October 1980

CS-TR-80-18

LITTLE BIG LISP

by

Jed B. Marti

Department of Comp. and Inf. Science
The University of Oregon
Eugene, Oregon 97403

ABSTRACT

This manual describes the Little Big LISP system for the Z80 microcomputer. The manual describes data structures, defined functions, operating procedures, a compiler, an RLISP parser, and support packages.

INTRODUCTION

Little Big LISP is a subset of Standard LISP [1] implemented for the Z80 microprocessor. It runs in a minimum of sixteen thousand bytes of storage and most effectively with thirty two thousand or more. The system consists of the following:

1. An interpreter

2. A program to load precompiled object files ("fast load" files)

3. A compiler for generating either fast load files or directly executable code

4. An arbitrary precision integer arithmetic package [2]

5. A parser for a subset of RLISP [3]

This manual is divided into five sections, the first describing data structures, the second the functions of the base system, the third the operating system interface. The fourth section describes the compiler. The fifth section describes the RLISP parser and gives examples of its use. Separate manuals describe the use of the system with the various operating systems that support it.

CHAPTER 1

DATA TYPES

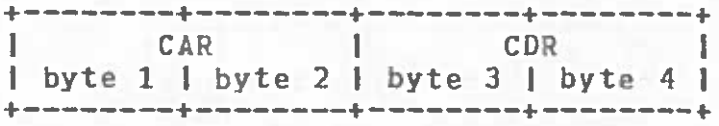
1.1 ITEMS

An item is a 16 bit quantity. The last 12 or 13 bits constitute the data portion of the value and the first 3 or 4 bits, its tag, indicating type and current accessability from the base system.

Bit	Use
0	Used by garbage collector to indicate item is in use.
1-2	Data type: 00 - Dotted-pairs. 01 - Identifiers. 10 - Integers. 11 - Strings and function pointers.
3	Subtype bit for strings and function pointers.

1.2 DOTTED-PAIRS

Up to 8192 dotted-pairs (32k bytes) may be referenced by the little big LISP system depending on the amount of available storage. A minimum of 300 pairs are required for the base system to operate. To address a full 8k pairs requires that the data portion of a dotted-pair pointer be an index into the "vector" of dotted-pairs. Dotted-pairs are two contiguous items, four bytes arranged in ascending storage order:



To compute the real address of a dotted-pair from its item pointer, the value portion of the item is shifted left two bits and the resulting value is added to the base address of the pair space.

Dotted-pairs are entered and printed in the same form as Standard LISP. The list representation of dotted-pairs is permitted as well as the use of ' to represent the QUOTE function.

1.3 IDENTIFIERS

Identifiers are the same as those defined in Standard LISP except that all identifiers are interned and may not be removed from the object list (the symbol table in this case). The system requires a minimum of 160 identifiers to operate and may reference up to eight thousand of them.

Identifiers have 1 to 255 character print names. The first character must be alphabetic or any other character preceeded by the ! escape character. Following characters may be alphanumeric or other characters prefixed by the escape character. There is no !*RAISE flag, lower case characters are not converted to upper case.

Each identifier is two items in the symbol table, the first being a pointer to the string by which the identifier is known to the outside world called the print name. The second is a pointer to values associated with the identifier called the property list. The symbol table is a vector of these pairs.

The property list is implemented as a list structure with the following attributes:

1. An atom is a flag (see the FLAG, FLAGP, and REMFLAG functions)

2. A dotted-pair is an indicator-value pair (see the GET, PUT, and REMPROP functions). There are three special pairs for global values and functions, these being (GLOBAL . xxx), (EXPR . xxx), and (FEXPR . xxx)

Thus the function REVERSE, a compiled EXPR has as its symbol table entry (note that $6003 is a hexadecimal quantity described later):

"REVERSE" (print name) ((EXPR . $6003))

1.4 INTEGERS

Integers are stored as 13 bit two's complement values. They conform to the Standard LISP conventions for fixed numbers in the range -4096 to +4095.

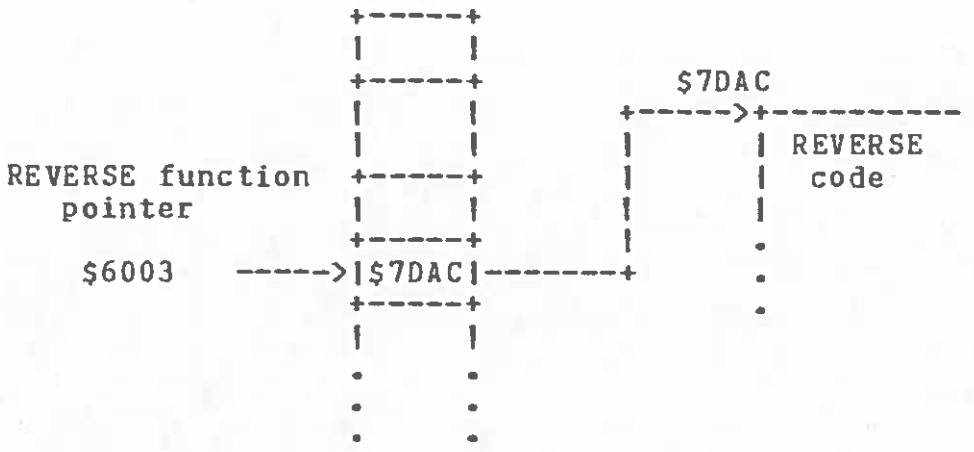
1.5 STRINGS

Strings are arbitrary character sequences from 0 to 255 characters in length. Strings serve as print names for identifiers or as constants. A string pointer is a 12 bit offset into the string space which is a single large character vector. The minimal system requires a few more than 1200 bytes of string space. Each string is a byte containing the number of characters in the string followed by that number of characters. Thus the string "REVERSE":

Strings are entered surrounded by "'s. Unlike Standard LISP, "'s are not allowed within the string.

1.6 FUNCTION POINTERS

Since compiled functions may occur almost anywhere in storage and thus their addresses look like an arbitrary item, real addresses of functions are hidden in the real address table. A compiled or primitive function is normally addressed indirectly through this table.



Function pointers may not be read in but are displayed as 4 hexadecimal digits preceded by a dollar sign. The number in the table may not be accessed except internally.

1.7 STACKS

There are two internal stacks. One contains stack frames, displays for local variables in compiled functions. The other contains a pushdown stack for return addresses and intermediate values. The stack frames are in ascending storage order and the pushdown stack descends. When they cross or are about to cross the system stops.

To assure that only valid items on the stack for the garbage collector forces the following requirements:

1. All values less than 8192 ($1FFF) are pointers to dotted-pairs.

2. All greater than 8192 are atomic. Thus, the first 8k of storage must not have routines which will have return addresses on the stack when the garbage collector might be called.

We have made this possible by putting dotted-pair space and stacks in the low 8k of the system. Since functions are stored above the 8k boundary, their return addresses look like constants and are not examined by the garbage collector.

CHAPTER 2

FUNCTIONS

The functions that follow are presented in the format of the Standard LISP Report [1]. Except for the low level and compiler support functions the function descriptions have been copied from the report.

2.1 LOW LEVEL FUNCTIONS

The following functions are accessible by the user but are not part of Standard LISP.

!$PA(x:integer)
Type: EVAL, SPREAD.
Using the last 8 bits of the integer X, print these bits as an ASCII character.

!$GA():integer
Type: EVAL, SPREAD.
Read the next character from the input file and return its character value as an integer from 0 to 255.

GETP!$(X:id):any
Type: EVAL, SPREAD.
Return the property list for the identifier X. X is not type checked for being an identifier.

PUTP!$(x:id,PROP:any)
Type: EVAL,SPREAD.
Replace the property list of the identifier X with PROP. X is not type checked for being an identifier.

CATCH(X:any):any
Type: EVAL, SPREAD.
Evaluate the argument X (X is preevaluated because CATCH is an EXPR) and return this value. If a THROW occurs during this second evaluation, return the value of the argument of THROW.

THROW(X:any)
Type: EVAL, SPREAD.
Cause a jump back to the most current CATCH restoring stack pointers and the like to the environment of the CATCH. The value returned by CATCH is the value of the actual parameter X. A THROW which is not in the scope of a CATCH is caught by the Standard LISP reader.

NCONS(X:any):dotted-pair
Type: EVAL, SPREAD.
Returns (X . NIL).

XCONS(A:any,B:any):dotted-pair
Tupte: EVAL, SPREAD.
Returns the dotted-pair (B . A).

RECLAIM():NIL
Type: EVAL, SPREAD.
Forces a garbage collection.

