UO-LISP MANUAL Version 1.5b

on

Z80 Base CPU, TRS-80 Model's I and III
with TRSDOS or TRSDOS like Operating Systems

by

Dr. Jed Marti

Distributed by

NORTHWEST COMPUTER ALGORITHMS

JULY 1984

ABSTRACT

This manual describes the Z80 based LISP system, UO-LISP, its data structures, built in functions, operating procedures, the complier and optimizer, an RLISP parser, a trace package, and a structure editor.

CONTENTS

INTRODUCTION

UOLISP is a subset of Standard LISP [1] implemented for the Z80 microprocessor. It runs in a minimum of thirty two thousand bytes of storage and most effectively with forty eight 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 optimizing phase for the compiler
5. A parser for a subset of RLISP [2]
6. A function trace feature
7. A LISP structure editor and pretty printer
8. Numerous support packages

This manual is not intended as an introduction to LISP. Readers interested in learning LISP are advised to consult one of the tutorials on the subject [3-6]. Some of the function names may be different from those used in the books but the correct name can usually be found by examining section titles of this manual. Users of Standard LISP [1] will find lists of differences in each section as well as with individual functions.

A text file full of replacements and additions for UOLISP is in file FUNS/LSP.
It has been compiled into a fast-load file for easy use - this is file FUNS
You probably should read FUNS in at the beginning of each session with UOLISP. If you want to add some more functions to FUNS, just edit FUNS/LSP with your favorite word processor, add the functions, save it back out (in ASCII format! - that's "S, A" for SCRIPSIT), and recompile it according to the instructions given in the FUNS/LSP file.

CHAPTER 1

DATA TYPES

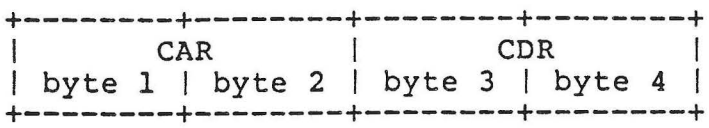
1.1 ITEMS

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
Ø	Used by garbage collector to indicate item is in use.
1-2	Data type: ØØ - Dotted-pairs. Ø1 - Identifiers. 1Ø - Integers. 11 - Strings and function pointers.
3	Subtype bit for strings and function pointers. 11Ø - Function pointer. 111 - String.

1.2 DOTTED-PAIRS

Up to 8192 dotted-pairs (32k bytes) may be referenced by the UOLISP system depending on the amount of available storage. A minimum of 3ØØ 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.

List notation eliminates extra parentheses and dots. The list (a . (b . (c . NIL))) in list notation is (a b c). List notation and dot notation may be mixed as in (a b . c) or (a (b . c) d) which are (a . (b . c)) and (a . ((b . c) . (d . NIL))). In BNF lists are recognized by the grammar:

```
<left-part> ::= ( | <left-part> <any>
<list> ::= <left-part>) |
           <left-part> .  <any>)
```

Note: () is an alternate representation of NIL.

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 may reference up to eight thousand identifiers, though there are usually only 500 or so free ones.

Identifiers can have from 1 to 255 character print names. The first character must be alphabetic or any other character preceeded by the ! escape character. Successive characters may be alphanumeric or other characters prefixed by the escape character. If the value of the !*RAISE flag is NIL, lower case characters are not converted to upper case. On machines with no lower case, there is no !*RAISE flag.

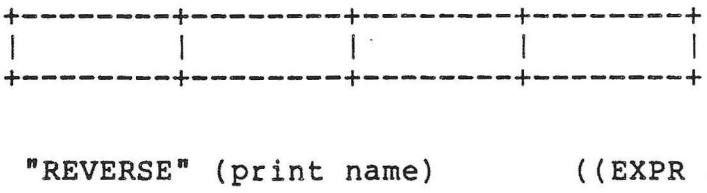
Each identifier is two items in the symbol table. The first is a pointer to the string, called the print name by which the identifier is known to the outside world. The second is a pointer to a structure of values associated with the identifier called the property list. The symbol table is a vector of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



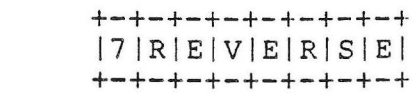
"REVERSE" (print name) ((EXPR . $6003))

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. Both positive and negative integers are recognized by the LISP reader.

1.5 STRINGS

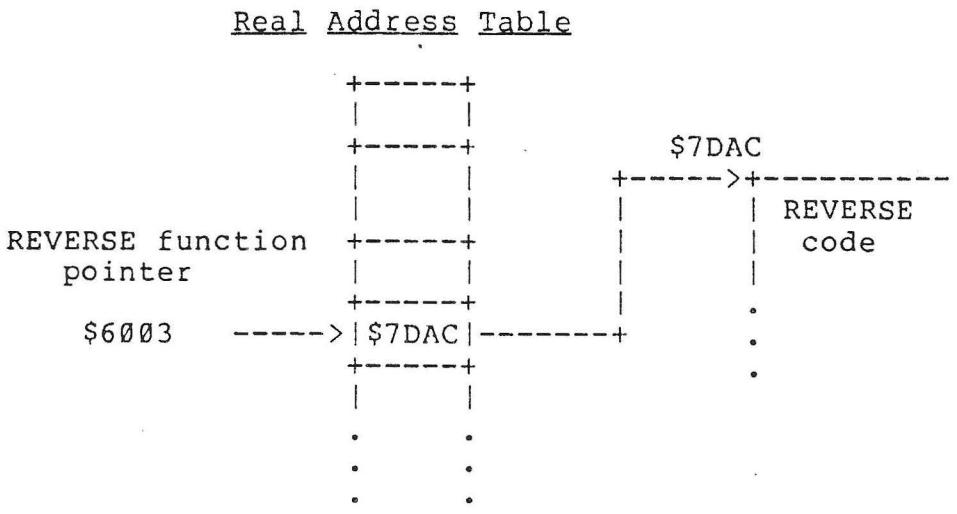
Strings are arbitrary character sequences from 0 to 255 characters in length. Strings serve as print names for identifiers or as constants. A string pointer is a 12 bit offset into the string space which is a single large character vector. The minimal system requires a few more than 1200 bytes of string 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, activation records for parameter bindings and 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.

The garbage collector examines both stacks for pointers to structures. To assure that only valid items are contained in the stacks means that:

1. All values less than 8192 ($2000) are pointers to dotted-pairs.

2. All items greater than or equal to 8192 are atomic. 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 closely resemble those of the report.

Each function name appears with formal parameter names and their expected types. These are any of the following:

alist - An association list. This is a list of dotted-pairs, the CAR of which is an identifier and the CDR an associated value of any type.

any - Any item or structure is permissible.

atom - Any item which is not a dotted-pair is an atom.

boolean - T (for true), or NIL (for false).

dlist - A list for the DEFLIST function consisting of a list of two element lists the first element being an identifier and the second a value to be added to its property list (see DEFLIST).

dotted-pair - Any value returned by CONS.

extra-boolean - NIL or any value. Any value other than NIL stands for true.

ftype - Either of the identifiers EXPR or FEXPR, one of the two function types implemented.

function - A lambda expression, or a function-pointer.

function-pointer - An indirect pointer to the starting address of a function.

id - An identifier.

integer - An integer value.

lambda-expression - A LISP S-expression of the form (LAMBDA (...) ...).

number - A numeric value (an integer).

string - A string of characters surrounded by double quotes.

word - A dangerous value used by the compiler during generation of absolute addresses of code.

