THE IMPLEMENTATION OF

LISP 2

by

Stanley L. Kameny *
Lowell Hawkinson **
Clark Weissman *
Jeffrey A. Barnett *
Robert A. Saunders **
Erwin Book *
Donna Firth *
Paul W. Abrahams ***

* System Development Corporation, Santa Monica, California
** Information International, Inc., Los Angeles, California
*** Information International, Inc., New York, New York

Produced by SDC and III in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Projects Agency Information Processing Techniques Office and Subcontract 65-107.

THE IMPLEMENTATION OF LISP 2

by

Stanley L. Kameny *
Lowell Hawkinson **
Clark Weissman *
Jeffrey A. Barnett *
Robert A. Saunders **
Erwin Book *
Donna Firth *
Paul W. Abrahams ***

Abstract

This paper discusses the internal structure of the LISP 2 programming system and the means by which it was created. The system is written in its own language. The I/O package transforms input into a stream of characters, which are converted into tokens by the finite state machine. The supervisor controls the various LISP 2 operations. SL is translated to IL by the syntax translator; IL is translated to assembly language by the compiler; and assembly language is translated to machine language by the LISP 2 assembler, LAP. Machine mobility is achieved through core image generation. LISP 2 memory management is based on dynamic storage allocation, with separate areas for different kinds of data. Data is recovered by garbage collection. The syntax translator is generated from syntax equations by the META compiler. The compiler consists of three main parts: the analyzer, the optimizer, and the user control facilities. LAP handles the manipulation of the push-down stack automatically. LISP 2 is generated by a bootstrap procedure which successively produces LAP, LAP with IL, and full LISP 2 on the Q-32. A fourth stage produces LISP 2 on a new machine, requiring only an octal loader and a system monitor on the new machine.

* System Development Corporation, Santa Monica, California
** Information International, Inc., Los Angeles, California
*** Information International, Inc., New York, New York

INTRODUCTION

In this paper we shall discuss the internal structure of the LISP 2 programming system and the mechanisms by which it was created. It is assumed that the reader is already familiar with the companion paper "The LISP 2 Programming Language" ~~LISP 2".~~

Quite early in the design of LISP 2 we decided to write the LISP 2 system in its own language, and use the existing LISP 1.5 system on the Q-32 [1] in order to bootstrap in the first version. There were a number of reasons for this decision:

(1) Economy in programming. In general, it is more economical to program in a problem-oriented language than in assembly language. Although we might have used a problem-oriented language other than LISP 2 cum LISP 1.5, we felt that any advantages of another language would be offset by extra design, implementation, maintenance, and tutorial work.

(2) Machine mobility. We wished to be able to transfer LISP 2 to a new machine as rapidly as possible. This point is discussed in further detail below.

(3) Dynamic storage allocation. Whatever language was used for writing the system needed to have dynamic storage allocation facilities; these are of course inherent in the LISP 2 language.

(4) Modularity. By taking advantage of the natural modularity of LISP, we were able to partition the programming task easily, and modify different parts of

the system independently.

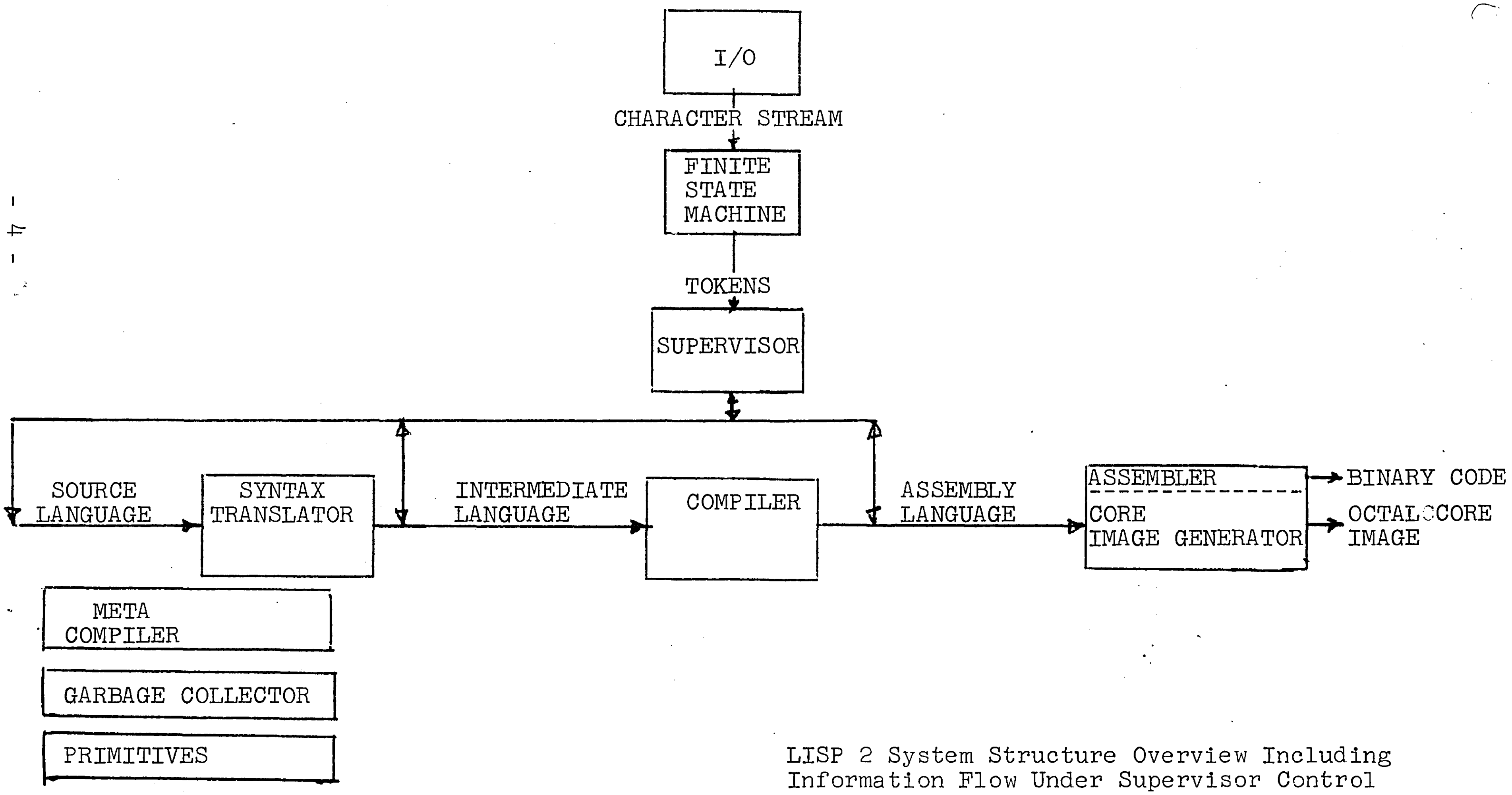
(5) Accessibility of system programs to the user. In the past, the system programs have appeared to the user to be an extension of his own program. He can call upon system functions as though he had written them himself, and he can modify the behavior of the system by changing the system functions. We wished to preserve this capability.