NTOK():atom
Type: EVAL, SPREAD.
The NTOK function reads the next token from the input stream and generally returns it. The token (if any) is stored in the global variable TOK!* and its type (an integer) in the variable TYPE!*.

TYPE!*	TOK!*	Meaning
0	nnn	Integer
1	id	Identifier
2	*	(
3	*	.
4	*	)
5	string	String
6	id	Single character converted to identifier

(* means "has no defined value")

ORDERP(A:any,B:any):boolean
Type: EVAL, SPREAD.

A 16 bit comparison of the values of A and B are made. This includes the tag fields. ORDERP returns T if A is less than B in the range 0 to 65535. The function is useful for determining the order of items within a space.

2.2 COMPILER SUPPORT FUNCTIONS

The following functions are used by the compiler to create absolute code or by the fast load program to load files.

BPUT(X:integer)
Type: EVAL, SPREAD.
The last 8 bits of the integer X are stored at the location in the global function pointer BPTR and the value in BPTR is incremented by 1.

CPLUS(X:integer):word
Type: EVAL, SPREAD.
Add the 12 bit sign extended value in X to the current value in the global function pointer BPTR and return this 16 bit value which must not be placed anywhere but in binary program space. CPLUS is used to create absolute jump addresses within a function.

LEFT(X:integer):integer
Type: EVAL, SPREAD.
Return the leftmost 8 bits of X as a positive integer 0 to 255.

MKCODE():function-pointer
Type: EVAL, SPREAD.
Create a new function pointer and store the real address in the function pointer BPTR in the real address table for the new function pointer. This function is used to enter a compiled function in the real address table.

MKGLOB(X:dotted-pair):list
Type: EVAL, SPREAD.
X is the dotted-pair (GLOBAL . xxx). Create a list of the address of xxx as two integers 0 to 255 which are the two bytes in reversed order of xxx.

MKREF(X:any):list
Type: EVAL, SPREAD.

This function is the same as MKGLOB except that X can be any object. If X is a dotted-pair (or list), it is added to the global uninterned variable MLIST so that it will not be removed by the garbage collector. MKREF is used by the compiler to generate the addresses of quoted items.

RIGHT(X:any):integer
Type: EVAL, SPREAD.
Return the rightmost 8 bits of what ever value X is as an unsigned positive integer in the range 0 to 255.

WPUT(X:any)
Type: EVAL, SPREAD.
Same as BPUT except that the two bytes of X are placed in reverse storage order.

2.3 ELEMENTARY PREDICATES

Functions return T when the condition defined is met and NIL when it is not.

ATOM(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is not a dotted-pair.

CODEP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a function pointer.

CONSTANTP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a constant (a number, string, or function pointer).

EQ(U:any,V:any):boolean
Type: EVAL, SPREAD.
Returns T if U points to the same object as V. Unlike Standard LISP, fixed integers (not BIGNUM's) are EQ if they have the same value.

EQN(U:any,V:any):boolean
Type: EVAL, SPREAD.
Returns T if U and V are EQ. In Little Big LISP, EQ and

EQN are the same.

EQUAL(U:any,V:any):boolean
Type: EVAL, SPREAD.
Returns T if U and V are the same. Dotted-pairs are compared recursively to the bottom levels of their trees. All atoms must be EQ (EQN is the same as EQ).

FIXP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is an integer (a fixed number).

IDP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is an identifier.

MINUSP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a number and less than 0. If U is not a number or is a positive number, NIL is returned.

NULL(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is NIL.

NUMBERP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a number. In Little Big LISP, NUMBERP is the same as FIXP.

ONEP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a number and EQ to 1. Returns NIL otherwise.

PAIRP(U:any):boolean
Returns T if U is a dotted-pair, else returns NIL.

STRINGP(U:any):boolean
Returns T if U is a string pointer otherwise returns NIL.

ZEROP(U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a number and has the value 0, returns NIL otherwise.

The following Standard LISP elementary predicates are not defined:

FLOATP VECTORP

2.4 FUNCTIONS ON DOTTED-PAIRS

The following are elementary functions on dotted-pairs. All functions in this section which require dotted-pairs as parameters detect a type mismatch error if the actual parameter is not a dotted-pair.

CAR(U:dotted-pair):any
Type: EVAL, SPREAD.
CAR(CONS a b) ==> a. The left part of U is returned. The type mismatch error occurs if the actual parameter is not a dotted-pair.

CDR(U:dotted-pair):any
CDR(CONS a b) ==> b. The right part of U is returned. The type mismatch error occurs if U is not a dotted-pair.

Unlike Standard LISP, the composites of CAR and CDR are supported only to three levels.

CAAAR	CAAR	CAR
CAADR	CADR	CDR
CADAR	CDAR	
CADDR	CDDR	
CDAAR		
CDADR		
CDDAR		
CDDDR		

CONS(U:any,V:any):dotted-pair
Type: EVAL, SPREAD.
Returns a dotted-pair which is not EQ to anything except itself and has U has its left (CAR) part and V as its right (CDR) part.

LIST([U:any]):list
Type: NOEVAL, NOSPREAD.
A list of the evaluation of each element of U is returned.

RPLACA(U:dotted-pair,V:any):dotted-pair
Type: EVAL, SPREAD.
The CAR portion of the dotted-pair U is replaced by V. If the dotted-pair U is (a . b) then (B . b) is returned. The type mismatch error occurs if U is not a dotted-pair.

RPLACD(U:dotted-pair,V:any):dotted-pair
Type: EVAL, SPREAD.
The CDR portion of the dotted-pair U is replaced by V. If dotted-pair U is (a . b) then (a . V) is returned. The type mismatch error occurs if U is not a dotted-pair.

2.5 IDENTIFIERS

All identifiers in Little Big LISP are interned as are all GENSYM's.

GENSYM():id
Creates an identifier which is the characters Gxxxx where xxxx is a hexadecimal number which is incremented each time GENSYM is called. The symbol generated is not guaranteed to be unique.

The following Standard LISP functions are not implemented in Little Big LISP.

COMPRESS EXPLODE INTERN REMOB

2.6 PROPERTY LIST FUNCTIONS

With each id in the system is a "property list", a list of items which are associated with the identifier for fast access. These entities are called "flags" if their use gives the id a single valued property and "properties" if the id is to have a multivalued attribute: an indicator with a property. In Little Big LISP, indicator-value pairs are dotted-pairs, and flags are atoms.

Flags and indicators may clash, consequently case should be take to avoid occurrences of indicators which have the same name as a flag. Likewise, the implementation of functions and globals requires that the indicators or flags EXPR, GLOBAL, and FEXPR not be used.

FLAG(U:id-list,V:id):NIL
Type: EVAL, SPREAD.
U is a list of ids which are flagged with V. The effect of FLAG is that FLAGP will have the value T for those ids of U which were flagged.

FLAGP(U:id):boolean
Type: EVAL, SPREAD.
Returns T if U has been previously flagged with V, else NIL.

GET(U:any,IND:any):any
Type: EVAL, SPREAD.
Returns the property associated with the indicator IND from the property list of U. If U does not have the indicator IND, NIL is returned.

PUT(U:id,IND:id,PROP:any):any
Type: EVAL, SPREAD.
The indicator IND with the property PROP is placed on the property list of the identifier U.

Standard LISP functions which are not implemented:

REMFLAG REMPROP

2.7 FUNCTION DEFINITION

Functions in Little Big LISP are global entities which are stored on the property list of the (EXPR . xxx) or (FEXPR . xxx) pair.

DE(FNAME:id,PARAMS:id-list,FN:any):id
Type: NOEVAL, NOSPREAD.
The function FN with the formal parameter list PARAMS is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is a LAMBDA expression unless the I*COMP

variable is T in which case the EXPR is compiled. The name of the defined function is returned.

DE(FNAME:id,PARAM:id-list,FN:any):id
Type: NOEVAL, NOSPREAD.
The function FN with formal parameter PARAM is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is a lambda expression unless the !*COMP variable is T in which case the FEXPR is compiled. The name of the defined function is returned.

GETD(FNAME:any):{NIL,dotted-pair}
Type: EVAL, SPREAD.
If FNAME is not the name of a defined function NIL is returned. If FNAME is a defined function then the dotted-pair:

(TYPE:ftype . DEF:{function-pointer,lambda})

is returned.

PUTD(FNAME:id,TYPE:ftype,BODY:function):id
Type: EVAL, SPREAD. Creates a function with name FNAME and definition BODY of type TYPE. If PUTD succeeds the name of the defined function is returned. The effect of PUTD is that GETD will return a dotted-pair with the functions type type and definition. Unlike Standard LISP, Little Big LISP does not have GLOBALP returning T for functions.

