SP-3043

SP *a professional paper*

THE SDC LISP 1.5 SYSTEM FOR IBM 360 COMPUTERS

by

J. A. Barnett and R. E. Long

11 January 1968

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA
90406

(page 2 blank)

ABSTRACT

This paper describes a LISP 1.5 system currently being developed at SDC for IBM 360 computers. This system will provide all the features presently available in the SDC LISP 1.5 system that runs on the Q-32 computer under the SDC Time-Sharing System, and will be compatible with that system. The design of this version of LISP 1.5 is based on SDC's experience in developing both LISP 1.5 and LISP 2 systems, as well as recent improvements in LISP technology.

INTRODUCTION

Since about 1961, a number of list-processing languages and processors have been produced at System Development Corporation. Original work involved such early languages as IPL-V and SLIP, which were developed initially for use in artificial intelligence research [1,2]. In 1963, work was begun on a LISP 1.5 system for the IBM AN/FSQ-32 computer, a large, high-speed machine which is operated principally under the SDC Time-Sharing System. This LISP system [3] has been used extensively since its release in 1964. Other efforts in this area have included development of LISP 2 for the Q-32 computer, and design of a LISP 2 system for the IBM 360 computer [4,5].*

During 1967, development of a LISP 1.5 system for IBM 360 computers was begun at SDC. The target date for completion of this work is April 1968. The system is designed to operate under the ADEPT time-sharing executive, which operates on Models 50 and 67 in the 360 series. Provision has also been made, however, to allow the system to run under OS/360.

The system is compiler-based, and is generally compatible with the LISP 1.5 system that runs on the Q-32 at SDC. No attempt has been made to make the system compatible with interpretive LISP systems. An editing program, called LISPED, has been provided to aid in debugging, program entry, and editing of source-language programs and data [6]. Also, the storage conventions adopted for the system allow a relatively simple addition of a program overlay feature.

The description of the LISP 1.5 system for the 360 presented here includes a discussion of added language features, the operation of the compiler and assembler, the input/output capabilities, and the storage conventions and garbage collection techniques employed.

---

* Much of the work on these earlier LISP systems was sponsored by the Advanced Research Projects Agency.

LANGUAGE

In general, the language accepted by this version of LISP 1.5 is identical to that accepted by the LISP 1.5 processor that runs on the Q-32 [7]. Several additions and modifications have been made to the language, however, to provide greater flexibility and ease of expression. These changes involve two major types: (1) those concerned with naming conventions for variables and changes to the kinds of data structures allowed, and (2) those concerned with added or changed language forms (PROGN, LABEL, BLOCK, SELECTQ, and FOR).

NAMING CONVENTIONS FOR VARIABLES

In this version of LISP 1.5, all names of global entities, i.e., special variables, functions, and macros, are ordered pairs. The first element is an identifier, and the second element is an integer in the range $0 \leq N \leq 127$. The second element is a section name that acts like a qualifier. Global entities may be written as dotted pairs. The sectioning mechanism described here allows for several global entities to share the same identifier name without conflicting.

Two parameters control the naming rules invoked by the compiler and assembler:

1. Guess, a section number, and
2. Guess-list, a list of section numbers.

Any global entity name used in a compiled or assembled form may be written as a dotted pair,* for example,

```
(SETQ (X . 3) ((CAR . Ø) (Z . 2)))
```

In this example, the special variable named X in section 3 is assigned the value CAR of the variable named Z in section 2. All the normal LISP form names (CAR,

*Names written as dotted pairs are referred to as "tailed."

SETQ, QUOTE, etc.) are defined in section Ø. When untailed names are used freely, the following rules are used by the compiler and assembler:

1. If a declaration exists for the name in any section in Guess-list, then use that name as if it had been explicitly tailed.
2. Else, create a declaration in section Guess and use that name as if it had been explicitly tailed.

A variable bound as a PROG, LAMBDA, or BLOCK variable is always bound as a special variable if it is tailed. If an untailed variable to be bound is declared special in section Guess, then the variable is specially bound as if it had been explicitly tailed in section Guess. Note that if section Guess is Ø and Guess-list is (Ø), the compiler and assembler behave similarly to the Q-32 LISP 1.5 system.

DATA STRUCTURES

In addition to the standard data structures provided by the Q-32 LISP system, this version of LISP 1.5 provides strings and one-dimensional arrays as legal data structures. Arrays and strings do not have names; rather they are autonomous data structures as are list nodes, numbers, etc. Strings and arrays are processed by subscripting.