(6) Previous experience. The project personnel had already had experience with LISP 1.5, and we wished to take advantage of that experience.

(7) Capabilities of Q-32 LISP 1.5. The LISP 1.5 system on the Q-32 had quite useful editing and debugging capabilities, and it works in an interactive environment. By using Q-32 LISP 1.5 in the first part of the production of LISP 2, we were able to utilize these capabilities.

SYSTEM OVERVIEW

A diagram of the LISP 2 system, which shows the relationships among its different components, is shown in Figure 1. Information enters the system via the I/O package in either SL or IL. The I/O package transforms the input into a stream of characters in an internal representation. The stream of characters becomes the input to the finite state machine, which in turn generates a stream of tokens. Among other things, the finite state machine performs the task of linking up a newly received identifier with a previous copy of the same identifier. The token stream produced by the finite state machine is routed by the supervisor to either the syntax translator or to a reading program for IL, depending



LISP 2 System Structure Overview Including Information Flow Under Supervisor Control

Figure 1

upon whether SL or IL is expected. In either case, the result is an expression in IL. The supervisor determines when compilation is to take place, and also handles processing requests and declaration made by the user and error conditions that arise during computation.

The syntax translator takes a stream of SL tokens and transforms it into an IL expression. This expression can be returned as output, passed to the compiler, or both. The choice is made by the supervisor under the control of the user. The syntax translator consists of parsing and generating programs that are compiled from a set of syntax equations. These syntax equations define SL in terms of IL.

The compiler, which is the most complex component of the system, converts IL into input for LAP, the LISP Asembly Program, or for the core image generator. Both LAP and the core image generator accept input in assembly language (AL). If LAP is being used, then the result of assembly is a relocatable segment of code stored in an area of the machine reserved for binary programs. If the core image generator is being used, then the result is a string of pairs of binary numbers, each consisting of a core location and the contents of that location, stored on a magnetic tape or other external medium. The core image generator is only used when a new system is being created.

The META compiler, the garbage collector, and the primitives are all implicitly involved in the operation of the system. The META compiler is a library program that generates a syntax translator from a set of syntax equations. The garbage collector is the program that collects dead storage when available storage has been exhausted. The primitives are the basic library functions in terms of which the entire system is written.

MACHINE MOBILITY

The bulk of the design and implementation effort to produce the first LISP 2 system has been directed toward the Q-32 computer. The Q-32 is a large military machine of considerable power for its age, even by today's standards; however, it is to be retired by the close of the year. Why, then direct so much effort toward a machine with little future? The answer is that it is far easier to create a LISP 2 system when one already exists than to create one from scratch, and the Q-32 offered the quickest route to creating the first one. Given the Q-32 LISP 2 system, we could then transfer LISP 2 to the two other machines that we were really interested in--the IBM 360 and the DEC PDP-6. The problem was complicated further by the fact that the 360 and the PDP-6 were not even available to us when the effort began. Thus, machine mobility became one of the primary design goals of LISP 2.

The decision to write LISP 2 in LISP 2 achieved the machine mobility that we desired. In order to create a LISP 2 system on a new machine, we use an existing LISP 2 system to compile the new one. This process is known as "core image generation", and it will be discussed in detail below. Core image generation achieves machine mobility in two ways. First, the amount of code that has to be rewritten for the new machine is minimized, since most of it does the same thing that it did on the old machine and is written in the same language. Second, the translating facilities of the old machine are brought to bear on the task of producing code for the new machine, and there is virtually no dependence on the existing software for the new machine.

There were other important consequences of the decision to write LISP 2 in LISP 2. First, it was necessary that LISP 2 include facilities for referencing machine words, inserting and extracting bits, and performing type changes on a datum without changing the datum itself. (The type-changing requirement was met through the inclusion of "cheater functions", which accomplish directly what generations of FORTRAN programmers have accomplished painfully through EQUIVALENCE statements.) Second, it was necessary to write an optimizing compiler, so that the system programs themselves would operate at a tolerable speed. Although the requirements of the implementation were the primary reasons for these decisions, they alos had the side effect of making additional facilities available to the user.

MEMORY MANAGEMENT

Most of the concepts of memory management used in LISP 1.5 are also used in LISP 2. Memory management in LISP 2 is based upon several considerations:

(1) LISP 2 data may vary in size by orders of magnitude at run time, and storage for such data must be allocated automatically.

(2) Since recursion is permitted, many generations of data must be retained simultaneously.

(3) Programs and data that are no longer needed must be purged without explicit action on the part of the user.

(4) Numerical data must be stored in such a way as to permit efficient numerical calculations.