If the function FNAME has already been defined, a warning message will appear:

(FNAME redefined)

The function defined by PUTD will be compiled before definition if the !*COMP variable is non-NIL.

Little Big LISP does not support the MACRO function type. The following Standard LISP functions are not defined in Little Big LISP:

DM REMD

2.8 VARIABLES AND BINDINGS

A variable is a place holder for an item which is said to be bound to the variable. The scope of a variable is the range over which the variable has a defined value. Little Big LISP supports three binding mechanisms.

Local Binding
: This type of binding occurs only in compiled functions. Local variables occur as formal parameters in lambda expressions and as PROG form variables. The binding occurs when a lambda expression is evaluated or when a PROG form is executed. The scope of a local variable is the body of the function in which it is defined.

GLOBAL binding
: Only one binding of a global variable exists at any time allowing direct access to the value bound to the variable. The scope of a global variable is universal. Variables declared GLOBAL must not appear as parameters in lambda expressions or as PROG form variables. A variable must be declared GLOBAL prior to its use as a global variable.

ALIST Binding
: Little Big LISP does not support compiled FLUID variables as does Standard LISP. However all interpreted functions bind local variables on an association list permitting fluid style access for interpreted functions only.

GLOBAL(IDLIST:id-list):NIL
: Type: EVAL, SPREAD.
The identifiers of IDLIST are declared global type variables. If an identifier has not been declared previously it is initialized to NIL. Identifiers already declared GLOBAL are ignored.

GLOBALP(U:any):boolean
: Type: EVAL, SPREAD.
If U has been declared GLOBAL T is returned, else NIL is returned.

SET(EXP:id,VALUE:any):any
: Type: EVAL, SPREAD.
EXP must be an identifier or an error occurs. The effect of SET is replacement of the item bound to the identifier by VALUE. If the identifier is not a local variable or has not been declared GLOBAL an error occurs. The other Standard LISP error checking is not performed.

SETQ(VARIABLE:id,VALUE:any):any
Type: NOEVAL, NOSPREAD.
SETQ has the same effect as SET except that the first argument is a variable and is not evaluated. The same errors occur.

The following Standard LISP functions are not implemented:

FLUID FLUIDP UNFLUID

2.9 PROGRAM FEATURE FUNCTIONS

These functions provide for explicit control sequencing, and the definition of blocks altering the scope of local variables.

GO(LABEL:id)
Type: NOEVAL, NOSPREAD.
GO alters the normal flow of control within a PROG function. The next statement of a PROG function to be evaluated is immediately preceded by LABEL. A GO may only appear in the following situations:

1) At the top level of a PROG referencing a label which also appears at the top level of the same PROG

2a) As the consequent of a COND item of a COND appearing on the top level of a PROG
2b) As the consequent of a COND item which appears as the consequent of a COND item to any level

3a) As the last statement of a PROGN which appears at the top level of a PROG or in a PROGN appearing in the consequent of a COND to any level subject to the restrictions of 2a,b
3b) As the last statement of a PROGN within a PROGN or as the consequent of a COND in a PROGN to any level subject to the restrictions of 2a,b and 3a

If LABEL does not appear at the top level of the PROG in which the GO appears, an error occurs:

***** LABEL is not a known label

PROG(VARS:id-list, [PROGRAM:<{id, any<}]):any
Type: NOEVAL, NOSPREAD.
VARS is a list of ids which are considered fluid when the PROG is interpreted and local when compiled (see the

"Variables and Bindings" section). The PROGs variables are allocated space when the PROG form is invoked and are deallocated when the PROG is exited. PROG variables are initialized to NIL. The PROGRAM is a set of expressions to be evaluated in order of their appearance in the PROG function. Identifiers appearing in the top level of the PROGRAM are labels which can be referenced by GO. The value returned by the PROG function is determined by a RETURN function or NIL if the PROG "falls through".

PROGN(U:any]):any
Type: NOEVAL, NOSPREAD.
U is a set of expressions which are executed sequentially. The value returned is the value of the last expression.

RETURN(U:any)
Type: EVAL, SPREAD.
Within a PROG, RETURN terminates the evaluation of a PROG and returns U as the value of the PROG. The restrictions on the placement of RETURN are exactly those of GO.

Standard LISP functions not implemented: PROG2.

2.10 ERROR HANDLING

ERROR(NUMBER:integer,MESSAGE:any)
Type: EVAL, SPREAD.
NUMBER and MESSAGE are passed back to a surrounding ERRORSET (the Little Big LISP reader has an ERRORSET). MESSAGE is placed in the global variable EMSG!* and the error number becomes the value of the surrounding ERRORSET. Local variable bindings are unbound to return to the environment of the ERRORSET. Global variables are not affected by the process.

ERRORSET(U:any,MSGP:boolean,TR:boolean):any
Type: EVAL, SPREAD.
If an error occurs during the evaluation of U, the value of NUMBER from the ERROR call is returned as the value of ERRORSET. In addition, if the value of MSGP is non-NIL, the MESSAGE from the ERROR call is displayed on the currently selected output device. The message appears prefixed with 5 asterisks. The MESSAGE from the ERROR call will be available in the global variable EMSG!*, the

number in ENUM1*.

If no error occurs during the evaluation of U, the value of (LIST (EVAL U)) is returned.

2.11 BOOLEAN FUNCTIONS AND CONDITIONALS

AND([U:any]):extra-boolean
Type: NOEVAL, NOSPREAD.
AND evaluates each U until a value of NIL is found or the end of the list is encountered. If a non-NIL value is the last value it is returned, or NIL is returned.

COND([U:cond-form]):any
Type: NOEVAL, NOSPREAD.
The antecedents of all U's are evaluated in order of their appearance until a non-NIL value is encountered. The consequent of the selected U is evaluated and becomes the value of the COND. The consequent may also contain the special functions GO and RETURN subject to the restraints given for these functions in the "Program Feature Functions" section. In these cases COND does not have a defined value, but rather an effect. If no antecedent is non-NIL the value of COND is NIL.

NOT(U:any):boolean
Type: EVAL, SPREAD.
If U is NIL, return T else return NIL (same as NULL function).

OR([U:any]):extra-boolean
Type: NOEVAL, NOSPREAD.
U is any number of expressions which are evaluated in order of their appearance. When one is found to be non-NIL it is returned as the value of OR. If all are NIL, NIL is returned.

2.12 ARITHMETIC FUNCTIONS

ABS(U:number):number
Type: EVAL, SPREAD.
Returns the absolute value of its argument.

ADD1(U:number):number
Type: EVAL, SPREAD.
Returns the value of U plus 1.

DIFFERENCE(U:number,V:number):number
Type: EVAL, SPREAD.
The value U - V is returned.

DIVIDE(U:number,V:number):dotted-pair
Type: EVAL, SPREAD.
The dotted-pair (quotient . remainder) is returned. The quotient part is computed the same as by QUOTIENT and the remainder the same as by REMAINDER.

GREATERP(U:number,V:number):boolean
Type: EVAL, SPREAD.
Returns T if U is strictly greater than V, otherwise returns NIL.

LESSP(U:number,V:number):boolean
Type: EVAL, SPREAD.
Returns T if U is strictly less than V, otherwise returns NIL.

MAX2(U:number,V:number):number
Type: EVAL, SPREAD.
Returns the larger of U and V. If U and V are the same value U is returned.

MIN2(U:number,V:number):number
Type: EVAL, SPREAD. Returns the smaller of its arguments. If U and V are the same value, U is returned.

PLUS([U:number]):number
Type: NOEVAL, NOSPREAD.
Forms the sum of all its arguments.

PLUS2(U:number,V:number):number

Type: EVAL, SPREAD.
Returns the sum of U and V.

QUOTIENT(U:number,V:number):number
Type: EVAL, SPREAD.
The quotient of U divided by V is returned. Division of two positive or two negative integers is conventional.

REMAINDER(U:number,V:number):number
Type: EVAL, SPREAD.
If both U and V are integers the result is the integer remainder of U divided by V. If either number is negative the remainder is negative. If both are positive or both are negative the remainder is positive.

SUB1(U:number):number
Type: EVAL, SPREAD.
Returns the value of U less 1.

TIMES([U:number]):number
Type: NOEVAL, NOSPREAD.
Returns the product of all its arguments.

TIMES2(U:number,V:number):number
Type: EVAL, SPREAD.
Returns the product of U and V.

The following Standard LISP functions are not implemented:

EXPT FIX FLOAT MAX MIN

2.13 MAP COMPOSITE FUNCTIONS

MAP(X:list,FN:function):any
Type: EVAL, SPREAD.
Applies FN to successive CDR segments of X. NIL is returned.