If the formal parameter may be of more than one type, the types are listed surrounded by braces { ... }. If there can be an indefinite number of formal parameters, the repeated parameter is enclosed in square brackets [...].

The type of value that the function returns follows its prototype. The method of evaluation of the function's arguments appears on the second line of the definition. A function either has its arguments evaluated before it is invoked (an EVAL type function), or are bound to the formal parameters without evaluation (a NOEVAL type function). The actual parameters of a function are either spread amoung the formal parameters (a SPREAD type function), or are collected into a list and bound to the single formal parameter (a NOSPREAD type function). EVAL, SPREAD type functions are called EXPR's, and NOEVAL, NOSPREAD functions FEXPR's. There are currently no EVAL, NOSPREAD or NOEVAL, SPREAD functions implemented in UOLISP.

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.

(R!$):id
Type: EVAL, SPREAD.
This function returns the character currently being pointed to by the input scanner. It does not however scan ahead another character as does READCH. This function is used by the RLISP parser to form diphthongs.

(GETP!$ X:id):any
Type: EVAL, SPREAD.
Return the property list for the identifier X. No type checking is performed.

(PUTP!$ X:id PROP:any)
Type: EVAL,SPREAD.
Replace the property list of the identifier X with PROP. No type checking is performed.

(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
Type: 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
Ø	nnn	Integer
1	id	Identifier
2	*	(
3	*	.
4	*	)
5	string	String
6	id	Single character converted to identifier
7	*	Quote character (')

(* 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 Ø to 65535. The function is useful for determining the order of items within a space.

(IDL!* X:id):integer

Type: EVAL, SPREAD.
Returns the number of characters in the print name of X. This does not include any !'s which might have to be included on special characters.

(STL!* X:string):integer

Type: EVAL, SPREAD.
Returns the number of characters in a string less the two enclosing "'s.

2.2 COMPILER SUPPORT FUNCTIONS

The following functions are used by the compiler to create absolute code or fast 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 of BPTR is incremented by 1.

(CPLUS X:integer):word

Type: EVAL, SPREAD.
Add the 12 bit sign extended value of 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 Ø to 255.

(MKCODE):function-pointer

Type: EVAL, SPREAD.
Create a new function pointer to return as the value of MKCODE. The current value of the global variable BPTR is stored in the real address table at the position pointed to by the new function pointer. This function is used to enter a compiled function into the real address table. It should be called before any code is deposited with BPUT or WPUT.

(MKGLOB X:dotted-pair):list

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

(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 variable MLIST so that it will not be removed by the garbage collector. MLIST is not accessible from LISP. 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 X as an unsigned positive integer in the range Ø 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

These 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. Strings with the same characters are always EQ.

(EQN U:any V:any):boolean
Type: EVAL, SPREAD.
Returns T if U and V are EQ. In UOLISP, 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 Ø. 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. NUMBERP is the same as FIXP.

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

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

(STRINGP U:any):boolean
Type: EVAL, SPREAD.
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 Ø, returns NIL otherwise.

Since floating point numbers are not implemented, FLOATP is the only Standard LISP function not defined. VECTORP is defined when the vector package is loaded.

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. This message looks like:

***** <xxx> is not a pair for <function>

where <xxx> is the invalid value, and <function> is the name of the function detecting the error.

(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. If there no remaining free dotted-pairs the garbage collector is called automatically. If there are still no remaining pairs, the system halts with the

******* Free Cells Exhausted

(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 (V . 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 and GENSYM's are interned.

(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 UOLISP.

COMPRESS EXPLODE INTERN REMOB

2.6 PROPERTY LIST FUNCTIONS

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

Flags and indicators may clash, consequently care should be taken to avoid occurrences of indicators which have the same name as a flag. Likewise, the implementation of functions and global variables requires that the indicators and 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 the ids of U. Both V and all members of U must be identifiers. No type checking is performed.

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

(REMFLAG U:any-list V:id):NIL
Type: EVAL, SPREAD.
Removes the flag V from the property list of each member of the list U. Both V and all elements of U must be identifiers.

(GET U:id IND:id):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.

(REMPROP U:id IND:id):NIL
Type: EVAL, SPREAD.
Removes the property with indicator IND from the property list of U. Unlike Standard LISP, NIL is always returned.

2.7 FUNCTION DEFINITION

Functions are global entities which are stored on the property list of the (EXPR . xxx) or (FEXPR . xxx) pair. To maintain compatibility with other systems, functions should not be defined with the PUT function.

(DE FNAME:id PARAMS:id-list FN:any):id

Type: NOEVAL, NOSPREAD.
DE defines an EXPR type function named FNAME with the body FN and formal parameter list PARAMS. 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 EXPR is compiled. The name of the defined function is returned.

(DF FNAME:id PARAM:id-list FN:any):id

Type: NOEVAL, NOSPREAD.
DF defines an FEXPR type function named FNAME with the body FN and a single parameter in the list PARAM. 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 and definition. Unlike Standard LISP, UOLISP 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.

(REMD FNAME:id):NIL
Type: EVAL, SPREAD.
Removes the function named FNAME from the set of defined functions. Unlike Standard LISP, NIL is always returned by the REMD function.

UOLISP does not support the MACRO function type. Consequently the DM function is not supported.

2.8 VARIABLES AND BINDINGS

A variable is a place holder for a value 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. UOLISP 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
UOLISP 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.

Retrieval of values of variables occurs when they are evaluated. The following functions declare the global property and implement the assignment operation.

(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.

(PROG2 A:any B:any):any

Type: EVAL, SPREAD.
The two arguments are evaluated in order, and the value of the second is returned.

This (or PROGN) should be used to surround a set of statements (more than one) in a function definition or COND. (Only one statement is normally allowed). PROGN may also be used, but is is slower

(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.

2.10 ERROR HANDLING

(ERROR NUMBER:integer MESSAGE:any)

Type: EVAL, SPREAD.
NUMBER and MESSAGE are passed back to a surrounding ERRORSET (the UOLISP reader has an ERRORSET). MESSAGE is placed in the global variable EMSG!*. The error number becomes the value of the surrounding ERRORSET as well as being placed in the global variable ENUM!*. 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 ENUM!*.

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, else NIL is returned.

```
(DF AND (U)
  (PROG ()
     (COND ((NULL U) (RETURN T)))
LOOP (COND ((NULL (CDR U)) (RETURN (EVAL (CAR U))))
           ((NULL (EVAL (CAR U))) (RETURN NIL)) )
     (SETQ U (CDR U))
     (GO LOOP) ))
```

(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).

```
(DE NOT (U) (EQ U NIL))
```

(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.

```
(DF OR (U)
  (PROG (X)
LOOP (COND ((NULL U) (RETURN NIL))
           ((SETQ X (EVAL (CAR U))) (RETURN X)))
     (SETQ U (CDR U))
     (GO LOOP) ))
```

2.12 ARITHMETIC FUNCTIONS

All arithmetic functions verify that their arguments are numeric before performing operations on them. The single error message:

***** Non-numeric argument

is used by all numeric functions. All integer values are in the range -4096 to +4095.

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

```
(DE ABS (U)
  (COND ((LESSP U Ø) (MINUS U))
        (T U) ))
```

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

```
(DE ADD1 (U) (PLUS2 U 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.

```
(DE DIVIDE (U V)
   (CONS (QUOTIENT U V) (REMAINDER U V)) )
```

(EXPT U:integer V:integer):integer
Type: EVAL, SPREAD.
Returns U raised to the V power. Unlike Standard LISP, negative exponents are not permitted. The function will create incorrect results when the computed value is greater than 4095.

(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.

(MAX [U:integer]):integer
Type: NOEVAL, NOSPREAD.
Returns the largest of the values in U.

(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.