LISP 2 data may be either variable or fixed in size. The variable data are arrays and symbolic expressions. Symbolic expressions are stored in the form of list structure, with each cell of the structure representing a node of a binary tree that in turn represents the symbolic expression. Arrays are stored in the form of integer-indexable blocks of consecutive cells that may contain numerical or symbolic data. Although an array, once established, does not change in size, the size of an array is frequently not known until the occasion arises to creat it. In the case of list structures, the situation is even more complex; a list structure may be modified in such a way as to increase or decrease its size.

Arguments of functions and internal parameters of blocks are stored on a pushdown stack. Since all temporary storage belonging to LISP 2 functions is recorded on the pushdown stack, which is maintained by the LISP 2 system, recursion is permitted with no special user provisions. Unlike LISP 1.5, LISP 2 stores numbers directly on the pushdown stack as single cells. Therefore, it is possible to perform efficient arithmetic without the loss of efficiency that would arise from packing and unpacking numbers that are referenced indirectly. Symbolic expressions and arrays, however, are accessed by means of pointers stored in the stack. The data thus pointed to are discarded when the function creating them has completed its execution; however, they do not disappear, but remain as garbage until the next garbage collection (see below).

In most programming languages, variable-size data structures are only created upon entrance to a block, subroutine, or other program unit; in LISP 2, they may also be created during the operation of a program unit by, for example, the concatenation of two symbolic data. Therefore, the frequently-used solution of storing such data on the pushdown stack does not work in LISP 2. (It also fails, for much the same reasons, in the case of dynamic own arrays in ALGOL 60).

Data Storage Areas -- In LISP 2, data are grouped according to their storage characteristics and a storage area is set aside for each group. The groups are:

(1) Elementary symbolic entities (symbolic constants, function and variable names, etc.)

(2) Compiled programs

(3) List structure

(4) Arrays

In addition, a storage area is set aside for the pushdown stack. These storage areas are arranged in pairs, where one member of the pair grows from the bottom up and the other grows from the top down. Consequently, the allocation of space among the different groups is less critical than it would otherwise be.

The elementary symbolic entities are each stored as a symbol block (three cells in the case of the Q-32). Among the elementary symbolic entities are identifiers, each of which has a unique symbolic block associated with it. Each identifier has associated with it in turn a set of cells, possibly empty, that contain the values of the identifier. An identifier may have several values associated with it because it may be used in several different sections. The values are chained together in a circular list known as the v-f chain (where "v-f" stands for "variable-function"). The values themselves are stored as symbol blocks. Identifiers are linked together in bucket-sorted chains, which are in turn pointed to from an integer-indexed array. The chain in which an identifier is stored is found by the "hashing" technique of treating the first six characters of its name as an octal number, dividing this number by a constant, and taking the remainder. The purpose of this procedure is to minimize the time needed to find existing copies of an identifier upon read-in.

Compiled programs are stored in the form of relocatable code. Each code segment has relocation bits associated with it so that it can be moved if necessary. List structure is stored as a set of nodes, one node per cell.Each node contains CAR and CDR of the datum that it represents (with some bits left over). Arrays are stored as blocks of cells, with a title cell at the head. Numbers that are being used in a symbolic context, i.e., that are governed by a declaration of SYMBOL or are part of a symbolic expression, are stored as one-element arrays.

<u>Garbage Collection</u> -- In LISP 2, data storage is obtained by taking storage space from the appropriate area until that area is exhausted (which occurs when its boundary meets the boundary of the area that is paired with it). At this point, the garbage collector is invoked. Garbage collection causes all inaccessible data to be erased, and its space made available for new data. For instance, if a LISP 2 function has been redefined, the program corresponding to its old definition is inaccessible and thus is erased. During garbage collection, the different

areas are compacted, relocating code and/or data if necessary, so as to eliminate the gaps left by erased data.

The different kinds of data are stored in different areas because their requirements in terms of garbage collection are different. For instance, the elementary symbolic entities cannot be moved, but the other kinds of data can be moved. Similarly, list structure consists of independent single cells, while arrays consist of blocks of different sizes.

At entrance to the garbage collector, the current pushdown stack location is saved. The marking phase then is executed. At the beginning of the marking phase, all LISP 2 storage is unmarked. During the marking phase, a mark is placed in a mark bit in each datum that is accessible to either the user or the system. A pushdown scanning function applies a marking function to all pushdown cells pointing to list structure, passing from the beginning of the stack to the current stack location. The mark bit is in a uniform position in all marked words. All list nodes in use are marked. Arrays in use are marked in the title word. Symbol blocks are marked in the second word in the case of the Q-32.

A pass is then made over all identifiers. Each identifier's v-f chain is scanned, and unmarked symbol blocks on the chain are pruned off. If any remain, the identifier is marked if it was not marked before. Identifiers remaining unmarked are then deleted from the bucket sort chains. The freed triples are then chained together, and the boundary of the symbol block area adjusted to point to the end of the area in use.

Next, the list nodes are compacted. A pointer is set to the top of the list space, and another is set to the bottom. The pointers are advanced toward each other. When the top pointer has found an idle, i.e., unmarked, node and the bottom pointer has found a marked one, the marked node is copied into the idle position, and the old position is set to a pointer to the new position. It will thus be possible at a later time to detect and fix up references to the vacated area.

It is now possible to tell how much space of each type is actually in use. During the marking phase, the space occupied by arrays in use was added to a counter as they were marked. Binary program space is counted as it is assigned and excised. The other areas have all been counted by the garbage collector itself. Therefore, it is possible to reallocate storage among areas. There are three areas to consider: symbol blocks and pushdown, binary program space, and array and list space. It is clear that reallocation is much more important if some areas are nearly full than if all are lightly used, and that as the fullest area becomes fuller, one can tolerate less and less difference between the percentage occupancy of the fullest area and the emptiest. Hence we calculate the fraction of occupancy, of each area, calculate the largest and smallest of these, and reallocate if unity minus the largest is less than the largest minus the smallest. If reallocation is done, the available space is divided so as to equalize the percentage of occupancy. The areas are copied into their new positions, taking care to copy in such an order that no information is lost.