MAPC(X:list,FN:function):any
Type: EVAL, SPREAD.
FN is applied to successive CAR segments of list X. NIL

is returned.

MAPCAN(X:list,FN:function):any
Type: EVAL, SPREAD.
A concatenated list of FN applied to successive CAR elements of X is returned.

MAPCAR(X:list,FN:function):any
Type: EVAL, SPREAD.
Returned is a constructed list of FN applied to each CAR of list X.

MAPCON(X:list,FN:function):any
Type: EVAL, SPREAD.
Returned is a concatenated list of FN applied to successive CDR segments of X.

MAPLIST(X:list,FN:function):any
Type: EVAL, SPREAD.
Returns a constructed list of FN applied to successive CDR segments of X.

2.14 COMPOSITE FUNCTIONS

APPEND(U:list,V:list):list
Type: EVAL, SPREAD.
Returns a constructed list in which the last element of U is followed by the first element of V. The list U is copied, V is not.

ASSOC(U:any,V:alist):{dotted-pair,NIL}
Type: EVAL, SPREAD.
If U occurs as the CAR portion of an element of the alist V, the dotted-pair in which U occurred is returned, else NIL is returned. ASSOC might not detect a poorly formed alist so an invalid construction may be detected by CAR or CDR.

DEFLIST(U:dlist,IND:id):list
Type: EVAL, SPREAD.
A "dlist" is a list in which each element is a two element list: (ID:id PROP:any). Each ID in U has the indicator

IND with property PROP placed on its property list by the PUT function. The value of DEFLIST is a list of the first elements of each two element list. Like PUT, DEFLIST may not be used to define functions.

DELETE(U:any,V:list):list
Type: EVAL, SPREAD.
Returns V with the first top level occurrence of U removed from it.

LENGTH(X:any):integer
Type: EVAL, SPREAD.
The top level length of the list X is returned.

MEMBER(A:any,B:list):extra-boolean
Type: EVAL, SPREAD.
Returns NIL if A is not a member of list B, returns the remainder of B whose first element is A.

MEMQ(A:any,B:list):extra-boolean
Type: EVAL, SPREAD.
Same as MEMBER but an EQ check is used for comparison.

NCONC(U:list,V:list):list
Type: EVAL, SPREAD.
Concatenates V to U without copying U. The last CDR of U is modified to point to V.

PAIR(U:list,V:list):alist
Type: EVAL, SPREAD.
U and V are lists which must have an identical number of elements. If not, an error occurs (the 000 used in the ERROR call is arbitrary and need not be adhered to). Returned is a list where each element is a dotted-pair, the CAR of the pair being from U, and the CDR the corresponding element from V.

REVERSE(U:list):list
Type: EVAL, SPREAD.
Returns a copy of the top level of U in reverse order.

SUBLIS(X:alist,Y:any):any
Type: EVAL, SPREAD.
The value returned is the result of substituting the CDR of each element of the alist X for every occurrence of the CAR part of that element in Y.

SUBST(U:any,V:any,W:any):any
Type: EVAL, SPREAD.
The value returned is the result of substituting U for all occurrences of V in W.

The following Standard LISP functions are not implemented:

DIGIT LITER SASSOC

2.15 THE INTERPRETER

APPLY(FN:{id,function},ARGS:any-list):any
Type: EVAL, SPREAD.
APPLY returns the value of FN with actual parameters ARGS. The actual parameters in ARGS are already in the form required for binding to the formal parameters of FN.

EVAL(U:any):any
Type: EVAL, SPREAD.
The value of the expression U is computed.

EVLIS(U:any-list):any-list
Type: EVAL, SPREAD.
EVLIS returns a list of the evaluation of each element of U.

QUOTE(U:any):any
Type: NOEVAL, NOSPREAD.
Stops evaluation and returns U unevaluated.

The following Standard LISP functions are not implemented:

EXPAND FUNCTION

2.16 INPUT AND OUTPUT

The user normally communicates with Little Big LISP through the terminal. Little Big LISP allows input from one disk file at a time and output to another.

CLOSE(FILEHANDLE:number):any
Type: EVAL, SPREAD. Closes the file with the internal name FILEHANDLE writing any necessary end of file marks and such. The value of FILEHANDLE is that returned by the corresponding OPEN. The value returned is the value of FILEHANDLE. If an error occurs during a file close or the wrong file handle is given, Little Big LISP stops with an operating system error.

OPEN(FILE:string,HOW:id):number
Type: EVAL, SPREAD.
Open the file with the system dependent name FILE for output if HOW is EQ to OUTPUT, or input if HOW is EQ to INPUT. If the file is opened successfully, a value which is internally associated with the file is returned. This value must be saved for use by RDS and WRS.

PRINT(U:any):any
Type: EVAL, SPREAD.
Displays U in READ readable format and terminates the print line. The value of U is returned.

PRIN1(U:any):any
Type: EVAL, SPREAD.
U is displayed in a READ readable form. In identifiers, special characters are prefixed with the escape character !, and strings are enclosed in "...". Lists are displayed in list-notation.

PRIN2(U:any):any
Type: EVAL, SPREAD.
U is displayed upon the currently selected print device but output is not READ readable. The value of U is returned. Items are displayed so that the escape character does not prefix special characters and strings are not enclosed in "...". Lists are displayed in list-notation.

RDS(FILEHANDLE:number):number
Type: EVAL, SPREAD.
Input from the currently selected input file is suspended and further input comes from the file named. FILEHANDLE is a number returned by the OPEN function for this file.

If FILEHANDLE is NIL the terminal input device is selected. When end of file is reached on a non-standard input device, the standard input device is reselected. RDS returns the internal name of the previously selected input file.

READ():any
Returns the next expression from the file currently selected for input. Valid input forms are: dot-notation, list-notation, numbers, strings, and identifiers with escape characters.

READCH():id
Returns the next interned character from the file currently selected for input. Two special cases occur. If all the characters in an input record have been read, the value of !$EOL!$ is returned. Comments delimited by % and end of line are not transparent to READCH.

TERPRI():NIL
The current print line is terminated.

WRS(FILEHANDLE:number):number
Type: EVAL, SPREAD.
Output to the currently active output file is suspended and further output is directed to the file named. FILEHANDLE is an internal name which is returned by OPEN. The file named must have been opened for output. If FILEHANDLE is NIL the standard output device is selected. WRS returns the internal name of the previously selected output file.

The following Standard LISP functions are not implemented:

EJECT LINELENGTH LPOSN PAGELENGTH POSN PRINC

2.17 SYSTEM GLOBAL VARIABLES

These variables provide global control of the LISP system, or implement values which are constant throughout execution.

!*COMP - Initial value = NIL.
The value of !*COMP controls whether or not PUTD compiles the function defined in its arguments before defining it. If

!*COMP is NIL the function is defined as a LAMBDA expression. If !*COMP is non-NIL, the function is first compiled.

!*ECHO - Initial value = NIL.
If *ECHO is T, input character will be written to the selected output file as they are read.

EMSG!* - Initial value = NIL.
Will contain the MESSAGE generated by the last ERROR call (see the "Error Handling" section).

ENUM!* - Initial value = NIL.
Contains the error number from the last ERROR call.

!$EOL!$ - Value = an uninterned identifier.
The value of !$EOL!$ is returned by READCH when it reaches the end of a logical input record.

!*FLINK - Initial value = NIL.
If !*FLINK is non-NIL, fast call instructions are generated in place of slow indirect calls in compiled code. Once a fast call has been generated it may not be changed back to a slow call. A slow call takes about 250 microseconds and a fast about 5.

!*GC - Initial value = NIL.
!*GC controls the printing of garbage collector messages. If NIL no indication of garbage collection may occur. If non-NIL, the number of free cells remaining after each collection will be displayed on the selected output file.

NIL - Value = NIL.
NIL is a special global variable.

T - Value = T.
T is a special global variable.

!*OUTPUT - Value = T.
If *OUTPUT is T then the result of each LISP reader evaluation is printed otherwise no value is printed.

The following Standard LISP global variables are not implemented:

!$EOF!$!*RAISE

2.18 STANDARD LISP DIFFERENCES

Functions supported by Little Big LISP but are not in the Standard LISP report are listed in the first two sections, low level functions and compiler support functions. The following Standard LISP functions are not currently supported for a variety of reasons:

COMPRESS	FLOAT	PAGELENGTH
CxxxxR	FLUIDP	POSN
DIGIT	FLUID	PRINC
DM	FUNCTION	PROG2
EJECT	INTERN	REMD
EXPAND	LINELENGTH	REMFLAG
EXPLODE	LITER	REMOB
EXPT	LPOSN	REMPROP
FIX	MAX	SASSOC
FLOATP	MIN	UNFLUID
		VECTORP

2.19 SYSTEM ERRORS

The system tries to maintain an operating environment. Some severe errors cause complete termination and program restart with global data intact but with stacks gone and so on. These errors appear with 7 astersisks preceeding them and are followed by the LITTLE BIG LISP prologue heading.