```
(DE MAX2 (U V)
  (COND ((LESSP U V) V)
        (T U)) )
```

(MIN [U:integer]):integer
Type: NOEVAL, NOSPREAD.
Returns the smallest of the values of U.

(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.

```
(DE MIN2 (U V)
  (COND ((GREATERP U V) V)
        (T U)) )
```

(MINUS U:number):number
Type: EVAL, SPREAD.
Returns -U.

```
(DE MINUS (U) (DIFFERENCE Ø U))
```

(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.

```
(DE SUB1 (U) (DIFFERENCE U 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:

FIX FLOAT

2.13 MAP COMPOSITE FUNCTIONS

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

```
(DE MAP (X FN)
  (PROG ()
LOOP (COND ((NULL X) (RETURN NIL))
           (T (PROGN (APPLY FN (LIST X))
                     (SETQ X (CDR X)) )))
     (GO LOOP) ))
```

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

```
(DE MAPC (X FN)
  (PROG ()
LOOP (COND ((NULL X) (RETURN NIL))
           (T (PROGN (APPLY FN (LIST (CAR X)))
                     (SETQ X (CDR X)) )))
     (GO LOOP) ))
```

(MAPCAN X:list FN:function):any

Type: EVAL, SPREAD.
A concatenated list of FN applied to successive CAR elements of X is returned.

```
(DE MAPCAN (X FN)
  (COND ((NULL X) NIL)
        (T (NCONC (APPLY FN (LIST (CAR X)))
                  (MAPCAN (CDR X) FN)))) )
```

(MAPCAR X:list FN:function):any

Type: EVAL, SPREAD.
Returned is a constructed list of FN applied to each CAR of list X.

```
(DE MAPCAR (X FN)
  (COND ((NULL X) NIL)
        (T (CONS (APPLY FN (LIST (CAR X)))
                 (MAPCAR (CDR X) FN) ))) )
```

(MAPCON X:list FN:function):any

Type: EVAL, SPREAD.
Returned is a concatenated list of FN applied to successive CDR segments of X.

```
(DE MAPCON (X FN)
  (COND ((NULL X) NIL)
        (T (NCONC (APPLY FN (LIST X))
                  (MAPCON (CDR X) FN) ))) )
```

(MAPLIST X:list FN:function):any

Type: EVAL, SPREAD.
Returns a constructed list of FN applied to successive CDR segments of X.

```
(DE MAPLIST (X FN)
  (COND ((NULL X) NIL)
        (T (CONS (APPLY FN (LIST X))
                 (MAPLIST (CDR X) FN) ))) )
```

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.

```
(DE APPEND (U V)
  (COND ((NULL U) V)
        (T (CONS (CAR U) (APPEND (CDR U) V))) ))
```

(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 does not detect a poorly formed alist so an invalid construction may be detected by CAR or CDR.

```
(DE ASSOC (U V)
  (COND ((NULL V) NIL)
        ((ATOM (CAR V))
          (ERROR Ø (LIST V "poorly formed ALIST")))
        ((EQUAL U (CAAR V)) (CAR V))
        (T (ASSOC U (CDR V))) ))
```

(ATSOC U:any V:alist):{dotted-pair, NIL}

Type: EVAL, SPREAD.
ATSOC is the same as ASSOC except that the EQN test is used for comparison purposes rather than EQUAL. ATSOC is faster than ASSOC when the items being checked for are identifiers or numbers. ATSOC does not check for a poorly formed alist.

```
(DE ATSOC (U V)
  (COND ((NULL V) NIL)
        ((EQN U (CAAR V)) (CAR V))
        (T (ATSOC U (CDR V))) ))
```

(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 should not be used to define functions.

```
(DE DEFLIST (U IND)
  (COND ((NULL U) NIL)
        (T (CONS
             (PROGN (PUT (CAAR U) IND (CADAR U))
                    (CAAR U))
             (DEFLIST (CDR U) IND))) ))
```

(DELETE U:any V:list):list

Type: EVAL, SPREAD.
Returns V with the first top level occurrence of U removed from it.

```
(DE DELETE (U V)
  (COND ((NULL U) NIL)
        ((EQUAL (CAR V) U) (CDR V))
        (T (CONS (CAR V) (DELETE U (CDR V)))) ))
```

(LENGTH X:any):integer

Type: EVAL, SPREAD.
The top level length of the list X is returned.

```
(DE LENGTH (U)
  (COND ((ATOM U) 0)
        (T (ADD1 (LENGTH (CDR X)))) ))
```

(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.

```
(DE MEMBER (A B)
  (COND ((NULL B) NIL)
        ((EQUAL A (CAR B)) B)
        (T (MEMBER A (CDR B))) ))
```

(MEMQ A:any B:list):extra-boolean

Type: EVAL, SPREAD.
Same as MEMBER but an EQ check is used for comparison.

```
(DE MEMQ (A B)
  (COND ((NULL B) NIL)
        ((EQ A (CAR B)) B)
        (T (MEMQ A (CDR B))) ))
```

(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.

```
(DE NCONC (U V)
  (PROG (W)
     (COND ((NULL U) (RETURN V)))
     (SETQ W U)
LOOP (COND ((CDR W) (PROGN (SETQ W (CDR W))
                           (GO LOOP)) ))
     (RPLACD W V)
     (RETURN U) ))
```

(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. 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.

```
(DE PAIR (U V)
  (COND ((AND U V)
            (CONS (CONS (CAR U) (CAR V))
                  (PAIR (CDR U) (CDR V)) ))
        ((OR U V)
           (ERROR Ø
             "Different length lists in PAIR"))
        (T NIL) ))
```

(REVERSE U:list):list

Type: EVAL, SPREAD.
Returns the top level reversal of the list U (the reversal does not go to all levels). The reversed list is a copy of the actual parameter.

```
(DE REVERSE (U)
  (PROG (W)
LOOP (COND (U (PROGN (SETQ W (CONS (CAR U) W))
                     (SETQ U (CDR U))
                     (GO LOOP) )))
     (RETURN W) ))
```

(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.

```
(DE SUBLIS (X Y)
  (COND ((NULL X) Y)
        (T (PROG (U)
                 (SETQ U (ASSOC Y X))
                 (RETURN (COND
                           (U (CDR U))
                           ((ATOM Y) Y)
                           (T (CONS
                                (SUBLIS X (CAR Y))
                                (SUBLIS X (CDR 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.