Now array space is reallocated. A pass over array space notes which arrays are marked. An address-sized field in each array title word is set to the position at which the array will finally be put. It is then possible to do a general sweep of storage, and update all references to arrays and list nodes. Binary program space is then reallocated and relocated, using the relocation bits included in the code. Finally, array space and binary program space are moved to their final positions.

Garbage collection as a means of storage recovery has significant advantages over its competitors, which are explicit erasure [2] and the use of reference counts [3]. In a system utilizing explicit erasure, the programmer designates those data to be returned to free space, and the data so designated are returned immediately. However, the programmer must be sure that when he erases a structure that there are no references to it in existence. In such a system, the results of either too little or too much erasure can be disastrous. (The LISP 2 programmer can, in effect, explicitly erase a list by setting all pointers to it to the null list.) In a system utilizing reference counts, each datum (or list node) has a count associated with it that specifies the number of times that it is currently referenced; when the reference count of a storage structure goes to zero, the structure can be erased. However, in this case more elaborate storage structures are required, and complicated updating must be done whenever a list is assigned to a variable. Reference counts are used to some extent in LISP 2 for symbol blocks, since updating these counts is straightforward and does not occur frequently, and the storage structure of symbol blocks is fairly complex in any case.

Garbage collection also simplifies many of the problems involved in dynamic creation of arrays.

There are two principal disadvantages to garbage collection: the overhead that it adds to the cost of creating list structure, and the fact that it occurs all at once and takes considerable time. The overhead is a direct consequence of the fact that the system rather than the user determines what storage is needed and what storage is not needed. The difficulty with having garbage collection occur all at once is that while the process is going on, the system is immobilized. If LISP 2 is being used for a real-time application, then the immobilization becomes intolerable if its duration exceeds the response time required.

THE SYNTAX TRANSLATOR AND THE META COMPILER

The translation from SL to IL is performed by a syntax translator which was generated by the META compiler. The META compiler is based upon a program developed by the Los Angeles SIGPLAN of the ACM [4]. The META compiler takes as input a specification of the syntax of SL, together with instructions on how each syntactic entity is to be transformed to IL. It produces an IL program that actually carries out the translation from SL to IL. The description of the syntax of SL is given in an extended version of Backus-Naur Form [5]. The extension allows both for the designation of things like indefinite numbers of occurrences and for the designation of the LISP program corresponding to a syntactic entity.

The META compiler produces top to bottom compilers with a controlled backup feature and an interface with the finite state machine. Both the controlled backup and the finite state machine are efficiency features. The controlled backup allows the designer of a language to specify in the syntax equations when the state of the machine must be saved because two or more alternatives start with the same construct or constructs. The finite state machine enables the syntax translator to parse expressions consisting of basic tokens of the source language instead of having to spend time reading expressions made up of characters. Since a large amount of the time of these compilers is spent examining characters the savings are considerable.

As it is possible to regenerate the syntax translator with new syntax equations at any time, the syntax and semantics of SL are not in principle rigidly fixed. In practice, variants on the syntax translator will be used in order to translate other languages into LISP 2 IL. These other languages, unlike SL, will normally not be semantically equivalent to IL.

INPUT-OUTPUT

One of the primary design aims in LISP 2 I/O has been the maintenance of machine independence as far as possible. This is accomplished by distinguishing user interfaces from system interfaces and insulating the user from the system interfaces. This effect is achieved by creating machine-independent data aggregates called "files", and permitting the user to operate with files by means of LISP 2 functions.

To the user, a file is a source or sink for information, which is filled on output and emptied on input. A file itself is both device and direction independent. The relationship of a file to an external device is determined by the user at run time, when he specifies whether the file is to be an input file, an output file, or both an input and an output file.

To the system, a file consists of a sequence of records, represented internally as an array of type OCTAL if the file is binary, and as a string if the file is composed of characters. (ASCII 8-bit characters are used internally throughout LISP 2.) To reduce buffer storage overhead, only one record for a given file can be in main memory at a time. String records are further structured into lines. The number of characters per line and lines per record are specified by the user, but must be consistent with the conventions used by the external monitor system.

When a record in a file is moved from an external device into core, it is transformed into a LISP 2 string. The transformation may involve character code conversions and insertion or deletion of control characters. The transformation is governed by a collection of control words associated with the file. During output this transformation, known as "string post-processing", is reversed.

File Activation and Deactivation. -- A file may be either active or inactive; an active file, in turn, may be either selected or deselected. No record is kept within LISP of inactive files; however, many files may be active concurrently.

A file is activated by evaluating the function OPEN which establishes all necessary communication linkages between LISP 2. and the monitor. The file is named by an identifier that is its referent throughout its active life. The user further specifies the desired file description at this time. This description is given only once and consists of a list of file properties desired by the user such as the unit (tape, disc, teletype, CRT, etc.), form (binary, ASCII, BCD, etc.), format (line and record sizes), and various protection and identification parameters.

Deactivation of a file is achieved by evaluating the function SHUT. SHUT breaks all the communication linkages and deletes all internal structures such as arrays, strings, and variables that were dynamically established by OPEN. The user may specify at this time the disposition of the file, e.g., save the tape or insert file in disc inventory. The external monitor is informed of such actions by LISP 2.

File Selection -- At any given time, exactly one file is selected for input and one for output; all other active files are deselected. The LISP 2 reading functions all operate on the currently selected input file; the printing functions all operate on the currently selected output file. The functions INPUT and OUTPUT are used for selecting the input file and the output file, respectively.

When a file is selected, the record, line and column controls for the deselected file are preserved, and the new file record, line, and column controls are reestablished. Once a file is selected, all I/O primitives act only on that file. Thus it is possible to write a LISP 2 program that is independent of form, format, and device by supplying the name of the file as an argument of the program at run time. This scheme allows a LISP program to be debugged with files generated on-line, and subsequently run with bulk data from tape or disc files simply by changing the selected file.