In order to perform subscripting, two subscript functions are used. The first retrieves the value of an element of an array, or a character from a string. The second assigns a value to an element of an array or places a character in a string. An array element may be assigned the value of any S-expression, including an array.

A syntax has been devised for reading and writing strings, arrays, functionals, and numerics with radices of 8, 10 and 16.

The system does not provide for character objects (as does the Q-32 LISP 1.5 system). Also, no use is made of property lists by the system. The property list belongs entirely to the user and may be any S-expression. (The system keeps a separate property list for itself.)

LANGUAGE FORMS

PROGN Form

A form called PROGN has been added to the language. Used as an expression, PROGN has as its body an indefinite number of expressions, each evaluated in a left-to-right order. The value of the last expression is the value of the PROGN form. If the body is empty, (PROGN), the value NIL is produced.

Used as a statement, PROGN has as its body an indefinite number of statements. The statements are operated in a left-to-right order. After the operation of the last statement, control falls through the PROGN form. Any of the embedded statements may either be conditional transfers, unconditional transfers, or return statements. The transfer statements may reference labels outside of the PROGN form.

Several forms in the language, COND, SELECT, SELECTQ, LAMBDA, RETURN, and SETQ, use an implicit PROGN. This is best explained with an example:

```
(COND (P1) (P2 X) (P3 (SETQ X Y) Z))
```

In this example, if P1 is true, the value of the COND is NIL; if P2 is true, the value of the COND is X; if P3 is true, X is assigned the value of Y and the value of the COND is Z. This is equivalent to the following expression:

```
(COND (P1 (PROGN))(P2 (PROGN X))
      (P3 (PROGN (SETQ X Y) Z)))
```

Similarly, the expression

```
(SETQ X (G Y) (CDR X))
```

is equivalent to

```
(SETQ X (PROGN (G Y) (CDR X)))
```

LABEL Form

A change has been made to the meaning of the LABEL form. Rather than being used to provide a local name for embedded LAMBDA forms, it is used to provide a means of attaching a statement label at points at which it would otherwise be impossible. The LABEL form contains two items, the label and a statement, for example:

```
L1 (COND(P1  (G X)) (P2 (LABEL L2 (H Y)))(T (I Z)))
```

In this example, the label L2 is placed in front of the statement (H Y). Thus, any statement that could legally transfer to label L1 can transfer to label L2.

The LABEL form may also embed any statement in a PROGN statement; the label placed there will be visible anywhere a label would be visible if it were embedding the PROGN itself. When the LABEL form attaches a label to a part of either a COND, SELECTQ or SELECT statement, and control is transferred to that label, operation will proceed with the item so labeled as if control had been placed there on a predicate match.

BLOCK Form

A BLOCK form has been added that combines the sequence of statement features of PROG's with the ability of LAMBDA's to preset bound variables to values other than NIL. Except for the syntax of the bound variable list, the syntax of a BLOCK is identical to that of a PROG.

In a BLOCK, the variable list is a mixture of any number of variables and lists whose CAR is a variable and whose CADR is any expression. In the variable case, the preset is NIL; in the list case, the preset is the value of the expression. For example,

```
(BLOCK  (A  (B  (CAR C))  D)
   L(COND ((NULL B) (RETURN A))
          ((ATOM (SETQ D (CDR B))) (RETURN D)))
    (SETQ B D)
    (SETQ A (CONS D A))
    (GO L))
```

is equivalent to

```
((LAMBDA (A  B  D)
    (PROG ()
      L(COND((NULL B)(RETURN A))
            ((ATOM(SETQ D (CDR B)))(RETURN D)))
       (SETQ B D)
       (SETQ A (CONS D A))
       (GO L)))
  NIL (CAR C) NIL)
```

In fact, the meaning of RETURN, the visibility rules for labels, and the order of evaluation of presets are precisely described by the transformation of a BLOCK into a combination of PROG and LAMBDA [8].

