALUATOR
EVGET	CAL	0.3	CSEVAL	GET FUNCTION DEFINITION Of ATOM
EVLIS	CAL	0.3	CSEVAL	EVALUATE LIST OF EXPRESSIONS
EXPLODE	SUBR	F.3	CSEVAL	CONVERTS ATOM TO LIST CHARS IN PNAY
GET	SUBR	F.4	CSEVAL	FINDS PROPERTY OF AN ATOM
LIST	FSUBR	F. 1	CSEVAL	MAKES A LIST OF THE ARG EXPRESSIONS
NLENGTH	CAL	G	C SEVAL	GET LENGTH OF LIST
PUTPROP	SUBR	F.4	CSEVAL	STORES PROPERTIES ON ATOMS PROP LST
QUOTE	FSUBR	F.4	CSEVAL	RETURNS ITS ARC UNEVALUATED
REMPROP	SUBR	F.4	CSEVAL	REMOVES PROPERTIES FROM P-LIST
SASSOC	SUBR	F.4	CSEVAL	FINDS ARG ON AN ASSOCIATION LIST
UNBIND	CAL	0.3	CSEVAL	RESTORE OLD BINDINGS OF ARG ATOMS
CHAROBS	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL CHARS
CSFREEST	CSECT	H	CSFREEST	FREE STORAGE, INCL INITIAL STRUCTS
EXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXPR FUNCTIONS
FEXPR	STRUC	0.2	CSFREEST	INDICATOR FOR S-EXP SPECIAL FNCTS
FPRQPS	STRUC	H	CSFREEST	STRUCTURE: ((SUBR . 1) (FSUBR . . . .
FSUBR	STRUC	0.2	C SFREEST	INDICATOR FOR ASSEMBLED SPECIAL FNC
NIL	STRUC	H	C SFREEST	ATOM WITH VALUE-NIL
OBLI ST	STRUC	H	CSFREEST	ATOM WITH VALUE - LIST OF ALL ATOMS
SUBR	STRUC	0.2	CSFREEST	INDICATOR FOR ASSEMBLED FUNCTIONS
T	STRUC	H	CSFREEST	ATOM WITH VALUE-T
UNBOUND	STRUC	H	C SFREEST	RECOGNIZED BY EVAL AS ERROR VALUE
ATCOL	CAL	E.3	CSGC	COLLECTS AN ATOM
ATCO	MISC	E.3	CSGC	PART OF ATCOL FOR TYPE 0 ATOMS
ATC1	MISC	E.3	CSGC	PART Of ATCOL FOR TYPE 1 ATOMS
CHOKE	MISC	E.3	CSGC	BRANCH TO IF STORE EXHAUSTED, ABEND
COLLECT	CAL	E.3	CSGC	CREATES IMAGE OF ARG IN NEW CORE
COLX	CAL	E.3	CSGC	CHECKS AND COLLECTS ONE POINTER
CSGC	CSECT	E	C SGC	GARBAGE COLLECTOR
GC	SUBR	E.3	CSGC	CONTROLS GARBAGE COLLECT ION
GCABEND	MISC	E.3	CSGC	BAL TO IF DATA STRUCTURE ERR, ABEND
GCPUT	MISC	E.3	CSGC	BAL'ED TO BY GCPUT MACRO
CSINIT	CSECT	0	CSINIT	INITIALIZATION
INIT	MISC	G	CSINIT	SET UP SWYM REGS AND OPEN FILES
CSMAIN	CSECT	0	CSMAIN	MAIN STUTTER LOOP
F INISH	MISC	G	CSMAIN	CLOSE FILES AND EXIT
MAIN	MI SC	0.1	CSMAIN	PAIN LOOP OF STUTTER INTERPRETER
C SPDL	CSECT	0	CSPDL	STACK