<u>Other I/O Functions</u> -- A variety of I/O functions are available for reading and writing binary and symbolic data. There are character level primitives that permit testing, printing, reading and transforming characters. Other functions call upon the finite state machine to allow reading at the LISP 2 token level, with equivalent token printing capability. There are also functions that read and print entire S-expressions. Additional features permit the user to control the form of printing and reading. By these means one can obtain formatted printing of S-expressions and special printing of tokens with unusual spellings that would ordinarily foil the finite state machine's parsing algorithms. There are special character mappings permitted so LISP 2 can accept legal input from restricted character-set devices.

<u>Finite State Machine</u> -- The finite state machine (FSM) is a token parsing program used by the syntax translator and the S-expression reader. Reading LISP 2 entities is expensive, not only in the original creation of the internal structures, but also in the time spent in garbage-collecting the space when they are discarded. Consequently, it is desirable to avoid backup at the character level with the resulting recreation of duplicate structures. Since backup must be used by the syntax translator, the FSM was imposed between it and the character stream to eliminate reprocessing of tokens. Having the bottom-to-top FSM interface with the top-to-bottom syntax translator eliminates a large portion of the overhead associated with reading in the LISP 2 system. The S-expression reader does not require backup, but since the FSM existed, it was convenient to use tokens for building S-expressions also.

The FSM behaves like a Turing machine. It moves from state to state as it reads characters; when a terminal state is reached it "prints" a character from its output alphabet (tokens) and sets its state to the initial one. Parsing and manufacture of structure are done simultaneously as characters are recognized. No reprocessing of the parsed characters is ever necessary since in a terminal state the token is already complete (except for a final action, such as combining the parts of a real number).

THE LISP 2 COMPILER

The LISP 2 compiler is a large one-pass optimizing translator whose input is a function definition in IL and whose output is an assembly-language list of instructions suitable for input to LAP. Most of the compiler is independent of the target machine, since the compilation concepts themselves are machine independent. The declarations of all fluid variables appearing within the function are written into the output listing, since these must agree with fluid variable declarations made elsewhere. Checks are made for both format and semantic errors during compilation. The compiler consists of three major sections: the analyzer, the optimizer, and the user control functions. Each of these will now be described.

Analyzer -- The top-level control of the compiler resides in the analyzer, which operates recursively. Each item to be compiled is passed to the analyzer either directly or indirectly. If the item is a variable, an appropriate declaration is found and code for retrieving the variable is generated; otherwise the code for a function call is generated, a macro expansion is performed and the result compiled, or linkage to an appropriate code generator is made. The analyzer is implemented by means of a pattern-matching function that compares an expression to be compiled against a pattern. The patterns are written in a modified form of Backus-Naur Form (not the same as the one used in the syntax translator). The pattern-matching function checks for syntactic correctness and distinguishes among different forms at the same time.

The analyzer needs to make special provision for situations where a GO statement transfers control from within the scope of a fluid variable declaration to outside that scope. This situation arises when a fluid variable is declared as an internal parameter of a block and a transfer takes place from within the block to a location outside the block. In this case, the current binding of the fluid variable disappears at the time of transfer and the previous binding must be restored.

As compilation proceeds, a list is kept of all labels to which transfers can be made. A list of all currently unsatisfied GO statements is also maintained. At the end of compilation of each block, checks are made for undefined labels. If any transfers out of a block are requested, a subroutine is generated to unbind the fluid variables of the block, restore their old values, and complete the transfer. When a forward transfer is requested which goes through one or more blocks, the check for label definition and fluid variable restoration may be made several times. When the appropriate information is finally obtained, the compiler patches the listing with the appropriate code.

<u>Optimizer</u> -- Optimization of the code produced by the LISP 2 compiler is handled by many groups of routines, each responsible for certain actions. The communicative mechanisms between these various parts and the rest of the compiler will be described in some detail below:

The movers are a highly machine dependent set of functions. They produce code that alters the state of a compilation in a specified way, such as moving an object to an accumulator or converting a datum to a specific type. Embodied in the movers is a predicate capability that answers the question "Is this move possible under these conditions (say one machine instruction)?" The movers are used to build all address and modifier fields of generated instructions. Associated with the movers is a post-processor that rewrites the output code after the main compiler has produced it. Redundant load-store sequences and some unnecessary branches are removed from the listing. Also, certain groups of instructions are rewritten to make use of machine-specific instructions.

The arithmetic optimization package handles code generation for addition and multiplication. The algorithm that is used is a standard one, namely, first sorting the arguments by type and then by priority sequence within a particular type. The sequence depends on whether the arguments are memory or accumulator references. A single set of functions handles both multiplication and addition, with the aid of several functional arguments.

Another kind of optimization is handled by the conditional expression processor. An example of a conditional expression is

$$(\text{IF } p_1\ e_1\ p_2\ e_2\ \text{- -}\ p_n\ e_n\ e_{n+1}).$$

The p_i are called the antecedents and the e_i are called the consequents. The value of this expression is defined thusly: evaluate the p_i in order, 1 to n, until one, say p_x, has the value TRUE. If such an x is found, e_x is the value of the entire IF expression. Otherwise the value is e_{n+1}. The code to implement this kind of expression would evaluate the p_i in order until one is found that is TRUE, and then evaluate the corresponding expression, say e_x. The value of e_x is brought to a standard accumulator, and program control is transferred forward. The transfer must be made to the same point, no matter which e_i is used for the value of the conditional expression. The point to which control is transferred is a confluence point.