SELECTQ Form

A form very similar to SELECT has been added, called SELECTQ. In SELECTQ, the select-expression is evaluated and matched to the unevaluated selectors. The selectors are either identifiers (literal atoms) or lists of identifiers. If the select-expression is EQ to either the identifier or any member of a list of identifiers, the implied PROGN to the right is operated in the same manner as in SELECT. For example,

```
(SELECTQ (READ)
    ((A B) (CAR X))
    ( C    (CDR X))
    (QUOTE OTHER))
```

is equivalent to

```
(SELECT (READ)
    ((QUOTE A) (CAR X))
    ((QUOTE B) (CAR X))
    ((QUOTE C) (CDR X))
    (QUOTE OTHER))
```

The implied PROGN's are either expressions or statements, depending on whether the SELECTQ is an expression or statement. A LABEL form embodying an implied PROGN is visible from any point that a label in front of the SELECT or SELECTQ would be visible.

FOR Form

A multi-generator parallel loop FOR macro has been added to the system as part of the language. Generators are STEP, IN, and ON. STEP is a numerical incrementor or decrementor; IN moves through the top-level elements of a list; and ON moves through successive CDR's of a list. The control elements of the FOR loop are UNTIL, WHILE, UNLESS and WHEN. UNTIL and WHILE can cause evaluation of the whole loop to cease, whereas UNLESS and WHEN only cause a particular iteration to be skipped.

The FOR macro expands as a BLOCK, and therefore has a value. Embedded RETURN statements will cause the value of the FOR loop to be the value of the RETURN's expression body. On fall-through cases, the value may be given by the VALUE form.

COMPILER

The compiler designed for this system is a one-pass program coded in LISP 1.5. Its structure is based on the LISP 2 compiler designed at SDC for IBM 360 computers [5,9,10]. The input is S-expression LISP 1.5 and the output is LAP 360. A front-end syntax-checking pass is optional. The main compiler assumes that no syntatic errors are present, and only diagnoses misuses of variable names and contextual errors, e.g., a GO statement being used as an expression would cause an error diagnostic. Both warning and error messages are issued by the compiler. Error messages inhibit the LAP 360 output from being assembled; warning messages do not.

Several kinds of optimizations are performed by the compiler to reduce the amount of output code. For example, the contents of the accumulator are remembered from one form to another. This tends to minimize the number of load instructions output by the compiler.

Other optimizations performed by the compiler are concerned with branching instructions. The scheme used causes the compilation to be divided into five modes:

1. Compilation of expressions for value.
2. Compilation of expressions with a label to transfer to with the value in the accumulator.
3. Compilation of statements.
4. Compilation of statements with a label to transfer to if the statement does not unconditionally transfer on its own.
5. Compilation of predicates: this is similar to expression compilation with two labels--one to transfer to on NIL values and the other to transfer to on non-NIL values.

This kind of optimization is achieved by having each form cause the compilation of its arguments in modes which depend on the mode in which the form itself is being compiled.

The statement, expression, and predicate labels are maintained by the compiler, along with their respective reference counts. When these labels are attached to the LAP 360 listing, several checks are made. If the reference count is Ø, it is not attached and therefore the contents of the accumulator are preserved. Also, if the last instructions on the listing are branches and one of them is to the label being attached, then at least one of those instructions is deleted and the remaining ones rewritten to produce an equivalent operation.

ASSEMBLER

The assembler, LAP, is a one-pass processor that handles programs up to a maximum size of 4,096 bytes in length. This allows the same base register to be used for all branching instructions.

The assembly language for the system, LAP 360, is a symbolic assembler language that will handle the full instruction set of the IBM 360 computer and provide for a macro capability. Variables in LAP 360 are referenced and bound by their symbolic names. The assembler automatically binds special variables specially and follows the same defaulting conventions as does the compiler. A block structure facility is provided to ease the burden of compilation of PROG, BLOCK, and LAMBDA forms.

The instruction assembly is done by a pattern-matching routine. Each 360 opcode is related to a pattern; the occurrence of register, mask, count or address fields as part of an instruction is specified by the pattern. When a match is found, the numerical equivalent of that field is planted in the binary program image.

The PUSH., POP., ARGS, CALL, FASTCALL, SLOWCALL, CALI, and BLOCK pseudo-opcodes are provided as macros in the basic system. FASTCALL, SLOWCALL, and CALI are used--respectively--for functional calls without error checks, functional calls

with error checks, and calls to functions with an indefinite number of arguments.

Temporary cells on the pushdown stack may be referenced by using the PUSH., POP., and TOP. address mnemonics. The PUSH. and POP. pseudo-opcodes may also be used to control stack allocation. The four calling macros automatically push and pop the stack when the call is made. The ARGS pseudo-opcode is used to help align the stack after calling.