******* STACK OVFLW
This occurs when the stack frame gets to close to the push down stack. This usually means that recursion has preceeded to deeply or infinitely.

******* SYMBOL TABLE FULL
This error occurs when too many symbols have been added to the symbol table. This is usually the result of to many GENSYM's being done or too large a program being read in.

******* STRING SPACE FULL
This error occurs when the string table overflows into the symbol table. This could be too many GENSYM's or too many large string messages.

******* FREE CELLS EXHAUSTED
This error occurs when all available free dotted-pairs have been used. To determine how many available free

pairs there are do:

```
(SETQ !*GC T)
(RECLAIM)
```

2.20 SYSTEM STORAGE ALLOCATION

The number of free pairs is dependent on the available storage and what percentage of the system is reserved for binary program space.Assuming that no binary program space is allocated and the following percentages are used:

Stack	%18
Strings	%13
Symbol Table	%11
Real Address Table	%2
Dotted-pairs	%56

the following systems sizes are possible:

System Size	Stack	Strings	Symbols	Pairs
16k	736	26	85	887
24k	1503	1105	313	2049
32k	2223	2145	533	3169
40k	2961	3211	758	4317
48k	3699	4277	984	5465
56k	4437	5311	1209	6613
64k*	5175	6409	1435	7761

The maximum space that a Jove system can operate in is 56k due to operating system requirements. For the TRS-80 system, approximately 40k is the largest amount of space that can be used due to operating system requirements and the large amount of ROM and program memeory used for memory mapped I/O that is used. 64k would be an ideal machine. Note that the 16k machine has a rediculously small string space. By altering the percentages for various machine sizes (in this case subtracting from free pairs) more reasonable allocations can be made. It is expected that about the smallest system that can possibly be made to do anything is about 12k bytes. By removing all non-required functions of Standard LISP this could probably be reduced to 8k or 9k.

CHAPTER 3

FAST LOAD

Rather than compiling the entire system or reading and compiling code every time, program modules are compiled into relocatable files which we will call fast load files. Most modern LISP systems provide this facility in one form or another. The fast loading program is normally built into the system. It reads binary code and top level S-expressions to interpret. To load a precompiled package enter:

```
(FLOAD "filename")
```

where "filename" is the name of the package on the default input unit (usually a floppy disk). If all goes well the system will respond with NIL. If you try to load the wrong type of file, the error message:

```
***** FAST LOAD ERROR
```

will appear.

To create a fast load file you must enter the following sequence:

```
(FLOAD "COMPILER")          %Load the compiler
(FSLOUT "filename")         %Create a file
  ...                       %LISP source code here.
  ...
  ...
FSLEND                      %End of source code.
```

The file "filename" will appear in the directory. All S-expressions read between the FSLOUT and the FSLEND are directed to "filename" with the exception of DE, DF, and PUTD's which are evaluated and cause compiled code to be directed to the file. To cause an expression to be evaluated during the FSLOUT the function should be flagged as EVAL. Thus (FLAG '(RDS GLOBAL) 'EVAL) will cause RDS and GLOBAL to be executed during the building process rather than deferred for evaluation during the load process.

CHAPTER 4

THE COMPILER

The compilation process is divided into two passes: the first translates LISP into pseudo-assembly code called LAP (for Lisp Assembly Program), the second translates this LAP into absolute machine code and places this in storage for execution or dumps it to a FAP file for later reloading.

4.1 OVERVIEW

The LISP interpreter contains code for reading functions into the LISP system and executing them interpretively much like other microprocessor based systems. Unfortunately interpreted functions require large amounts of storage and execute very slowly.

A more efficient scheme reads functions in the interpretive form, and then compiles them to machine code to be executed directly by the microprocessor. The interpreted version of the function disappears, its storage becomes available for use at a later time.

For example, the function FACT which computes the factorial of a number recursively is defined in Little Big LISP as follows:

```
(DE FACT (N)
   (COND ((LESSP N 2) 1)
         (T (TIMES2 (FACT (DIFFERENCE N 1)) N))))
```

In Little Big LISP, dotted-pairs, of which this function is composed, take 4 bytes each. 23 dotted-pairs are used to define FACT for a total of 92 bytes. Little Big LISP's compiler generates the following code for FACT:

```
0000           (ENTRY FACT)
0000 D7        (RST ALLOC)
0001 02        (DEFB 2)
```

```
0002 FFF2       (STDX HL 0)
0004 F7C0       (LDX HL 0)
0006 110240     (LDI DE 2)
0009 EF         (RST LINK)
000A 6521       (DEFW LESSP)
000C E7         (RST CMPNIL)
000D CA1600     (JPEQ G0002)
0010 210140     (LDI HL 1)
0013 C32600     (JP G0001)
0016            (LABEL G0002)
0016 F7C0       (LDX HL 0)
0018 110140     (LDI DE 1)
001B EF         (RST LINK)
001C 6621       (DEFW DIFFERENCE)
001E EF         (RST LINK)
001F 6721       (DEFW FACT)
0021 F780       (LDX DE 0)
0023 EF         (RST LINK)
0024 6821       (DEFW TIMES2)
0026            (LABEL G0001)
0026 DF         (RST DALLOC)
0027 FE         (DEFB -2)
0028 C9         (RRET)
  (FACT USED 41 BYTES AT 0)
  FACT
```

A total of 41 bytes, less than half the size of the interpreted version. The execution of the compiled version uses considerably less free space than the interpreted version and runs about 5 to 10 times faster.

4.2 COMPILATION MECHANISMS

Much support software is needed for compiled programs which simply move information between registers and call subroutines to perform most operations. In this section we describe how various LISP constructs are implemented in LAP and enumerate the various support functions required.

4.2.1 Parameter Passing

Zero to 3 parameters may be passed to a function. The first argument of a function (if it has any) will always be in the HL register pair, the second in DE, and the third in BC. Functions with more than three arguments cannot be compiled.

4.2.2 Stacks

Function parameters and PROG type variables are kept in a stack frame, a contiguous block of locations pointed to by the IX index register. When a function is invoked it creates a new frame on the top of the stack by calling the ALLOC support subroutine. ALLOC adds a number to IX to create a new empty stack frame. It also checks for stack overflow and signals an error if this has happened or is about to happen. When a function terminates it calls the DALLOC routine which subtracts the number of locations used from IX freeing the space for use by the next function.

Storing and retrieving values from the stack frame is accomplished by the two support routines LDX and STOX. Since these operations occur frequently in compiled code it is necessary that they use as little storage as possible. Therefore the LDX and STOX routines should be called using the Z80 RST instruction with the following byte containing what register pair is to be stored (or loaded), and the displacement from the top of the stack frame. The format of the control byte is given in the source code listings of LDX and STOX. The LAP instructions generated by the compiler are also called LDX and STOX and contain the register pair name and what displacement is to be used.

Let us examine a LAMBDA function with an imbedded PROG and look at the code generated by the compiler.

```
(LAMBDA (A B) (PROG (C D) ...) ...  )
```

The generated LAP code pushes and pops the stack frame and stores registers into the frame.

LISP LAP Stack Frame

```
(LAMBDA (A B) ...

          (RST ALLOC)      +--------------+
          (DEFB +4)        |L             |<-- new IX
          (STOX HL 0)      +--     A    --+
          (STOX DE -1)     |H             |
               .           +--------------+
               .           |E             |
               .           +--     B    --+
               .           |D             |
               .           +--------------+
               .           .              .<-- old IX

..(PROG (C D) ...

          (RST ALLOC)      +--------------+
          (DEFB +4)        |L             |<-- new IX
          (LDI HL NIL)     +--     C    --+
          (STOX HL 0)      |H             |
          (STOX HL -1)     +--------------+
               .           |L             |
               .           +--     D    --+
               .           |H             |
                           +--------------+
                           .      A       .<-- old IX
                           .      B       .
```

Nested PROGs cause more frames to be allocated up to a maximum of 64 accessable variables. The limiting factor is the 6 bits of displacement in the LDX and STOX macros.