Consider the following example:

$$(\text{IF } p_1\ \ (\text{IF } p_{11}\ e_{11}\ \text{---}\ p_{1n}\ e_{1n}\ e_{1(n+1)})\ p_2\ e_2\ \cdots\ p_m\ e_m\ e_{m+1})$$

For e_2 through e_{m+1} the code generation is as described above. However an interesting case arises if p_1 has value TRUE. The p_{1i} are then evaluated until one of these, say p_{1x}, is found to have value TRUE. e_{1x} is then both the value of the embedded conditional expression and, the value of the embedding expression. Therefore the embedded expression can share the confluence point of the embedding expression. Confluence points can be combined in this way for embedded conditional expressions nested to an arbitrary depth. In order to handle confluence points efficiently, the compiler is capable of operating in any one of five modes. When the analyzer is called, internal variables of the compiler are set so as to indicate which mode is applicable. In each mode, confluence points are handled differently. These modes are:

(1) Expression mode. In this mode of compilation, and expression is to be compiled and no confluence point has been established. If the expression to be compiled is a conditional expression or a block expression, a confluence point is established and compilation continues in the terminal expression mode.

(2) Terminal expression mode. In this mode of compilation, an expression is being compiled and a confluence point has been established by some higher-level embedding form.

(3) Statement mode. In this mode, a statement is being compiled that is not itself a consequent of a conditional statement. Such a statement produces a side effect but no value. If the statement is a conditional statement or a block statement, a confluence point is established and compilation continues in terminal statement mode.

(4) Terminal statement mode. In this mode, a consequent of a conditional statement is being compiled and a confluence point has already been established. As with conditional expressions, the confluence point may be shared by conditional statements enbedded to an arbitrary depth.

(5) Predicate mode. This mode is in effect when an antecedent of a conditional expression or conditional statement is being compiled. Tn this situation, the value of the antecedent is not used as a datum, but does affect the place to which program control must go. Therefore, two new confluence points are established: one for TRUE and one for FALSE. These confluence points are used in the compilation of AND and OR; if the predicate begins

with NOT, the two confluence points are reversed. Nested compositions of AND, OR, and NOT that are equivalent under De Morgan transformations produce the same code.

It is interesting to note that in compiling a BLOCK expression, an expression confluence point must be established since more than one RETURN statement may exist. The procedures for compiling IF expressions and BLOCK expressions are quite similar. However, the procedures for IF statements and BLOCK statements are different because BLOCK statements have neither an expression confluence point nor a statement confluence point.

When an expression is compiled, the characteristics of the value that it is to produce must be specified. These characteristics include its data type, whether it is in a special register or in an ordinary memory cell,its address modifier (direct or indirect), which registers it may be left in, whether the actual value is needed or whether the negative or reciprocal of the value will do, etc. These characteristics are specified by <u>state variables</u>, which are bound for each call to the analyzer. As a statement or expression is compiled, a listing is generated and the state variables set to reflect the state of the compilation. The compiler is passive in the sense that a compilation produces the minimum amount of code necessary to achieve the results required by the state variables.

<u>User Control Facilities</u> -- The user can give the compiler explicit instructions to aid in the compilation process. As in LISP 1.5, macros are an integral part of the language. Many of the facilities of the language, e.g., FOR statements and relational

expressions, are implemented by means of system macros. These expand in terms of highly optimized compiler controlling functions. Thus it is essential to produce good code for a small, selected number of things in terms of which everything else is defined.

Certain machine-dependent operators are particularly useful as primitives in compilation. CORE is an operator that acts like an array whose contents is all of the machine memory. Therefore CORE(x) is the contents of location x. BIT is an operator that specifies a certain contiguous portion of a word. There are also several operators that permit an expression to be forced to a certain type or permit a datum of one type to be used as through it were of another type. Though such mechanisms are in most compilers, LISP 2 has made these items available through the language.

The user may instruct the compiler to create open subroutines for certain LISP functions. Open subroutines are specified by instruction sequences. The user defines the instruction sequence by giving a function that constructs the sequence. This function uses the internal variables of the compiler (which are fluid for just that reason). The input to the function is the operational form that specifies the open subroutine; the output is a sequence of instructions to be included in the compiler output. The instruction sequence for a particular function is inserted whenever an operational form is encountered that has the function as its operator; at this point the compiler invokes the function that generates the instruction sequence. The LISP functions CAR and CDR are implemented in this way, and the code generated for them is no longer than that for a closed subroutine cell.

THE LISP 2 ASSEMBLY PROGRAM

The LISP 2 Assembly Program, LAP, is a program that generates a code segment from a list of symbolic instructions and labels. LAP also allocates storage for variables on the pushdown stack, and insures that references to fluid and own variables are consistent among different compiled functions. LAP does more than most assemblers, in that it handles all aspects of pushdown stack mechanics; consequently, references to variables are made by naming the variable in the appropriate field of any instruction that references it. Thus, the pushdown stack need never be referenced explicitly.

LAP includes a number of system macros specifically designed for LISP 2 programming. The prologue and epilogue of a function are generated by BEGIN and RETURN respectively; CALL is used to generate a call to a LISP 2 function in the standard format. Storage allocation on the pushdown stack is performed by the BLOCK, DECLARE, and END macros; FLBIND creates any necessary bindings for fluid variables. LAP does not have a generalized macro facility; any effect that could be achieved by such a facility, however, can also be achieved by preprocessing.

The actual workings and structure of the pushdown stack could be changed considerably without affecting the LAP input language. For that reason, even the bulk of LAP itself is machine dependent. At any time, LAP is aware of the most recently allocated cell on the pushdown stack. Allocation or release of a pushdown cell is purely a matter of internal LAP housekeeping; it does not cause any extra instructions to be generated. The address field of an instruction may be used to affect pushdown storage allocation. The address fields TOP. and POP. are normally used with load-type instructions. Both TOP. and POP. refer to the most recently allocated pushdown cell, but POP. has the additional effect of releasing that cell. PUSHA. and PUSHP. both cause a new pushdown cell to be allocated, and refer to that cell; PUSHA. and PUSHP. are normally used in store-type instructions. PUSHA. is used for absolute quantities and PUSHP. for symbolic quantities, so that a map of the pushdown stack can be maintained.