```
(DE SUBST (U V W)
  (COND ((NULL W) NIL)
        ((EQUAL V W) U)
        ((ATOM W) W)
        (T (CONS (SUBST U V (CAR W))
                 (SUBST U V (CDR W)) )) ))
```

The following Standard LISP functions are not implemented:

DIGIT LITER SASSOC

2.15 THE INTERPRETER

(APPLY FN:{function-pointer,lambda} 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. FN can be either a function-pointer, or a lambda expression.

APPLY, unfortunately, simply doesn't work with anything other than a LAMBDA expression. Define a new function APPL (or any other name you like), as:

```
(DE APPL (FN VARS)
  (COND ((OR (ATOM FN)
             (NOT (EQ (CAR FN) 'LAMBDA)))
         (EVAL (CONS FN VARS)))
        (T (APPLY FN VARS))))
```

(EVAL U:any):any

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

The standard LISP function LAMBDA is not defined in UO-LISP but can be defined simply by:

```
(DF LAMBDA (L)
  (APPLY ((CONS (QUOTE LAMBDA) L) (READ))))
```

This will define LAMBDA as given in the LISP 1.5 Primer by Weissman

ex: (LAMBDA (X) (PLUS X 5)) (7)
has value
12

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

(FUNCTION FN:function):function
Type: NOEVAL, NOSPREAD.
The function FN is to be passed to another function. If FN is to have side effects its free variables must be GLOBAL. FUNCTION is like QUOTE and, unlike Standard LISP, its argument is not compiled. The FUNARG mechanism is not supported.

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

The Standard LISP function EXPAND is not supported. Macros are not supported at all.

2.16 INPUT AND OUTPUT

The user normally communicates with UOLISP through the standard console device. UOLISP allows input from one disk file at a time and output to another. Special devices are supported as noted in the appropriate installation guides.

(CLOSE FILEHANDLE:number):number
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, UOLISP displays an error but processing will continue.

(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.

(LINELENGTH LEN:{integer, NIL}):integer

Type: EVAL, SPREAD.
If LEN is an integer the maximum line length to be printed before the print functions initiate an automatic TERPRI is set to the value LEN. The initial line length is set to the width of the standard output device. The previous line length is returned except when LEN is NIL. This special case returns the current line length and does not cause it to be reset. An error occurs if the requested line length is less than 0. The maximum line length is 4095. If the line length is set to 0, no automatic TERPRI's will be done.

(POSN):integer

Returns the number of characters in the current output buffer. When the buffer is empty, 0 is returned.

(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. READ ignores comments. A comment starts with a percent sign (%) and is terminated by the end of line.

(READCH):id
Returns the next 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, unless it is a device that is always open (like the console). 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 LPOSN PAGELENGTH 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. The timing ratio of slow to fast links is approximately 50 to 1.

!*GC - Initial value = NIL.
!*GC controls the printing of garbage collector messages. If NIL no indication of garbage collection will 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.

!*RAISE - Value = NIL.
If !*RAISE is T then lower case characters are converted to upper case during input. If NIL, no conversion takes place. On machines which do not normally support lower case, this flag is not implemented.

UOLISP does not implement the Standardard LISP !$EOF!$ variable.

2.18 STANDARD LISP DIFFERENCES

Functions supported by UOLISP but are not in the Standard LISP report are listed in the first two sections of this chapter. The following Standard LISP functions are not currently supported for a variety of reasons:

COMPRESS	FLOATP	PRINC
CxxxxR	FLOAT	REMOB
DIGIT	FLUIDP	SASSOC
DM	FLUID	UNFLUID
EJECT	INTERN	
EXPAND	LITER	
EXPLODE	LPOSN	
FIX	PAGELENGTH	

The vector functions GETV, MKVECT, PUTV, UPBV, and VECTORP are implemented as a package which is interfaced to the RLISP high level language.

2.19 ERRORS

Many error conditions are signaled by a system call to the ERROR function. The following errors and their corresponding numbers are detected in this manner.

1. Caused by a user typing the program interruption key (implementation specific).

2. Undefined function call from compiled code.

3. Not used.

4. The argument of CAR, CDR, RPLACA, RPLACD and so on is not a dotted-pair.

5. An arithmetic function was called with a non-numeric argument.

6. An input or output file could not be opened. This will usually be prefixed by some operating system error message.

7. A poorly formed association list was detected by ASSOC.

8. Not used.

9. Not used.

10. An input or output error was detected from which the operating system could not recover. This will usually be preceeded by an operating system error message.

11. An unbound variable was detected during the evaluation of a function or functional form.

12. The object of a GO could not be found within the current PROG.

2.20 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 frame stack 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)
```

CHAPTER 3

FAST LOAD

Rather than compiling an entire system or reading and compiling code every time, program modules are compiled into relocatable fast load files. Most modern LISP systems provide this facility in one form or another. The fast loading program is 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 a disk file name. 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 "COMP")              %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 PUTDs which are evaluated and cause compiled functions to be dumped to the file.

If a function must be evaluated during the fast load generation process it should be tagged with the EVAL flag by the FLAG function. The system has already flagged the RDS, IN, ON, and OFF functions as EVAL type. Some functions must be both dumped and evaluated. These are tagged with the EVALS flag by FLAG. GLOBAL is the only one one these flagged by the

All GLOBALS must appear together at the beginning of the program

system.

In RLISP the same sequence is accomplished except that RLISP syntax is used in place of S-expressions.

Fast load files are both relocatable and implementation independent. This means that they may be loaded into at any storage location. To some degree files are also machine independent.

CHAPTER 4

THE COMPILER AND OPTIMIZER

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 fast load file for later reloading. An optional third pass optimizes the LAP before assembling it.

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 UOLISP as follows:

```
(DE FACT (N)
   (COND ((LESSP N 2) 1)
         (T (TIMES2 (FACT (SUB1 N)) N))))
```

In UOLISP, dotted-pairs, of which this function is composed, take 4 bytes each. 22 dotted-pairs are used to define FACT for a total of 88 bytes. UOLISP's compiler and optimizer generates the following code for FACT:

```
0000                            ENTRY   FACT EXPR
0000 CD1294                     CALL    ALLOC
0003 02                         DEFB    2
0004 E7FE                       STOX    HL -1
0006 110240                     LDI     DE 2
0009 F7                         RST     LINK
000A 1620                       DEFW    LESSP
000C EF                         RST     CMPNIL
000D 2805                       JREQ    $1
000F 210140                     LDI     HL 1
0012 1813                       JR      $0
0014                        $1:
0014 DFBF                       LDX     HL -1
0016 F7                         RST     LINK
0017 8120                       DEFW    SUB1
0019 F7                         RST     LINK
001A A920                       DEFW    FACT
001C DF7F                       LDX     DE -1
001E F7                         RST     LINK
001F 1921                       DEFW    TIMES2
0021                        $0:
0021 CD8494                     CALL    RDLLOC
0024 FE                         DEFB    -2
(FACT used 37 bytes)
FACT
```

A total of 37 bytes, less than half the size of the interpreted version. The execution of the compiled version uses no dotted-pairs and runs nearly 20 times faster.

4.2 COMPILATION MECHANISMS

Much support software is needed for compiled programs. Compiled programs 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. The routine RDLLOC is called from optimized code. It performs the same functions as DALLOC and in addition does a double return to the function which called the function which called RDLLOC. This saves one byte at the end of most functions.

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.

Since these functions slow down the object code considerably, the optimizer can replace them with their 6 byte indexed move equivalents. This will speed up many functions over 30%.

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) ...		
	(CALL ALLOC)	+--------------+
	(DEFB +4)	|L |<-- new IX
	(STOX HL -1)	+-- A --+
	(STOX DE -2)	|H |
	.	+--------------+
	.	|E |
	.	+-- B --+
	.	|D |
	.	+--------------+
	.	. .<-- old IX
..(PROG (C D) ...		
	(CALL ALLOC)	+--------------+
	(DEFB +4)	|L |<-- new IX
	(LDI HL NIL)	+-- C --+
	(STOX HL -1)	|H |
	(STOX HL -2)	+--------------+
	.	|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 either because it is not yet defined or it is interpreted. 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 the 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 inserting labels and values into a template. RETURN's not in PROGs and illegally nested GO's are not checked.