The Z80 internal stack (pointed to by the SP register) is used for saving return addresses and intermediate values during function evaluation. A call to a function FUN3 with 3 arguments stores the results of evaluation of the first two arguments on the Z80 stack while the third is being computed. The values are popped into the appropriate registers just before the function is invoked.

```
(FUN3 (FUNA ...) (FUNB ...) (FUNC ...))
```

would generate the following code sequence:

```
... evaluate FUNA ...
    (PUSH HL)      ;Save result of FUNA on stack.
... evaluate FUNB ...
    (PUSH HL)      ;Save result of FUNB on stack.
... evaluate FUNC ...
    (LDHL BC)      ;Move BC to HL.
```

```
(POP DE)        ;Result of FUNB is second argument.
(POP HL)        ;Result of FUNA is first argument.
(RST LINK)      ;Call FUN3.
(DEFW FUN3)
```

4.2.3 Calling Functions

The compiler will not always know the address of a function being called because it might not be defined yet. Even if the function is defined the compiler does not know whether it will be compiled or interpreted at run time. A special internal subroutine called LINK is used to transfer control at run time. Since both compiled and interpreted functions can exist at the same time, LINK will perform either of two functions. If an interpreted function is being called from compiled code the LISP interpreter will be invoked for that function. If the function being called is compiled or is a system function the call to LINK will be replaced by a direct call to that function. The call to the LINK function must be an RST type link so that the 3 byte Z80 CALL instruction will exactly replace the compiled call. If the system global variable !*FLINK is NIL, the substitution will not take place and the slow link form will be used. This is a useful debugging tool as it allows you to compile functions and change their definitions (for tracing) without reloading the system.

Compiled as:	Changed by LINK to:
(RST LINK) (DEFW function-name)	(CALL function-address)

The two byte DEFW attached to the LINK contains the symbol table pointer of the function being called. At execution time the LINK routine looks for either a compiled or interpreted function attached to he name and either invokes EVAL, generates the CALL, or if the *FLINK flag is on, just transfers to the function. If no such function is defined, the undefined function error will occur.

4.2.4 The LIST Function

The LIST function is compiled in a special way to take advantage of the Z80 internal stack. The arguments of the LIST function are compiled and the results of each are pushed onto the stack. When all have been computed the support function CLIST is called.

```
(LIST (F1 ...) ...  (Fn ...))
```

compiles to:

```
... evaluate F1 ...
    (PUSH HL)     ;Save result of F1 for CLIST.
        .
        .         ;Evaluate other arguments.
        .
... evaluate Fn ...
    (PUSH HL)     ;Save result of Fn for CLIST.
    (LDA n)       ;Number of values on stack for
    (CALL CLIST)  ;call to CLIST routine.
```

4.2.5 COND Compilation

The LISP COND function is compiled into a series of tests and conditional jumps. The CMPNIL support routine compares the result of a predicate to NIL and sets the Z80 NZ and Z flag bits which control the conditional branch instructions generated. If the last predicate of the COND is T, the predicate and jump will not be compiled (this is the usual case).

```
(COND (a0 c0) ...  (an cn))
```

generates the following code:

```
... evalate a0 ...
    (RST CMPNIL)     ;Is a0 NIL?
    (JPEQ G0001)     ;Yes, jump to next antecedent.
... Evaluate c0 ...
    (JP G0002)       ;First consequent evaluated, quit.
    (LABEL G0001)    ;Come here if a0 is not true.
      .
      .              ;Evaluate other antecedents.
      .
    (LABEL G000x)    ;Try last predicate.
*... evaluate an
*   (RST CMPNIL)     ;Is last one NIL?
*   (JPEQ G0002)     ;Go return NIL then.
 ... evaluate cn ...
    (LABEL G0002)    ;Always come here when done.
```

Lines preceeded by an asterisk are not generated if the last predicate is T.

4.2.6 PROG, GO, And RETURN

The PROG function and the control constructs GO and RETURN are compiled by plugging labels and values into a template. The compiler does not check for GO's to undefined labels, this is done by LAP. RETURN's not in PROGs and illegally nested GO's are also not checked.

```
(PROG (X)
      .
LBL ...
      .  ... (RETURN val)
      .
      .
      (GO LBL)
      .
      ...)
```

compiles to:

```
      (RST ALLOC)      ;Space to save variable X allocated.
      (DEFB +2)
      (LDI HL NIL)     ;PROG variable set to NIL.
      (STOX HL 0)
          .
          .
      (LABEL LBL)      ;A PROG label generates a LABEL.
          .
          .
  ... evaluate val ...
      (JP G0001)       ;Jump to end of this PROG.
          .
          .
      (JP LBL)         ;(GO LBL) generates a jump.
          .
          .
      (LABEL G0001)    ;All RETURN's come here.
      (RST DALLOC)     ;Free the stack frame allocated
      (DEFB -2)        ;for X.
```

4.2.7 AND And OR Compiled

AND and OR are compiled identically except that the evaluation of the arguments of AND terminates if one is NIL, and the evaluation of OR terminates if one is non-NIL. The compilation of AND generates JPEQ instructions after a comparison to NIL, and the compilation of OR generates JPNEQ instructions.

```
(AND a0 ..  an)
```

compiles to:

```
... evaluate a0 ...
    (RST CMPNIL)     ;Is result of a0 NIL?
    (JPEQ G0001)     ;Stop evaluation if yes.
       .
       .             ;Evaluate other arguments.
       .
... evaluate an ...
    (LABEL G0001)    ;Always end up here.
```

The OR function instance compiles exactly the same way, but JPNEQ is generated instead of JPEQ.

4.2.8 Constants, Variables, And Quoted Values

These items are loaded directly into the correct register for the function to which they are to be passed. Local and Global variables may have values assigned to them with the appropriate store instructions.

Quoted items are saved on a list of compiled quoted values so that the garbage collector will not remove them. The value representing the quoted item is loaded into the appropriate register.

4.3 THE LAP INSTRUCTION SET

The LISP Assembly Program accepts the following instruction set generated by the compiler (or user) and generates absolute machine code or the correct information to place in a FAP file. The following symbols are used:

pp - denotes a register pair HL, DE, or BC.
nn - an immediate 16 bit value.
n - denotes an immediate 8 bit value.
lbl - denotes a label found somewhere.
dsp - denotes an 8 bit stack displacement.
addr - denotes a 16 bit global address.

(ENTRY name)
Serves as the entry point of function "name". ENTRY does not generate any Z80 instructions.

(LABEL lbl)
Defines a label referenced elsewhere in the current function. Labels are not known outside of a function.

(LDHL pp)
Causes two Z80 register to register instructions to be generated to transfer the contents of HL to BC or DE.

(LDI pp nn)
Generates a "load immediate" instruction to load the register pair pp with the 16 bit value nn. nn may be a number, T or NIL, or a quoted item.

(LDX pp dsp)
Generates a call to the LDX routine to load the register pair pp with a 16 bit value at dsp*2 bytes from the top of the current stack frame. The control byte contains both the register identifier and the displacement.

(LDA n)
Causes a single "Load A Immediate" instruction to be generated which loads the 8 bit value n into the Z80 A register. This instruction is used in the compilation of the LIST function.

(STOX pp dsp)
Generates a call to the STOX reoutine to store register pair pp at the displacement dsp*2 bytes from the top of the currently active stack frame. The control byte generated to follow the short call to the STOX routine contains both the register identification to store and the 6 bit displacement.

(STO pp addr)
Generates a "store direct" instruction to store the value in register pair pp in the value cell of a global variable at addr.

(JP lbl)

(JPEQ lbl)

(JPNEQ lbl)
A long Z80 jump instruction is generated to get to the location of the label named. The JP instruction is an unconditional jump. The JPEQ instruction generates a jump conditional on the Z condition code and the JPNEQ based on the NZ condition code set.

(PUSH pp)
Generates the single byte instruction to push register pair pp onto the Z80 stack.

(POP pp)
Generates the single byte instruction to pop the Z80 stack into the register pair pp.

(CALL name)
Generates a long 3 byte call instruction to the absolute address of name. This absolute address is stored under the CALL property as two integers representing the bytes of the address in reverse order. Currently ALLOC, DALLOC, and the CLIST support routine addresses are so stored and called.

(RST name)
Generates the single byte Z80 call instruction to one of 8 possible routines. A minimum of 3 RST calls must be available for the compiled code to operate correctly, one for LINK, one for LDX, and one for STOX. The other RST's used in this system may be changed into Z80 CALL instructions, but the compiled code will be significantly longer. Current calls are to:

CMPNIL - compare HL to NIL, set Z, NZ.
STOX - store register pair in stack frame.
LDX - retrieve register pair from stack frame.
CAR - take the CAR of HL.
CDR - take the CDR of HL.
LINK - slow link to defined function.

(RET)
Generates the Z80 "return from subroutine" instruction.