CSPRINT	CSECT	0	CSPRINT	PRINT ROUTINES
EJECT	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT PAGE
PRINT	SUBR	f.3	CSPRINT	PRINTS ITS ARG AND GOES TO NEXT LIN
PRIN1	SUBR	F.3	CSPRINT	PRINTS ITS ARG
PUTSTR	CAL	G	CSPRINT	PRINT A CHARACTER STRING ATOM
TERPRI	SUBR	F.3	CSPRINT	MOVES PRINTER TO NEXT LINE

MNEMONIC	CLASS	APP	CSECT	COMMENTS
CSREAO	CSECT	C	CSREAD'	READ ROUTINES
GETCH	CAL	C	C SREAD	GET A CHARACTER
GETOBJ	SUBR	F.2	CSREAD	FINDS SYMBOL FOR CHAR STRING ARG
IVCCH	SUBR	F.2	CSREAD	RETURNS NEXT INPUT CHAR
IVQMO	SUBR	F.2	CSQEAD	RETURNS STATUS OF QUOTE MODE
MAKSTRNG	SUBR	F.2	CSREAD	MAKES CHR STR ATM FROM LIST OF CHRS
PBCLOSE	CAL	C	C SREAD	FINISH CHAR STRING ATOM
PROPFN	CAL	C	CSREAD	START MAKING CHAR STRING ATOM
PUTEYTE	CAL	C	C SREAD	PUT BYTE INTO CHAR STRING
RDAT	CAL	C	CSREAD	QEAO AN ATOM
RDERR	CAL	C	CSREAD	INDICATE INPUT SYNTAX ERROR
RDERRCNT	CAL	C	CSREAD	SYNTAX ERR-PARENS MADE BEFORE '>'
RDLIST	CAL	C	C SREAD	READ A LIST
RDSE	CAL	C	CSREAD	READ AN S-EXPRESSION
READ	SUBR	F.2	CSREAD	READS ONE EXPRESSION FROM CARD
READCH	SUBR	F.2	C SREAD	READS ONE CHARACTER FROM CARD
STIVCCH	SUBR	F.2	C SREAD	SETS CURRENT INPUT CHAR
STIVQMO	SUBR	F.2	CSREAO	SETS QUOTE MODE
ATOP	SUBR	F.1	C SSUBS	STUTTER ROUTINE FOR-IS ARG ATOM?
CSSUBS	CSECT	''0	c SSUBS	BASIC SUBROUTINES
EQ	SUBR	F.I	CSSUBS	STUTTER RTN FOR-ARG1=ARG2?
ERROR	SUBR	F.5	c SSUBS	WRITES MESSAGE AND GOES TO TOP LVL
FST	SUBR	F.I	c SSUBS	STUTTER RTN FOR -1STELEM OF LIST
NULL	SUBR	F.1	C SSUBS	STUTTER RTN FUR - IS ARG = NIL?
RST	SUBR	F.1	c SSUBS	STUTTER RTN FOR - REST OF LIST
STAKN	CAL	G	c SSUBS	GET FREE STORAGE BLOCK
STIME	CAL	G	CSSUBS	START TIMER
TAK2	SUBR	F.1	cSSUBS	MAKES LIST W/FSTARG1 AND RST ARG2
TTIHE	CAL	G	c SSURS	HOW LONG SINCE LAST STIME
AT	MISC	M	CSSWYM	EQUATED TO ATOM OFFSET(6)
ATAMT	SWYM	C	CSSWYH	ATOM OFFSET (6)
A I	REG	I	CSSWYM	ARGUMENT REGISTER & RESULT REGISTER
A2	REG	I	C SSWYM	ARGUMENT REGISTER
A3	REG	I	CSSWYM	ARGUMENT REGISTER
A4	REG	I	CSSWYH	ARGUMENT REGISTER
A5	REG	I	CSSWYM	ARGUMENT REGISTER
A6	REG	I	CSSWYM	ARGUMENT REGISTER