Unlike function definitions in either SL or IL, LAP programs are context independent. While function definitions derive much of their declaratory information from the current environment, programs in LAP do not. All such information is included in the LAP program generated by a compilation. Thus, a library of LAP programs can be maintained, and any program in the library can be read in at any time with complete consistency checks on declarations.

LISP 2 PRIMITIVES

The LISP 2 primitives are a set of basic functions, routines, macros, and instruction sequences in terms of which all other LISP 2 programs are written. The primitives are used both by the system itself (since the system is written in terms of them) and by the user. All of the LISP 2 primitives were programmed directly in LISP 2. The primitive CONS, for example, is defined by:

```
SYMBOL SECTION SYS;
FUNCTION CONS $LISP (A, B);
BEGIN SYMBOL S ← CHEAT(INTEGER, SYMBOL, LSP ← LSP -1);
  CORE(CHEAT(SYMBOL,INTEGER,S)) ← CHEAT(SYMBOL,OCTAL,B);
  CAR S ← A;
  IF ARP >= LSP THEN RECLAIM(1); RETURN(S);END
```

where ARP and LSP are system variables containing the upper boundary of array space and the lower boundary of list storage respectively.

CREATION OF LISP 2 SYSTEMS

Bootstrapping from LISP 1.5 to LISP 2 on the Q-32 proceeded in three stages. At the end of the first stage, a LISP 2 system was produced that only accepted input in LAP. At the end of the second stage the system accepted input in LAP and IL, and at the end of the third stage it accepted input in LAP, IL, or SL. A fourth stage is required to obtain LISP 2 on a new machine. The successive stages are illustrated in Figures 2 through 5, and are discussed below. Each of the figures shows a succession of steps in the bootstrap. The notation used in the diagrams will become obvious from the exposition.

Stage 1

A. The compiler, which translates IL into LAP, is written in LISP 1.5.

B. The core image generator, which translates the LAP language into an octal core image, is written in LISP 1.5. The core image is a memory map, in octal, of all code and associated storage structures required by the input to the core image generator. It can be loaded into the Q-32 by a simple octal loader, and produces executable code for the programs that were fed into it.

C. LAP, which translates the LAP language into Q-32 machine language, is written in LISP $\overset{2}{\boxed{1.5.}}$ Most of this program is the same as the core image generator B.

D. The primitives are written in IL. These include the garbage collector, the input-output functions, and the primitives required for creating, testing, and manipulating LISP 2 data.

E. The primitives are combined with LAP to produce a LAP-level system written in IL.

$$\boxed{A} \equiv \text{IL} \xrightarrow{\text{L1.5}} \text{LAP} \qquad \text{(compiler)}$$

$$\boxed{B} \equiv \text{LAP} \xrightarrow{\text{L1.5}} \text{CI} \qquad \text{(core image generator)}$$

$$\boxed{C} \equiv \text{LAP} \xrightarrow{\text{IL}} \text{ML} \qquad \text{(assembler)}$$

$$\boxed{D} \equiv \text{Primitives in IL}$$

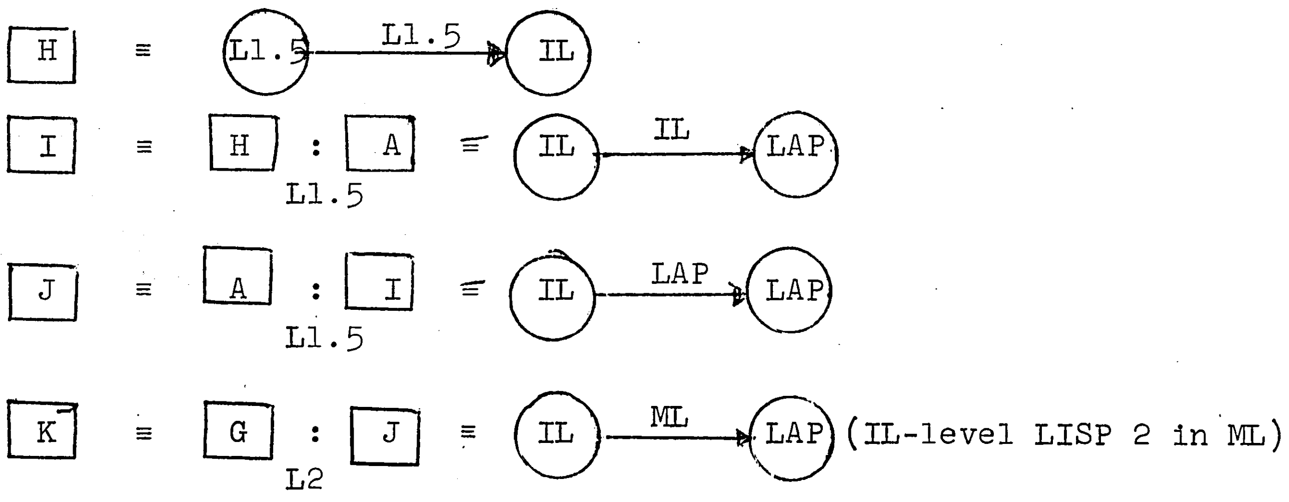
$$\boxed{E} \equiv \boxed{C}\boxed{D} \quad \cup \quad \text{(LAP-level LISP 2 in IL)}$$

$$\boxed{F} \equiv \boxed{A}\boxed{B} \equiv \text{IL} \xrightarrow{\text{L1.5}} \text{CI}$$

$$\boxed{G} \equiv \boxed{F} \underset{\text{L1.5}}{:} \boxed{E} \qquad \text{(LAP-level LISP 2 in ML)}$$