(DEFW name)
Generates an identifier name for the LINK call. LINK expects a symbol table pointer.

(DEFB n)
Generates a single byte numeric value which is used as the control byte for the STOX and LDX stack frame primitives and for the ALLOC and DALLOC calls.

4.4 USING THE COMPILER

The compiler is normally kept as a FAP file on the same disk as the interpreter. It must be manually loaded by typing:

```
(FLOAD "COMPILER")
```

The name of the compiler varies from system to system. After 10 or 20 seconds the machine will respond with the value NIL and the prompt character. There are two options at this point. You may either manually compile functions by typing:

```
(COMP fn type body)
```

Where "fn" is the name of the function, "type" is either EXPR, or FEXPR, and "body" is the LAMBDA expression of the function

to be compiled. To compile the factorial function presented earlier using this method, you would enter:

```
(COMP 'FACT 'EXPR
  '(LAMBDA (N)
     (COND ((LESSP N 2) 1)
           (T (TIMES2 N (FACT (DIFFERENCE N 1)))))))
```

Functions may be compiled as normally entered by setting the !*COMP switch to T. When a function is entered using either PUTD, DE, or DF and this flag is on it will be compiled before being defined. Thus:

```
(SETQ !*COMP T)
(DE FACT (N)
  (COND ((LESSP N 2) 1)
        (T (TIMES2 N (FACT (DIFFERENCE N 1))))))
```

will result in the function being compiled before being defined.

4.4.1 Compiler Flags

The following flags and global variables are used by the compiler and are of interest to the user.

!*COMP
: When non-NIL, causes DE, DF, and PUTD to automatically call the compiler to define a function.

!*FLINK
: When non-NIL, the RST LINK - DEFW name LAP instructions ae replaced by fast CALL instructions when executed. This happens only when the function call is executed.

!*FSLOUT
: When non-NIL, causes the assembler to generate the code for a FAP file. !*FSLOUT should be set only by the FSLOUT function discussed under generating FAP files.

!*LAPP
: When non-NIL, causes the LAP generated by the compiler, and the hexadecimal machine code generated by the assembler to be listed on the selected output device. This flag should not be set while generating FAP files.

4.4.2 Using LAP

The Lisp Assembly Program may be called directly with a list of LAP instructions in the global variable LAPS. This may be useful for optimizing functions that are critical to the execution of a program. Likewise, it is easy to modify the assembler to add new instructions to provide the ability to build special I/O functions, special data transfer functions and the like without modifying the source of the interpreter.

CHAPTER 5

RLISP

Some may consider the rigours of coding in LISP with all its parentheses a bit onerous. To provide a syntax more amenable to users of contemporary high level programming languages, a parser from RLISP to LISP has been implemented. This syntax was invented by A. C. Hearn in 1973 to facilitate the implementation of a symbolic algebra system, REDUCE [3]. The subset described here is reasonably complete and is restricted only by the subset of Standard LISP implemented in Little Big LISP.

The RLISP parser contains its own top level EVAL loop which reads LISP expressions in RLISP syntax, parses them into LISP and if there are no syntax errors, evaluates them. The user can drop into LISP at any time.

The remainder of this section presents the syntax of RLISP together with examples of its use. The section concludes with a list of known differences with the distributed version of RLISP.

5.1 PROCEDURES

Functions are defined in RLISP as procedures with parameters. The following syntax is used:

1. <function> ::= <ftype> PROCEDURE <id> <parameter list>;
   <unlabeled statement>;

2. <ftype> ::= EXPR | SYMBOLIC | FEXPR

3. <parameter list> ::= () | <id> | (<id list>)

4. <id-list> ::= <id>[,]*

A <function> is a PROCEDURE statement preceded by its type. Note that EXPR and SYMBOLIC both stand for EXPR (EVAL/SPREAD) type procedures. The identifier which must follow the PROCEDURE keyword is the name of the function being defined. The parameter list must be () if the function has no parameters. If the function has a single formal parameter it need not be enclosed in parentheses. Two or more parameters must be enclosed in parentheses and the identifiers must be separated by commas. Functions with more than three parameters may be defined but may not be compiled. The statement following the procedure heading may be a compound BEGIN - END block or a simple statement or function call.

The RLISP procedure is parsed into a DE or DF function form. The name and formal parameters from the heading line become parts of the call and the statement following becomes the body of the function. The LAMBDA expression is generated by DE and DF's call to PUTD.

5.2 STATEMENTS

There are several different statement types in RLISP corresponding to the different control constructs. The BEGIN - END block is translated into a LISP PROG function.

5. <BEGIN-END block> ::=
   BEGIN SCALAR <id-list>; <statement>[;]* END |
   BEGIN <statement>[;]* END

The identifiers in the optional SCALAR clause are variables local to the BEGIN - END block. These become the variables of the PROG while the statements separated by semicolons become the body.

6. <statement> ::= <id>: <unlabeled statement> |
   <unlabeled statement>

Labeled statements may occur only within BEGIN - END blocks. A statement may have a single label which serves only as the object of a GO TO statement. Labels are transferred, as is, to the generated PROG form.

7. <unlabeled statement> ::=<BEGIN-END block> |
   <IF statement> |
   <do group> |
   <WHILE statement> |
   <FOR statement> |
   <RETURN statement> |
   <GO TO statement> |
   <value statement>

An unlabeled statement may be a control construct or a value statement, a general catch all for stand alone function invocation, assignment, and the like.

8. <IF statement> ::=
IF <expression> THEN
<unlabeled statement 1> ELSE
<unlabeled statement 2> |
IF <expression> THEN <unlabeled statement>

The IF statement is in the classical form as either IF ... THEN ... ELSE... or just plain IF ... THEN. Like all other RLISP statements, an IF statement has a value. If the expression has a non-NIL value, then the value is the value of unlabeled statement 1 otherwise the value of unlabeled statement number 2. If there is no ELSE clause and the value of the expression is NIL, the value of the statement is NIL. Multiple IF...THEN...ELSE IF...THEN...ELSE IF... statements are parsed into a single COND with multiple antecedent consequent pairs.

9. <do group> ::= << <unlabeled statement>[;]* >>

The do group is translated into the LISP PROGN form. Statement labels are not permitted within the group, but GO TO's and RETURN's are permitted within the scope of a surrounding BEGIN - END block. The value of the do group is the value of the last statement.

10. <WHILE statement> ::=
WHILE <expression> DO
<unlabeled statement>

The WHILE statement repeatedly evaluates the unlabeled statement while the expression is non-NIL. The value of a WHILE statement is NIL unless there is a RETURN within the unlabeled statement which is not embedded within a BEGIN - END block. The statement is translated into a PROG form with an internal loop. The unlabeled statement is the consequent of a COND or a single statement within this PROG, thus any RETURN will be the value of the loop or the value of an internal PROG from the use of a nested BEGIN - END block.

11. <RETURN statement> ::= RETURN |
RETURN <unlabeled statement>

RETURN may be used only within a BEGIN - END block and is translated directly into the regular RETURN function call. RETURN without a parameter is translated into (RETURN NIL).

12. <GO TO statement> ::= GO TO <id>

The GO TO statement may be used only within a BEGIN - END block and only to a label at the current lexical level within that block.

13. <FOR statement> ::=
FOR EACH <id> IN <expression>
DO <unlabeled statement> |
FOR EACH <id> IN <expression>
COLLECT <unlabeled statement> |
FOR <id>:=<expression 1>:<expression 2>
DO <unlabeled statement>

There are three forms of the FOR statement. The first form evaluates the unlabeled statement with the identifier set to each successive element of the list resulting from the expression. This FOR is mapped into something like the MAPC function but in an internal form more suitable for compilation. The value of a FOR statement of the first form is always NIL. The second form of the FOR statement is like the first but the word COLLECT instead of DO signifies that the results of the statement being evaluated are collected into a list which is returned as the value of the FOR statement. This form is translated into an internal form roughly equivalent to a MAPCAR statement. The only difference between these forms and MAPC and MAPCAR is that local variables may be used within the unlabeled statement with impunity whereas they would have to be GLOBAL or FLUID in other systems. The final form of the FOR statement is the usual iterative form which sets the identifier to the value of the first expression and increments it evaluating the unlabeled statement each time until the value of the variable is greater than the value of expression 2. Expression 2 is recomputed each time through the loop. This form of the FOR statement always has the value NIL and is translated into a nested PROG. It may not have GO TO's out of the range of the loop.