B	REG	I	CSSWYM	BASE REG FOR ALL ROUTNS
CARORDR	SWYM	M	CSSWYM	DCB FOR READING CARDS
CELFNC	FIELD	M	CSSWYM	ATOM HEAD-FUNC DEF TYPE BITS
CELREL	FIELD	M	C SSWYM	ATOM HEAD-CELL IS RELOCATABLE
C ELVAL	FIELD	M	CSSWYM	ATOM HEAD-CELL HAS VALUE(NOTFNC)
CSSWYM	CSECT	M	c SSWYM	GLOBAL INFORMATION FOR SWYM RTNS
c 4	REG	I	CSSUYM	ODD REGISTER CONTAINING F'4'
DUBWORK	SWYM	M	CSSWYM	DOUBLE WORD WORK AREA
F	REG	I	CSSWYM	FREE STORAGE POINTER
FALSE	MISC	G	CSSWYM	L A1,NIL; RET; (BRANCH TO IT)
FEND	SWYM	E.4	CSSWYM	POINTS AT END OF FREE SOTR
GCABAD	SWYM	E.4	CSSWYM	GC ABENDS FOR BAD DATA STRUCTURE
GCTIME	SWYM	E.4	CSSWYM	GC COMPUTES ITS TIME
L	REG	I	C SSWYM	LINKAGE REG (RETURN ADDRESS)
MEMNXT	SWYM	E.4	CSSWYM	ALTERNATE FREE STOR
YEMSIZ	SWYM	E.4	CSSWYM	SIZE OF FREE STORAGE

MNEMONIC	CLASS	APP	CSECT	COMMENTS
MEMUSE	SWYM	E.4	CSSWYM	FREE STOR IN USE
MI	FIELD	E.2	CSSWYM	GARB COL MARKING BIT
M2	FIELD	E.2	CSSWYM	GARB COL MARKING BIT
N	REG	I	CSSWYM	POINTS AT NIL
NUMAT	SWYM	M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
NUMATVAL	SWYM	M	CSSWYM	WORK AREA FOR PRINTING NUMBERS
P	REG	I	CSSWYM	STACK POINTER
PBHD	SWYM	C	CSSUYM	HOLDS ADRS OF AT-HD DURING PUTBYTE
PRATBAD	SWYM	F.3	CSSWYM	ARE4 FOR PRINGING '?TYPN'
PRINTER	SWYM	M	CSSYYM	DCB FOR PRINTING
PRLNG	SWYM	F.3	CSSWYM	LENGTH OF PRINT LINE
PRPEND	SWYM	F.3	CSSUYM	WHERE TO PUT LAST PRINT CHAR
PRPT	SWYM	F.3	CSSWYM	WHERE TO PUT NXT PRINT CHAR
PUTCH	MISC	G	CSSWYM	PUT CHARACTER IN PRINT LINE
RBCHAR	SWYM	C	CSSWYY	LAST CHAR READ
RDCLASS	SWYM	C	CSSWYM	CLASS OF LAST CHARACTER READ
RDCOL	SWYM	C	CSSWYM	LOC OF LAST WORD READ
RDEND	SWYM	C	CSSUYM	LOC OF LAST CHAR TO READ
QDERCNT	SWYM	C	CSSWYM	PRINT #PARENS CREATED BEFORE '>'
RDERLOC	SWYM	C	CSSWYH	SYNTAX ERROR CARD COLUMN INDICATIDN
RDERMS	SWYM	C	CSSWYM	READ SYNTAX ERROR MESSAGE AREA
RDERNO	SWYM	C	CSSWYM	SYNTAX ERROR NUMBER
RDLNG	SWYM	C	CSSWYM	NUMBER OF CHAR READ FROM EACH CARD
RDSTAT	SWYM	C	CSSUYM	READ ROUTINES STATUS INFO BYTE
RDSUPCTR	SWYM	C	CSSWYM	COUNT #PARENS CREATED BEFORE '>'