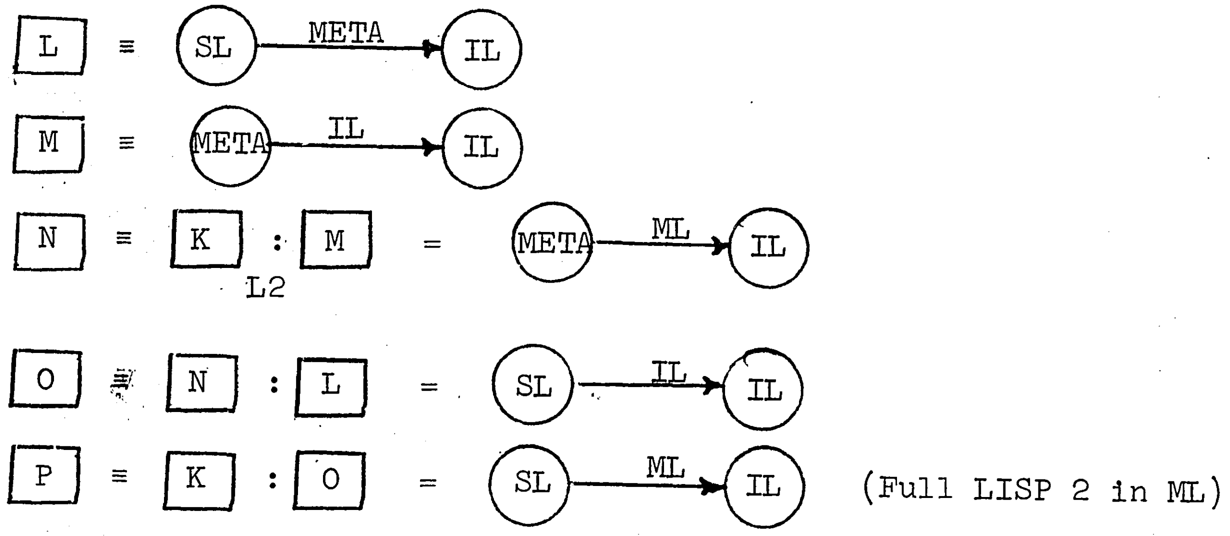
Stage I of LISP 2 Bootstrap

Figure 2



Stage II of LISP 2 Bootstrap

Figure 3



Stage III of LISP 2 Bootstrap

Figure 4

$\boxed{Q} \equiv \mathrm{IL} \xrightarrow{SL} \mathrm{LAPX}$

$\boxed{R} \equiv \mathrm{LAPX} \xrightarrow{SL} \mathrm{MLX}$

$\boxed{S} \equiv \mathrm{LAPX} \xrightarrow{SL} \mathrm{CIX}$

$\boxed{T} \equiv$ Primitives for X in SL

$\boxed{U} \equiv \boxed{Q}\boxed{R}\boxed{T}$

$\boxed{V} \equiv \boxed{P} \underset{L2}{:} \boxed{U}$ (IL-level LISP 2 for X in IL)

$\boxed{W} \equiv \boxed{V}\boxed{M}\boxed{O}$ (Full LISP 2 for X in IL)

$\boxed{X} \equiv \boxed{Q} \underset{L2}{:} \boxed{W}$ (Full LISP 2 for X in LAPX)

$\boxed{Y} \equiv \boxed{S} \underset{L2}{:} \boxed{X}$ (Full LISP 2 for X in CIX

Stage IV of LISP 2 Bootstrap
Figure 5

F. The compiler and the core image generator are combined to produce a LISP 1.5 program that translates IL into a core image.

G. The program F is applied to the program E in order to obtain a core image of a LAP-level LISP 2 system. The operation is performed in the LISP 1.5 environment. This core image is then loaded into the Q-32 in order to obtain the system in working form.

Stage 2

H. A program that translates LISP 1.5 into IL is written in LISP 1.5.

I. The program H is applied to the program A in order to obtain a version of the compiler written in IL. The operation is performed in the LISP 1.5 environment.

J. The program A is applied to the program I in order to obtain a version of the compiler written in LAP. The operation is performed in the LISP 1.5 environment.

K. LAP-level LISP 2 is used in order to assemble the compiler, which was written in LAP as a result of Stage J. This operation and all succeeding operations, are performed in the LISP 2 environment. This operation could also have been performed through core image generation in the LISP 1.5 environment.

Stage 3

L. The syntax translation specifications for translating SL to IL are written in the META language.

M. The META compiler, which produces a syntax translator in IL from a set of syntax translation specifications, is written in IL.

N. The compiler is applied to the META compiler in order to obtain an operating version of the META compiler in machine language.

O. The META compiler is used in order to obtain an IL program that translates SL to IL.

P. The IL program just obtained is compiled, and incorporated into the system. The result is a complete LISP 2 system on the Q-32.

Stage 4

Q. An SL version of the existing compiler is modified so as to produce LAP for machine X (the new machine) rather than for the Q-32.

R. An SL version of LAP for the Q-32 is modified so as to produce machine language for machine X.

S. The SL version of LAP for the Q-32 is also modified so as to produce a core image for machine X.

T. The primitives for machine X are written in SL.

U. The compiler, LAP, and the primitives, now all written in SL, are joined together to produce IL-level LISP 2 for machine X.

V. The program U is translated into IL.

W. META and the SL-to-IL translator are joined with V to produce full LISP 2 for machine X in IL.

X. LISP 2 for machine X is compiled on the Q-32.

Y. LISP 2 for machine X is assembled on the Q-32 to obtain a core image. This core image, when loaded into machine X by an octal loader, produces a complete and working version of LISP 2 on machine X.

REFERENCES

1. Saunders, R.A., "The LISP System for the Q-32 Computer," in The Programming Language LISP, Information International, Inc. Cambridge, Massachusetts, 1964, pp. 220-238.

2. Kelly, H.S., and Newell, A., (ED) Information Processing Language-V Manual (Prentice-Hall, Inc., Englewood Cliffs, N. J. 1964), 2nd ed.

3. Collins, G. E., "REFCO III, A Reference Count List Processing System for the IBM 7094," IBM Research Report RC-1436, (1965).

4. Schorre, D.V., "META II A Syntax-Directed Compiler Writing Language," Proc. ACM p. D1.3-1 (1964).

5. "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM 6, 1 (1963).

6. De Morgan transformation reference