INPUT/OUTPUT

The input/output capabilities of the LISP 1.5 system for the 360 resemble those offered by the LISP 1.5 system that runs on the Q-32 at SDC [11]. The devices handled are tape, disc, and teletypewriter. Further extensions are planned for core files and CRT displays.

Input/output under LISP 1.5 is file-oriented. A named file may be opened or shut. This means that the LISP system acquires files from the operating system, and returns files to the operating system. After a file has been opened and before it is shut, it may be selected for either input or output. Only one file may be selected for input and one file selected for output simultaneously. All reading and printing functions implicitly use the files that are selected for input and output.

The available input functions are READ (reads an S-expression), RATOM (reads either an atom or a delimiter), and READCH (reads a character). The available output functions are PRINT (prints an S-expression), and PRINCH (prints a character). The PRINT operation is controlled by the value of two Boolean-valued variables. The first determines whether symmetric printing or normal printing is done; the second controls whether "pretty" formatted printing or packed formatted printing is done. A POSITION function is available for rewinding tapes, skipping files, etc.

The READ function can input constant arrays, strings, and functionals in addition to unusually spelled identifiers with the $$ artifact. The upper limit on length of strings and identifiers read is 255 characters.

STORAGE CONVENTIONS AND GARBAGE COLLECTION

One of the most important and innovative parts of the LISP 1.5 system for the 360 is that concerned with methods for storage management. The storage conventions and garbage collection techniques employed were originally intended for use in the LISP 2 system designed at SDC for the 360 [4]. The major storage management features provided by the system are listed below:

1. 16-bit CAR/CDR; hence, at most 65,536 words of LISP-addressable data structures
2. Small integers (at least 4,095 positive and 4,096 negative)
3. 32-bit two's complement integers and hexadecimal numbers
4. Double-precision (64-bit floating-point) numbers
5. Dynamic arrays and strings
6. Relocatable binary programs
7. 4,096-byte maximum binary program size
8. Push-down stack packed with no holes on it
9. Push-down stack constructed to allow for easy scanning for previous values of special variables and for finding information about active return addresses on the stack; in cases of stack unwrapping or error set, the stack may be unwrapped without consulting binary program images; allows for easier implementation of a swapping mechanism
10. Functionals may be the values of special variables or parts of list structure (not so on the Q-32)
11. Unique quoted structures
12. Identifiers reclaimed and folded
13. Identifiers with 1-character print names fixed in the system and never garbage collected or relocated (also not distinguishable from other identifiers)

14. Property list reserved for the user
15. Separate system property list

A five-phase garbage collector is used by the system. Each phase has responsibility for a different garbage-collector function. The phases and their functions are described below:

1. Mark phase: All active data structures are marked. A 2,048-word bit table is maintained by the system. Each bit of the table corresponds to one 32-bit word in LISP-addressable space. Only one word of a multi-word structure needs to be marked to make the whole structure active.
2. Prune phase: The oblist is pruned and various quote and variable structures no longer needed are unlisted from system property lists.
3. Fold and Plan phase: Structures in node, identifier, and number spaces are folded. Relocation is planned for the resident structures of other spaces.
4. Update phase: Each pointer quantity in every active structure is changed to the new address of the object pointed at (as determined by the Fold and Plan phase).
5. Move phase: Arrays, binary programs, and any other structures not folded (for which moving is necessary) are moved to the locations determined during the Fold and Plan phase.

Each of the five phases of the garbage collector detailed above is table-driven, that is, for each kind of data space their may exist up to five functions corresponding to any or all of the five phases of garbage collection. Thus during phase one, all phase one functions are called; during phase two, all phase two functions are called; etc. The garbage collector, then, is merely a set of five loops that call the functions for each space in sequence.

Memory is divided into quanta of 256 words. Each data space is comprised of an integral number of quanta. A table, called the quantized core map (QCM), contains one byte for each quantum of LISP-addressable memory. The byte corresponding to a particular quantum contains information describing the kind of space of which the particular quantum is a part. Predicates such as NUMBERP, ATOM, etc., use their pointer arguments to look up the description in the QCM, rather than doing boundary checks. The Mark and Update phases of the garbage collector also use the QCM.