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

compiles to:

```
        (CALL ALLOC)     ;Space to save variable X allocated.
        (DEFB +2)
        (LDI HL NIL)     ;PROG variable set to NIL.
        (STOX HL -1)
            .
            .
        (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.
        (CALL 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 aØ .. an)
```

compiles to:

```
... evaluate aØ ...
    (RST CMPNIL)     ;Is result of aØ NIL?
    (JPEQ GØØØ1)     ;Stop evaluation if yes.
       .
       .             ;Evaluate other arguments.
       .
... evaluate an ...
    (LABEL GØØØ1)    ;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. The load register instructions automatically add the correct tag bits.

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 fast load file. The optimizing phase implements many more instructions which can be used only when the optimizer is loaded. 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 type)
Serves as the entry point of function "name". ENTRY does not generate any Z80 instructions. It must always be the first instruction in every function as it causes the creation of the code pointer to the first instruction of the function and the definition of the function.

(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, RDLLOC, 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. Different implementations use different sets based on the number of available RST's. Consult the implementation guide to find what set is used. Currently the following appear to be the best choices.

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.
ALLOC - Allocate stack frame.
RDLLOC - Deallocate stack frame and return to caller.

(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 stored as a fast load file. In LISP it must be manually loaded by typing:

```
(FLOAD "COMP")
```

The name of the compiler file varies from system to system. After 30 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:

```
(COMPD 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:

```
(COMPD 'FACT 'EXPR
  '(LAMBDA (N)
     (COND ((LESSP N 2) 1)
           (T (TIMES2 N (FACT (SUB1 N)))))))
```

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

```
(SETQ !*COMP T)
(DE FACT (N)
  (COND ((LESSP N 2) 1)
        (T (TIMES2 N (FACT (SUB1 N))))))
```

will result in the function being compiled before being defined.

Compiling functions into a "fast load" file involves the FSLOUT function. FSLOUT is a special LISP reader which accepts LISP programs, either typed in or from a file and compiles them into the relocatable format. It controls setting of all flags and proper formatting of the file for the FLOAD function. The argument of FSLOUT is a file name. Use of FSLOUT is described in the Fast Load Chapter.

4.4.1 Compiling FEXPR Calls.

When compiling calls to user defined FEXPRs or calls to PLUS, TIMES, MIN, or MAX, the argument list is passed as a list to the FEXPR for evaluation. This interpreted form interacts poorly with compiled code for the following reason. All local variable names declared in a function are replaced with their stack frame locations by the compiler. Thus when the FEXPR tries to evaluate its argument in the environment of the calling routine, the variable names in the S-expression cannot be found. The solution is to declare any variables to be passed to an FEXPR for evaluation as GLOBAL. Note that this need not be done for COND, PROGN, PROG, LIST, OR, and AND because these forms are compiled into object code rather than calls to these functions.

4.4.2 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.

FAPOUT
: When non-NIL, causes the assembler to generate the code for a FAP file. FAPOUT 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. The LAPP printing package causes the LAP code to be displayed in hexadecimal and the text to be formatted.

4.5 THE LISP ASSEMBLY PROGRAM

The Lisp Assembly Program may be called directly with a list of assembly functions by calling:

LAPZ8Ø(NME:id, TYPE:id, LAPS:list):NME
Type: EVAL, SPREAD.
NME is the name of the function to be compiled, TYPE is either EXPR or FEXPR. LAPS is the list of LAP instructions to be assembled. 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.

4.5.1 Augmenting LAP

To augment the LAP assembler perform the following steps:

1. On the property list of the name of the instruction with the indicator BCNT place the number of bytes used by the instruction.

2. Create a function with the name of the instruction. This function should have arguments which correspond to the operands of the instruction being defined. The function should return a list of integers which represent the bytes of the instruction being generated.

As an example consider adding an OUT instruction. This instruction has no arguments. In the HL register pair should be the device address to send the last 8 bits of register pair DE to. The Z8Ø code sequence generated is (in TDL mnemonics):

```
MOV   C,L
OUTP  E
```

Inclusion of this instruction would permit the LISP user to implement by hand output to an arbitrary device. The LISP to implement this instruction would then be:

```
(PUT 'OUT 'BCNT 3)
(DE OUT ()
   (LIST3 77 58 1Ø5))
```

To get use of this instruction you must hand code a function in LAP and pass it to the LAPZ8Ø assembler. For example, the !$OUT function below has two numeric arguments, HL is the device number to send the second argument, the last 8 bits of DE to.

```
(LAPZ80 '!$OUT 'EXPR
  '((ENTRY !$OUT EXPR) (OUT) (RET)))
```

The output from the LAPZ80 program with the LAPP global set to T is:

```
(0 NIL (ENTRY !$OUT EXPR))
(0 (77 78 105) (OUT))
(3 (201) (RET))
(!$OUT USED 4 BYTES)
!$OUT
```

The best way to learn how to add functions to the compiler and the LAP assembler is to set LAPP to T and watch the output for a number of functions.

4.5.2 LAP Support Routines.

A number of routines support the operation of the assembler.

1. MKREF - Returns a list of two integers corresponding to the high and low order bytes of the item passed to it.

2. MKGLOB - Returns a list of two integers corresponding to the address of a global variable. To use this function do:

   ```
   (MKGLOB (GGET name 'GLOBAL))
   ```

3. FAPABS - Given a list of integers of an instruction, if the fast load output switch is on, causes the bytes to be dumped to the output file. If not they are returned as is. This function must be used if fast load files are to be generated. The OUT function above would then be coded:

   ```
   (DE OUT () (FAPABS (LIST3 77 58 105)))
   ```

4. FAPQUO - Used to output quoted items to fast load files (do not use FAPABS for this).

4.5.3 The LAPP Printing Package.

The LAPP package replaces the simple minded LAP printing (just list notation) provided with LAP. All numbers are printed in hexadecimal and there is some attempt to format the LAP instructions in conventional format. The assembly listing of FACT at the beginning of the compiler section is an example of its output. The package must be loaded with FLOAD in the usual

fashion. Output is enabled by setting the LAPP variable to T.

4.5.4 A Demonstration Of LAP.

The following code demonstrates some of the methods and possibilities of using LAP (RLISP syntax).

```
%
% Define some functions for intructions not part of LAP.
%

DEFLIST(
  '((B 0)(C 1)(D 2)(E 3)(H 4)(L 5)(M 6)(A 7)),
  'RR);
EXPR PROCEDURE MOV(R, RP);
%Generate MOV R,RP where R and RP are A,B,C,D,E,H,L, and M.
FAPABS LIST(64 + 8*GET(R, 'RR) + GET(RP, 'RR));
PUT('MOVRR, 'BCNT, 1);

EXPR PROCEDURE ANI X;
%Generate an AND immediate instruction with X as its value.
FAPABS LIST(230, X);
PUT('ANI, 'BCNT, 2);

EXPR PROCEDURE ORI X;
%Generate an OR immediate instruction with X as its value.
FAPABS LIST(246, X);
PUT('ORI, 'BCNT, 2);

DEFLIST('((BC 0) (DE 16) (HL 32) (SP 48)), 'SS);
EXPR PROCEDURE DAD RR;
%Generate a double add to HL instruction from register
%pair RR.
FAPABS LIST(9 + GET(RR, 'SS));
PUT('DAD, 'BCNT, 1);


%
% Now do a fast PLUS2 without any type checking.
%
LAPZ80('PLUS2!*, 'EXPR,
 '((ENTRY PLUS2!* EXPR)
   (DAD DE)        % Do the addition, screw up tags.
   (MOV A H)       % Make first 3 bits into numeric tag.
   (ANI 31)        % Clear garbage created by DAD.
   (ORI 64)        % Creates numeric tag in first 3 bits.
   (MOV H A)       % HL is numeric type now.
   (RET) ));       % Return to caller.
```

```
EXPR PROCEDURE SLAR RR;
% Generate an SLAR instruction for register RR.
FAPABS LIST(203, 32 + GET(RR, 'RR));
PUT('SLAR, 'BCNT, 2);

EXPR PROCEDURE RALR RR;
% Generate a RALR instruction for register RR.
FAPABS LIST(203, 16 + GET(RR, 'RR));
PUT('RALR, 'BCNT, 2);

EXPR PROCEDURE DJNZ LBL;
%Generate a short DJNZ instruction to LBL. Assume that LBL is
%defined. LALST is the global association list of labels and
%locations (relative to base of function). LOC is the  current
%instruction location.
FAPABS LIST(16, (CDR ATSOC(LBL, LALST) - LOC) - 2);
PUT('DJNZ, 'BCNT, 2);

%
% SHIFT!-LEFT(A, B) -
%     Shift the quantity in A left B number of times.
LAPZ80('SHIFT!-LEFT, 'EXPR,
 '((ENTRY SHIFT!-LEFT EXPR)
   (MOV B E)          %B gets the number of shifts to do.
   (LABEL LOOP)       %Label for loop of shifting.
   (SLAR L)           %Shift right byte left to carry.
   (RALR H)           %Shift left byte left,
                      %  carry into low order.
   (DJNZ LOOP)        %Continue until all shifted.
   (MOV A H)          %Restore numeric tag.
   (ANI 31)           %Clear garbage shifted in.
   (ORI 64)           %New numeric tag created.
   (MOV H A)
   (RET) ));          %Return to caller.
```