RSTA1	MISC	6.1	CSSWYM	RST(A1). BAL'EDTO BY RST MACRO
RSTA2	MISC	B. 1	CSSWYM	RST(A2). BAL'EDTO BY RST MACRO
RSTA3	MISC	B. 1	CSSWYM	RST(A3). BAL'EDTO BY RST MACRO
RSTT	MISC	B.I	CSSWYM	RST(T). BAL'EOTO BY RST MACRO
R STTT	MISC	6.1	CSSWYM	RST(TT). BAL'EDTO BY RST MACRO
S	REG	I	CSSWYM	BASE REG FOR CSSWYN
ST	SWYM	M	CSSUYM	POINTER AT T
SWERROR	MISC	G	CSSWYM	SYSTEM ERROR
SWYM	SWYM	M	CSSWYY	FIRST LOC IN CSSWYM
SWYMSAVE	SWYM	M	CSSWYM	SAVE ARE4 FOR CALLING OS
SYSFOO	SWYM	M	CSSWYM	SAVE AREA FOR SAVING OS LIMK REGS
T	REG	I	CSSWYM	TEMP (EVEN, NEXT TO TT)
TIME	SWYM	M	CSSWYM	TIME SET AT LAST STIME
TRUE	MISC	G	CSSWYM	L A1,T; RET; (BRANCH TO IT)
TT	REG	I	CSSWYM	TEMP (ODD, NEXT TO T)
TVEND	SWYM	M	CSSWYM	LABEL OF LAST ENTRY IN TV TABLE
TVSTART	SWYM	M	CSSWYH	LABEL OF START OF TRANS VECT TABLE
VCHAROBS	SWYM	M	CSSWYM	POINTER AT CHAR OBJECTS LIST
VFPROPS	SWYM	M	CSSWYM	POINTER AT FPRDPS STRUCTURE
VOBLIST	SWYM	M	CSSWYM	POINTER AT ALL OBJECTS LIST
VUNBND	SWYM	M	CSSWYM	PCINTER AT SPECIAL 'UNBOUND'
#M1M2	SWYM	E.4	CSSWYM	USED BY GC TO 'OR' IN ML & M2 BITS
#PO	SWYM	M	CSSWYM	ADRS OF BEGINNING OF STACK
#XXXX	SWYM	M	CSSWYM	TRANSFER VECTOR, ADRS OF RTN XXXX
BELL	SUBR	F.5	CS2250	RINGS BELL ON 2250
CS2250	C SECT	0	CS2250	2250 EXPERIMANTAL INTERFACE
AND	MACRO	6.7	MACLIB	COMBINE TWO PREDS
A TOM	MACRO	6.1	MACLIB	? IS ARG AN ATOM

MNEMONIC	CLASS	APP	CSECT	COMMENTS
BCMAC	MACRO	8.7	MACLIB	MAKE A BR CONDITION INSTRUCTION
BIT	MACRO	8.5	MACLIB	IDENTIFY MNEMONIC WITH BIT IN WORD
BITTBLMK	MACRO	9.5	MACL 18	MAKE A TABLE FOR 'BIT'MACRO
CAL	MACRO	6.6	MACLIB	SUBROUTINE CALL
CELL	MACRO	8.2	MACLIB	LOADS ATOM CELL INTO REG
CHAR	MACRO	6.3	MACLIB	CREATES A CHAR OBJECT ATOM
CHTBL	MACRO	8.8	MACLIB	MAKE A CHARACTER TABLE (FOR TR)
ELSE	MACRO	8.7	MACLIB	COND - END TRUE: START FALSE PART
ENDIF	MACRO	8.7	MACLIB	COND - END FALSE; END CONDITIONAL
EQ	MACRO	8.1	MACLIB	? ARG1 = ARG2(TESTS TWO POINTERS)
EVCH	MACRO	6.3	MACLIB	GETS ARITH VAL OF EBCDIC BITS
FINDBIT	MACRO	8.5	MACLIB	FIND BIT MNEMONIC FOR BYTE-IN-WORD