A function called UMARK (the universal marker) is used by phase one functions to mark active structures. UMARK is given one argument, a pointer to any structure in LISP-addressable memory. UMARK marks this structure active, then looks up the byte in the QCM corresponding to the quantum in which the structure pointed at resides. This byte contains a number between 0 and 32; it is used as an index to a function table, called the universal mark function table, UMFT. The elements of UMFT then continue the marking process in a manner which depends upon the space with which they are associated. For example, the particular function in UMFT associated with list nodes, namely MARKNODE, could well have the following definition:

```
(MARKNODE(P)
    (PROG()
       (UMARK (CAR P))
       (UMARK (CDR P))))
```

where P is the pointer given to UMARK, and subsequently passed to MARKNODE.

The universal updater function, UPDATE, works with a table of functions to update addresses in a manner similar to the way UMARK works to mark active data structures. For example, to update a node, N, the following statements would suffice:

```
(RPLACA N (UPDATE (CAR N)))
(RPLACD N (UPDATE (CDR N)))
```

In each case, UPDATE would call the function appropriate for finding the new address for CAR and CDR of N as determined by the Fold and Plan phase.

A point of interest about the storage convention is that the pushdown stack, binary programs, and the bit table need not be in LISP-addressable space. If they are, these structures may reside in space that is pointed to as "small-integer" memory. This flexibility allows the actual amount of memory comprising the system to range between 20- and 90-thousand 32-bit words.

The quantized core map used in conjunction with table-driven garbage collection also allows for two important extensions to the system: First, a "growing-pain" program may be added. This program reallocates the number of blank quanta (256-word blocks) that are available for the different spaces after a garbage collection has been performed. The second extension is a capability to experiment with new data structures. This is achieved by adding new functions to the garbage collector tables and operating the growing-pain program to provide blank quanta for the new space. This may be accomplished without changing those parts of the garbage collector that deal with old spaces.

CONCLUSION

The LISP 1.5 system described in this paper is currently under development at SDC, and is expected to be complete in April 1968. The major design goals of the system are to improve on the features provided by the SDC Q-32 LISP 1.5 system, while maintaining compatibility with it, and to produce a system that is modular and open-ended so that new features can be tested and added at a later date.

The major language innovations to the system involve changes in the naming conventions for variables, changes in the kinds of data structures allowed, and added or changed language forms. The major features provided by the processor portion of the system involve several compiler optimizations that reduce the amount of output code and allow more economical branching operations, the use of pattern-matching techniques in the assembler, and the use of table-driven storage management techniques.

REFERENCES

1. Shaffer, S. S., "Current Status of IPL-V for the Philco 2000 Computer". Communications of the ACM, September 1962, Vol. 5, No. 9, p. 479.
2. Londe, D. L., "An Introduction to SLIP (Symmetric List Processor)". SDC document TM-2399, May 13, 1965, 29 pp.
3. Saunders, R. A., "The LISP System for the Q-32 Computer". In Berkeley, E.C. and Bobrow, D. G. (Eds.), The Programming Language LISP: Its Operation and Applications, Cambridge, Massachusetts: The MIT Press, 1966.
4. Abrahams, P. W., Barnett, J. A., et al., "The LISP 2 Programming Language and System". AFIPS Conference Proceedings, Fall Joint Computer Conference, 1966, Vol. 29, pp. 661-676.
5. Barnett, J. A., Long, R. E., et al., "LISP 2 for the IBM S/360". SDC document series TM-3417, April 26, 1967, 15 volumes.
6. Kameny, S. L. and Hawkinson, L., "LISP Edit Program LISPED". SDC document TM-2337/100/01, April 13, 1966, 19 pp.
7. Weissman, C., LISP 1.5 Primer. Belmont, California: Dickenson Publishing Company, Inc., 1967.
8. Berkeley, E. C. (Ed), "Thesaurus and Dictionary of Functions in LISP" (in press).
9. Saunders, R. A., Barnett, J. A., and Firth, D. C., "The LISP 2 Compiler". SDC document TM-2710/320/01, February 1, 1966, 55 pp.
10. Barnett, J. A., "Algorithmic Compilation of Predicates". SDC document SP-2856 (in press), 12 pp.
11. Kameny, S. L. and Weissman, C., "The Q-32 LISP 1.5 Mod. 2.6 System: Operating System, Input/Output, File, and Library Functions". SDC document TM-2337/103/00, April 11, 1966, 27 pp.