4.6 THE OPTIMIZER

The code produced by the UOLISP compiler is generally large and inefficient. The optimizer is an additional phase for the compiler to increase the efficiency of generated code. It does this by:

1. Removing useless instructions.

2. Converting long jumps into short ones wherever possible.

3. Inverting conditionals wherever possible.

4. Open coding some arithmetic functions.

5. Folding stack frame allocations and deallocations and removing them when not needed.

6. Substituting special Z80 instructions that are shorter and faster than standard LAP forms.

Some of the optimizations produce smaller code, some faster but larger. In general only a few functions in a program need to be very fast. According to the maxim: "90% of the time is spent in 10% of the code". The optimizations which generate large but fast code can be turned on and off at the users discretion.

4.6.1 Space Optimizations

The following optimizations produce short code and do not in any way change the semantics of execution.

1. DALLOC - If several DALLOC calls occur in a row, they are combined into a single call. Thus:

```
CALL  DALLOC        becomes        CALL  DALLOC
DEFB  -4                           DEFB  -10
CALL  DALLOC
DEFB  -6
```

   Secondly, if the folded DALLOC is directly followed by a RET instruction, the RET is removed and the RDLLOC routine is called. This routine incorporates the RET saving both time and space.

```
CALL  DALLOC        becomes        CALL  RDLLOC
DEFB  -10                          DEFB  -10
RET
```

2. LDX, STOX - This set of optimizations tries to remove extra LDX and STOX instructions. Whenever an LDX or STOX instruction is encountered, the optimizer scans ahead across instructions which do not modify the associated register. If it encounters an LDX instruction which loads the same quantity that is already in the register this extra LDX is removed. For example:

```
CALL  ALLOC        becomes        CALL  ALLOC
DEFB  +2                          DEFB  +2
STOX  HL -1                      *STOX  HL -1
LDX   HL -1                       RST   CAR
RST   CAR                         RST   CDR
RST   CDR                         CALL  DALLOC
CALL  DALLOC                      DEFB  -2
DEFB  -2                          RET
RET
```

Since the LDX reloads what is already in HL, it is removed. A second optimization removes STOX instructions whose value is never loaded by an LDX instruction. Thus the STOX instruction in the optimized version (marked with an *) is also removed.

3. Z80 instructions - Two optimizations changing long forms into shorter instructions are performed. One changes the 4 byte load global variable into register into its three byte version if the register is HL. A second optimization converts the move HL to DE instruction (two single byte register moves) into a single XCHG instruction. Each of these optimizations saves one byte. Thus:

```
LDD   HL global    becomes        LHLD  global
LDHL  DE                          XCHG
```

4. Dead Code - Two cases of dead code can occur. The first of these is code following a JP or JR instruction. The second is code following a function which never returns a value. ERROR and THROW are two functions which never return. Code following these forms up to but not including a LABEL instruction is removed. If the function epilog is removed by this process, the function is added to the list of those which do not return a value. Thus:

```
JP    $0           becomes        JP    $0
JP    $1                          LABEL $2
LABEL $2
```

5. Forward Jump - If the destination of a jump instruction is the next instruction it is removed.

6. Short Jumps - If the destination of a jump instruction is less than 127 bytes from the current location, it is replaced by its short form. Though this does not speed up the code it is by far the most commonly occurring optimization and results in the most significant saving of storage.

7. Jump Inversion - Many COND tests have multiple jumps which can be combined by inverting one of the tests. Thus:

```
JPEQ  $Ø          becomes        JPNEQ $1
JP    $1                       LABEL $Ø
LABEL $Ø
```

8. Double Negation - The method of converting T/NIL into a condition code will occaisionally introduce a double negation followed by a conditional jump instruction. The jump instruction is inverted and the call to NOT or NULL is removed. Thus:

```
RST   LINK        becomes        RST   CMPNIL
DEFW  NULL                       JPNEQ $Ø
RST   CMPNIL
JPEQ  $Ø
```

9. Frame Removal - If by removal of STOX and LDX instructions there are no stack frame references left in a function, the ALLOC and DALLOC (or RDLLOC) frame allocation calls are removed. A previous example first has its STOX and LDX removed and then is optimized as follows:

```
CALL  ALLOC       becomes        RST   CAR
DEFB  +2                         RST   CDR
RST   CAR                        RET
RST   CDR
CALL  RDLLOC
DEFB  -2
```

This optimization saves a considerable amount of space and greatly speeds up the execution of small functions.

10. C..R reduction - On machines which have short calls to CAR and CDR, the calls CAAR, CADR, CDAR, and CDDR are replaced with two short form calls to CAR and CDR saving one byte. Thus:

```
RST   LINK        becomes        RST   CDR
DEFW  CADR                       RST   CAR
```

4.6.2 Fast Optimizations.

There are a few optimizations which increase the code size but greatly increase the execution speed of the program. These do not alter the semantics of execution.

1. Deallocate Frame - This optimization replaces a call to the DALLOC routine with code to decrement the stack frame register (IX). Thus:

```
CALL  DALLOC        becomes        LXI   DE -10
DEFB  -10                          DADX  DE
RET                                RET
```

This is at the expense of one byte, but it greatly decreases execution time.

2. Stack Frame Reference - This optimization replaces the LDX/STOX function calls with 2 indexed load/store instructions. This costs 4 more bytes but runs about 10 times faster than the subroutine call. Code speed ups of 30% or more are possible with this optimization.

```
LDX   HL -1         becomes        MOV   L -1(X)
                                   MOV   H 0(X)

STOX  DE -2         becomes        MOV   -3(X) E
                                   MOV   -2(X) D
```

4.6.3 Dangerous Optimizations

This class of optimizations introduces some changes in the semantics of functions. In particular, some of the safety checks are disabled. These should be enabled only after the operation of the functions being optimized has been verified to be correct.

1. Frame Allocation - The stack overflow check is removed and the allocation of the frame is open coded. There are two forms used determined by the size of the frame.

```
CALL  ALLOC         becomes        INX   X
DEFB  +2                           INX   X

CALL  ALLOC         becomes        EXX
DEFB  +10                          LXI   D +10
                                   DADX  D
                                   EXX
```

The only problem with this optimization is recursion to a great depth. It is possible that the frame stack might cross the system stack and result in a system crash.