F IXUP	MACRO	6.8	MACLIB	GC-MAKE ENTRY IN FIXUP TABLE
FST	MACRO	8.1	PACLIB	FIRST ELEMENT OF LIST
FSUBR	MACRO	6.3	MACLIB	CREATES AN ATOM WITH FSUBR PROP
GCPUT	MACRO	8.8	YACL IB	GC-PUT WORD TO NEW CORE
GETNAME	MACRO	B.2	MACL I B	LOADS PTR AT PNAME CHR STR ATM
GETNUM	MACRO	8.2	MACLIB	GET VALUE OF NUM CHAR STR ATOM
GOTO	MACRO	8.7	MACL 18	BRANCH
HASH	MACRO	B.3	MACL 18	HASH CODE AN IDENT FUR OBLIST
HEAD	MACRO	B.2	MACLIB	LOADS HEAD OF ATOM
I F	MACRO	8.7	MACLIB	COND - START PREDICATE
INST4	MACRO	8.8	MACLIB	ASSEMBLE INSTRUCTION WO/ ALIGN ERR
I NVERTB	MACRO	8.5	HACLI B	CHANGE BIT
MATOM	MACRO	8.3	PACLIB	CREATES AN ATOM STRUC (IN CSFREESTI
NOT	MACRO	6.7	MACLIB	NEGATE PREDICATE MACRO TEST
NULL	MACRO	6.1	MACLIB	? ARG = NIL
ORX	MACRO	8.7	MACLIB	COMBINE TWO PREDS
POP	MACRO	6.4	MACLIB	GETS TOP OFF STACK-HEDUCES STACK
POPN	MACRO	8.4	MAC118	REDUCES STACK N TIMES
PUSH	MACRO	8.4	CACLIB	PUTS ARG ATOP STACK
QCHAR	MACRC?	8.3	MACLIB	CREATES A CHAR OBJ FOR '(")",'
RESET8	MACRO	6.5	MACLIB	TURN OFF BIT
RET	MACRO	8.6	MACLIB	SUBROUTINE RETURN
RPLCEL	MACRO	8.2	MACLIB	REPLACES ATOM CELL
RPLF	MACRO	8.1	MACLIB	REPLACES FIRST PTR OF LIST
RPLHD	MACRO	B.2	PACLIB	REPLACES HEAD OF ATOM
RPLTOP	MACRO	8.4	MACLI8	REPLACE TOP ITEM ON STACK
RPLTOPN	MACRO	6.4	MACLIB	REPLACE NTY ITEM Of STACK
RST	MACRO	8.1	MACLIB	ALL BUT 1ST ELEMENT OF LIST
RSTMAK	MACRO	6.1	CACLIB	MAKE ROUTINES FOR 'RST' TO BAL TO
SETB IT	MACRO	6.5	MACLIB	TURN ON BIT
STRAT	MACRO	6.3	MACLIB	CREATES STRING ATOM STRUC (FREEST)
SUB	MACRO	8.6	MACLIB	SUBROUTINE ENTRY
SUBR	MACRO	8.3	MACLIB	CREATES AN ATOM WITH SUBR PROPERTY
SWEAR	MACRO	6.8	PACLIB	SYSTEM ERROR
TAIL	MACRO	8.2	MACLIB	Loam PTR AT TAIL OF ATOM
TESTB	MACRO	8.5	MACLIB	TEST BIT
THEN	MACRO	6.7	MACLIB	COND - END PRED; START TRUE PART
TOP	MACRO	8.4	MACLIB	GETS TOP OF STACK-BUT LEAVES IT
TOPN	MACRO	8.4	MACLIB	GETS NTH ITEM ON STACK
TVMAK	MACRO	6.6	MACLIB	MAKE A TRANSFER VECTOR FOR CAL
VALUE	MACRO	8.3	MACLIB	CREATES AN ATOM WITH A VALUE
XB	MACRO	6.6	MACL IB	TRANSFER INTO MIDDLE OF SUBROUTINE