5.3 VALUE STATEMENTS

Any statement which can not be parsed as a control construct is assumed to be a value statement, that is, an infix expression. The infix operators implemented are listed in increasing order of precedence:

```
:=
:=
OR
AND
<, >, LEQ, GEQ, NEQ, EQ, =
+ -
* /
**
.
```

What follows is the BNF for expressions starting with the lowest precedence and working to the highest. Expressions are

what you would expect with the exception that function calls with single arguments need not have the arguments enclosed in parentheses, and the . operator for CONS, and the ' for QUOTE.

14. <value expression> ::=
<id> := <unlabeled statement> |
<boolean term>

A value expression can assign the value of a statement to a variable or is just a boolean term. Note that an unlabeled statement may be another value expression (the usual case).

15. <boolean term> ::= <boolean secondary> |
<boolean secondary> OR <boolean term>

A boolean term is a number of boolean secondaries separated by OR's. Note that all the terms are collected into a single OR by the parser to keep down the size of expressions.

16. <boolean secondary> ::= <relational expression> |
<relational expression> AND <boolean secondary>

A boolean secondary is like a boolean term only AND is the connective. An expression ...AND...AND...AND... is collected into a single (AND ...).

17. <relational expression> ::= <CONS expression> |
<CONS expression>
<relational operator>
<CONS expression>

18. <relational operator> ::=
< | > | = | NEQ | LEQ | GEQ | EQ

A relational expression is two expressions separated by a diadic operator which returns NIL or something else. The < operator is translated into GREATERP, the > operator to LESSP, the = operator to EQUAL, and the other operators are translated into themselves.

19. <CONS expression> ::= <arithmetic expression> |
<arithmetic expression> . <CONS expression>

Two expressions separated by a . are the CAR and CDR parts of a CONS function call. The dot operator is right associative, so in a string of dot operators, the rightmost one is done first. Dots within LISP S-expressions are not affected.

20. <arithmetic expression> ::= <arithmetic term> |
<arithmetic term> + <arithmetic expression> |
<arithmetic term> - <arithmetic expression>

The + and - operators are right associative and are translated into PLUS2 and DIFFERENCE respectively.

21. <arithmetic term> ::= <arithmetic secondary> |
<arithmetic secondary> * <arithmetic term> |
<arithmetic secondary> / <arithmetic term>

The * and / operators are right associative and are translated into TIMES2 and QUOTIENT calls respectively.

22. <arithmetic secondary> ::= <QUOTE expression> |
<QUOTE expression> ** <arithmetic secondary>

The exponentiation operator ** is right associative and translates into an EXPT function invocation. Exponentiation is allowed only to positive integer powers.

23. <QUOTE expression> ::= <primary> |
'<LISP S-expression>

The ' operator causes the LISP S-expression reader to be invoked to read the following LISP S-expression. Note that ' may not be used to quote an RLISP expression. One must use the QUOTE function explicitly to do this.

24. <primary> ::= <unsigned integer> |
<string>
(<unlabeled statement>) |
<id> |
<id> <expression> |
<id>() |
<id> (<expression>[,]*)

A primary is an atom (like an unsigned integer, a variable name, or a string), or an unlabeled statement (usually an expression) enclosed in parentheses, or a function call. A function with no arguments must have () following it to distinguish it from a variable. A function with a single formal parameter may be followed directly by its parameter which need not be enclosed in parentheses. Functions with multiple parameters must have these parameters enclosed in parentheses and separated by commas.

5.4 SYSTEM FLAGS

For the most part the RLISP reader works exactly like the Little BIG LISP reader. There are a number of flags which affect the way in which the system operates. These are all prefixed by a * and may be set on by setting them to T or off by setting them to NIL.

1*DEFN - Initial Value = NIL.
If this variable is non-NIL, the parser form of the RLISP expression entered will be displayed and not evaluated. By

this means you may examine the parsing of a function or convert RLISP into LISP. By directing output to a file and turning on the !*DEFN flag and reading in an RLISP file, a file with nothing but LISP can be created.

!*OUTPUT - Initial Value = T.
If this variable is NIL, the results of an evaluation of an expression read by the RLISP reader will not be printed.

WS -
This variable will always contain the results of the last evaluation of the RLISP reader.

5.5 ERROR MESSAGES

The RLISP parser implemented for Little Big LISP is not always successful in parsing. All parsing errors are caught by the reader which scans to a semicolon when an error is detected and restarts at the top level. The errors are listed here together with their probable causes.

***** Missing Semicolon
When the parser finishes with a form the last token must always be a semicolon or dollar sign. If this is not the case, an error occurs and the parser scans until one is found.

***** Missing PROCEDURE
The word PROCEDURE did not follow the keywords EXPR, FEXPR, or SYMBOLIC. This is usually a misspelling of the word.

***** Missing procedure name
The token following the word PROCEDURE was not an identifier.

***** Missing THEN
In an IF statement, the THEN could not be found. This usually means that the expression of the IF was improperly constructed.

***** Missing DO
In a WHILE or FOR statement, the DO keyword could not be found. This usually means the conditional expression or FOR loop object was not properly parsed.

***** Missing END
The last statement of a BEGIN - END block must not be followed by a semicolon, but rather an END. This usually means that the last statement has been improperly constructed. If the last statement has a semicolon on it, the END will be an unrecognizable statement.

***** Missing >>
The last statement of a do group (<< ... >>) must not be followed by a semicolon, but rather the >> terminator. If the last statement is improperly constructed, this error will occur. If a semicolon follows the last statement the unrecognizable statement error will occur.

***** Missing TO
In a GO TO statement, the TO is missing. This is usually caused by forgetting that RLISP uses GO TO and LISP uses just plain GO for transfer of control.

***** Unrecognizable statement
This happens when the first token of a statement is not a keyword, nor can the expression parser make an expression out of it. If the first word of a statement is a keyword like ELSE, TO, DO, or COLLECT, this error will occur. Usually it means a semicolon in the middle of a statement before the error, or a semicolon as the last statement in a block.

***** Missing (
A formal parameter list that has more than a single variable or none at all must start with a left parenthesis.

***** Missing)
A formal parameter list that is poorly formed or is missing the closing right parenthesis will cause this error as will improperly balanced parentheses in expressions.

***** Non-id
Formal parameters must always be identifiers.

***** Operator misplaced
This error occurs when two infix operators occur without an intervening operand.

***** ERROR TERMINATION
All errors will be suffixed by this message meaning that parsing will procede only with more user input.

When an error occurs during evaluation, the error message will be printed followed by the omnipresent ERROR TERMINATION message. The WS global variable will contain the error message number.

5.6 STARTING UP RLISP

The RLISP system must first be loaded from the system disk in the fast load format. The FLOAD function is entered in LISP format with the name of the file. When the system has been loaded properly you enter:

```
(BEGIN)
```

and the system will respond immediatly with:

```
RLISP - <date>
```

where the <date> is the date the system was last created. To exit from RLISP back into LISP parsing you enter:

```
END;
```

to which the system should immediately respond:

```
ENTERING LISP ...
```

You may reenter RLISP at any time. All the functions of the basic Little Big LISP system are available in RLISP and you may load other packages on top of it, including the compiler, big number package and so on.

5.7 EXAMPLES

The following few functions illustrate some of the features of RLISP. They are given with their equivalent LISP translations.

```
% Factorial in RLISP (see compiler section for LISP).
EXPR PROCEDURE FACT N;
IF N < 2 THEN 1
  ELSE N * FACT(N - 1);


% SUPREV - super reverse of tree to all levels.
EXPR PROCEDURE SUPREV A;
IF ATOM A THEN A
  ELSE SUPREV CDR A . SUPREV CAR A;
(DE SUPREV (A)
  (COND ((ATOM A) A)
        (T (CONS (SUPREV (CDR A))
                 (SUPREV (CAR A)) ))))


% A procedure with a WHILE loop.
```

```
EXPR PROCEDURE SEMISCAN();
<< WHILE NOT(TOK!* EQ '!; AND EQN(TYPE!*, 6))
     DO NTOK();
   NTOK() >>;
(DE SEMISCAN NIL (PROGN
   (PROG NIL
    G0008 (COND
            ((NULL (NOT (AND
                 (EQ TOK!* (QUOTE !;))
                 (EQN TYPE!* 6))))
               (RETURN NIL)))
          (NTOK)
          (GO G0008) )
   (NTOK) ))
```

List of References

1. Marti, J. B., A. C. Hearn, M. L. Griss, C. Griss, "Standard LISP Report", SIGPLAN Notices, Vol. 14, No. 10, October 1979, pp. 48-68, reprinted in SIGSAM Bulliten, Vol. 14, No. 1, 1980.

2. Griss, M. L., private communication.

3. Hearn, A. C., "REDUCE 2 User's Manual", Utah Symbolic Computation Group, UCP-19, March 1973.