2. ADD1/SUB1 - This optimization replaces calls to ADD1 and SUB1 with register increment and decrement instructions. The optimization disables the check for correct type. In addition, a change of sign will cause the type of the value to change to something else. This optimization can be used only if the sign

of the value will never change.

```
RST   LINK        becomes       INX   HL
DEFW  ADD1

RST   LINK        becomes       DCX   HL
DEFW  SUB1
```

4.6.4 Use Of The Optimizer

The optimizer works in conjunction with the compiler and LAP. To enable the optimizer and the compiler from LISP enter:

```
(FLOAD "COMP") (FLOAD "OPT")
(SETQ !*COMP (SETQ !*OPT T))
```

At any time optimization can be enabled and disabled by setting !*OPT to T or NIL.

The two other levels of optimization are controlled by the flags !*FAST and !*DANGER. Either or both of these may be enabled.

CHAPTER 5

THE LISP EDITOR

The LISP editor is a set of functions which enable the user to:

1. Enter and test functions while remaining in the LISP system.

2. Save and restore sets of functions and commands from disk.

3. Modify functions and test them before saving.

The editor must first be loaded into storage using the fast loading program. It may coexist with any of the other packages with the following exceptions:

1. Compiled functions may not be edited. A function which is edited and then compiled cannot be written back to disk.

2. Functions entered in RLISP syntax must be edited in LISP S-expression format.

5.1 OPERATION

The editor must first be loaded from its fast load file. Operation is not automatic. Functions must be defined by using the CREATE and CREATEF functions. Output of the editor is in standard PRINT format unless the PRETTY pretty print package is also loaded. In this case, all output from the editor is through the pretty printer.

Using a Word-processor to develop LISP programs.

Just type the functions into your WP program, and save them out - as an ASCII file. In LISP, do a

(OPEN "yourtextfilename")

You will get a number. Do

(RDS number)

There they are!

Now, if you made a mistake, you can go edit your file easily.

5.2 EDITOR COMMANDS

The commands of the editor are for the most part top level function calls. They are listed alphabetically here.

(CREATE function-name)

Execution of this command causes the creation of an EXPR type function with the name "function-name". The user will be prompted for the argument list and the body of the function. CREATE causes the function to be added to a list of functions which can be edited and later saved on disk. The equivalent of:

```
(DE FACT (N)
   (COND ((LESSP N 2) 1)
         (T (TIMES2 N (FACT (SUB1 N))))  ))
```

is the following sequence (machine output is underlined):

```
(CREATE FACT)
ARGUMENTS: (N)
BODY: (COND ((LESSP N 2) 1)
            (T (TIMES2 N (FACT (SUB1 N))))  )
```

(CREATEF function-name)

This is the same function as CREATE except that an FEXPR is created instead of an EXPR.

(DEFINE S-expression)

This function causes the S-expression to be added to the list of items which can be saved on disk. In addition, the S-expression is evaluated for its effect. Thus the expression:

```
(DEFINE (GLOBAL '(X)))
```

will cause X to be declared as a global variable and the declaration will be written to the disk file during a disk SAVE.

(EDIT function-name)

This function permits the editing of the function named provided it is not a compiled function. Commands to the editor are normally single characters or digits sometimes followed by S-expressions. The commands are:

1. nH - Examine the head (CAR portion) of the currently displayed expression. If n is present, scan down n CAR portions.

2. nT -Examine the tail (CDR portion) of the currently displayed expression. If n is present scan down n CDR portions.

3. n^ - Backup to the structure being examined previously. If this is at the top level, the function is redefined. If n is present, backup n times.

4. I S-expression - Insert the S-expression onto the front of the expression currently being examined. This operation is equivalent to

   (APPEND (LIST S-expression) current-expression)

5. A S-expression - Append the S-expression to the tail of the expression currently being examined. This is equivalent to:

   (APPEND current-expression (LIST S-expression))

6. F S-expression - Find all occurrences of S-expression in the current expression for examination and possible replacement.

7. R S-expression - Replace the expression currently being examined with the new S-expression.

8. C S-expression - Replace the CAR portion of the expression currently being examined with the new S-expression.

9. D - Remove the head (CAR portion) of the current expression. This is equivalent to:

   (SETQ current-expression (CDR current-expression))

(EDITDEF id)

This function permits editing of clauses entered using the DEFINE function. Here, id is the head of one of these clauses. EDITDEF will display each definition with id as a head. The user responds with an N to each definition until the clause to be edited is reached and then responds with Y. EDITDEF then invokes EDIT to modify this clause.

RESTRE

(~~RSTR~~ file-name)

Executing this function causes the file "file-name" to be read from the disk and added to the list of editable functions. As each function is read in, its name is displayed. The file must be terminated with an END. If ~~RSTR~~ RESTRE does not display the messge:

"** COMPLETED **"

then enter an END from the keyboard. If there are errors in the list, the loading process will terminated without completing. The editor will only let you save a file without errors in it.

(SAVE file-name)
Execution of this function will cause all functions and definitions created using DEFINE, CREATE, and CREATEF to be stored in the file "file-name".

5.3 EXAMPLE EDIT SESSION.

The following session llustrates some of the methods used in the definition of a function, some global quantities, and patching an improper function. Output from LISP is underlined.

```
(FLOAD "EDIT")                  % Load the editor.
NIL                             % Proper response when loaded.

(DEFINE (GLOBAL '(X)))          % Define a global variable X.
NIL                             % Proper response to definition.

(DEFINE (SETQ X T))             % Define setting X to T.
T                               % X is set to T and (SETQ X T)
                                % is saved.
(CREATE SUBLS)                  % Create the function SUBLS
ARGUMENTS*(A Y)                 % Define its arguments.
BODY*                           % Now define its body.
(COND ((NULL X) Y)              % Here are the lines of the body.
  (T (PROG (U)
       (SETQ U (ASSOC Y X))
       (RETURN (COND (U (CDR U))
                     ((ATOM Y) Y)
                     (T (CONS (SUBLS X (CAR Y))
                              (SUBLS X (CDR Y)))) )))))
SUBLS                           % SUBLS is now defined.

*(SUBLS '((THIS . THAT) (A . B)) '(THIS IS A))
***** (T IS NOT A PAIR FOR CAR)
      % SUBLS doesn't work because X is a GLOBAL.

*(EDIT SUBLS)                   % Edit the definition of SUBLS.
(EXPR LAMBDA (A Y) (COND ((NULL ...
*F X                            % Look for all X's.
X                               % An X was found.
*R A                            % Change an X to an A.
*^                              % Find the next X.
X                               % Found another X.
*R A                            % Change X to an A.
```

```
.
.       % Find all X's and replace with A.
.
*^                              % Back out of editor and
(SUBLS REDFINED)                % redefine SUBLS.
SUBLS

*(SUBLS '((THIS . THAT) (A . B)) '(THIS IS A))
(THAT IS B)                     % SUBLS now works.

*(SAVE "file")                  % Save the function SUBLS and
                                % the global X.
```

The file "file" now contains:

```
(GLOBAL (QUOTE X))
(SETQ X T)
(DE SUBLS (A Y) (COND ((NULL A) Y) (T (PROG (U) (SETQ
U (ASSOC Y A)) (RETURN (COND (U (CDR U)) ((ATOM Y) Y) (
T (CONS (SUBLS A (CAR Y)) (SUBLS A (CDR Y)))))))))))
FSLEND
```

CHAPTER 6

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 the symbolic algebra system, REDUCE [2]. The subset described here is reasonably complete and is restricted only by the subset of Standard LISP implemented in UOLISP. Users should note that there are significant differences between the RLISP supported here and that used to support the REDUCE system. Users interested in REDUCE should consult reference [2].

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.

6.1 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 | FEXPR

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

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

A <function> is a PROCEDURE statement preceded by its type. 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.

Example procedure prototype lines are given without their bodies.

```
EXPR PROCEDURE NOARGS();
EXPR PROCEDURE ONEARG SARG;
EXPR PROCEDURE MANYARGS(AØ, A1, A2);
FEXPR PROCEDURE DOFEXPR X;
```

6.2 STATEMENTS

There are several different statement types in RLISP corresponding to the different control constructs.

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.

```
BEGIN
  A := A + 1;              % A, B, and C are globals.
  B := C                   % Last statement has no ;
END;

BEGIN SCALAR X, Y;         % X and Y are local variables.
  X := A + 1;
  PRINT (Y := X - 5)
END;
```

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> |
   <REPEAT statement> |
   <FOR statement> |
   <RETURN statement> |
   <GO TO statement> |
   <ON/OFF statement> |
   <IN/OUT/SHUT 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.

```
IF A < 10 THEN PRINT A;

IF ATOM A THEN A
   ELSE REV CDR A .  REV CAR A;
```

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.

```
EXPR PROCEDURE PRINT2 X; % A different PRINT.
<< PRIN2 X; TERPRI(); X >>;
```

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.

```
WHILE X DO
   << PRINT CAR X;
      X := CDR X >>
```

11. <REPEAT statement> ::=
REPEAT <unlabeled statement>
UNTIL <unlabeled statement>

The REPEAT ... UNTIL ... statement mirrors the WHILE ... DO ... construct except that the test for loop termination occurs at the end of the loop rather than the beginning. The value of REPEAT ... UNTIL ... is NIL unless there is a top level RETURN present.

```
REPEAT << PRIN2 CAR X;
          X := CDR X;
          IF X THEN PRIN2 ", " >> UNTIL NULL X;
```

12. <RETURN statement> ::= RETURN <unlabeled statement>

RETURN may be used only within a BEGIN - END block and is translated directly into the regular RETURN function call.

13. <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.

14. <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.

```
FOR EACH X IN '(A B C) DO PRINT X;

FOR EACH X IN '(A B C) COLLECT ATOM X;

FOR I:=1:10 DO PRINT I;
```

15. <ON/OFF statement> ::=
    ON <id-list> |
    OFF <id-list>

These two functions set global variables to T and NIL respectively. ON and OFF work directly on global variables or on variables which have an ON property. If the value of the ON property is an identifier, this identifier is set to T or NIL. This permits setting of internal variable names with !* as in RLISP. If the ON property is a list, there should be three elements. The first is the name of the variable to assign T or NIL to. If this variable is NIL, no assignment is done. The second element is the name of a function. If this function is not present the file name in the third element position is loaded by the fast loader.

name	internal	associated file / package
COMP	!*COMP	"COMP" - compiler.
OPT	!*OPT	"OPT" - optimizer.
DEFN	!*DEFN	
ECHO	!*ECHO	
FLINK	!*FLINK	
OUTPUT	!*OUTPUT	
RAISE	!*RAISE	
GC	!*GC	
PRETTY	!*PRETTY	"PRETTY" - pretty printer.
VECTORS		"VECTORS" - vector package.

16. <IN/OUT/SHUT statement> ::=
    IN "file-name" |
    OUT "file-name" |
    SHUT "file-name"

These 3 statements perform abbreviated versions of RDS, WRS, and CLOSE. The IN statement opens the file name given for input and selects that file as the standard input device. An error will be given if no such file exists. Files may be chained together if the last statement in the file is an IN but may not be nested as in regular RLISP since only one disk file may be open at a time. The OUT statement opens the file name given for output and assigns the standard output device to be this file. An error occurs if there already is a file by this name on the disk. All subsequent output is directed to this file. The SHUT statement closes either an input file selected by IN, or an output file selected by OUT.

6.3 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

standard infix with the exception that function calls with single arguments need not have the arguments enclosed in parentheses, the . operator for CONS, and the ' for QUOTE.

17. <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).

18. <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.

19. <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 ...).

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

21. <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.

22. <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.

23. <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.

24. <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.

25. <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.

26. <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.

27. <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.

6.4 SYSTEM FLAGS

For the most part the RLISP reader works exactly like the UOLISP 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.

!*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.

6.5 ERROR MESSAGES

The RLISP parser implemented for UOLISP 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. 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.

***** 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.

***** Missing IN
In a FOR EACH statement, the noise word IN is missing. The correct format is: FOR EACH <variable> IN <expression> ...

***** Missing DO/COLLECT
In a FOR EACH statement, either of the keywords DO or COLLECT is missing. The correct format is: FOR EACH <variable> IN <expression> DO ... or FOR EACH <variable> IN <expression> COLLECT ...

***** Missing id
The identifier in a FOR EACH, or iterative FOR statement cannot be found after the EACH or the FOR.

***** Missing :
The colon in an iterative FOR statement cannot be located this error occurs. The syntax of RLISP requires : rather than TO as one might expect.

***** 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.

6.6 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.

```
(FLOAD "RLISP")

(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:

```
LISP;
```

to which the system should immediately respond:

```
ENTERING LISP ...
```

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

6.7 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) ))
```

CHAPTER 7

THE TRACE PACKAGE

A rudimentary trace package permits monitoring the entrance to and exit from functions. The trace package must first be loaded:

```
(FLOAD "TRACE")
```

To trace a particular function or set of functions enter:

```
(TR f1 f2 ... fn)
```

where f1 ... fn are the functions to be traced. During the evaluation of these functions, just before each function is evaluated, its name and arguments will be displayed on the currently selected output device. Just before the function exits, its name and value are displayed.

If the function to be traced is a compiled or system defined function, the TR function will ask for the number of its arguments.

```
NUMBER OF ARGUMENTS FOR fn*
```

You should then enter Ø, 1, 2, or 3. Remember that compiled functions can have no more than 3 arguments.

To remove the trace property of a function or functions enter:

```
(UNTR f1 f2 ... fn)
```

The trace information will no longer be displayed with each function. The UNTR will try and verify that the functions named have been traced (it is possible to fool it). A message will appear if the functions ae not traced.

A BREAK function is implemented. This function is particularly useful for stopping the evaluation of a function and for examining the contents of variables on the stack and global variables. The function takes one argument, the value of which is printed when BREAK is entered. At this time BREAK

goes into a READ - EVAL loop similar to the top level UOLISP read - eval loop. The syntax is always LISP even if RLISP is loaded. If the function which has BREAK in it is not compiled, you may display the values of any of its local variables and even modify them. Likewise any global variables may be listed and modified. You may even define functions. When you are ready to resume execution you type EXIT. Note that errors made during a BREAK do not cause you to return to the main read loop.

7.1 IMPLEMENTATION

Tracing a function is accomplished by embedding the definition of the function in a new function with the name of the old one. The old definition is hidden away with a GENSYM name. The new function has the same number of arguments as the old and the code to print all information upon entering and exiting the function. The UNTR function locates the hidden name of the function and redefines it under its real name, the trace code then disappears.

7.2 INTERACTION WITH THE SYSTEM.

Nearly all functions may be traced but there are a number of interactions with the interpreter and compiler wich must be explained.

1. Fast link function calls which have been converted from slow links cannot be traced. If a function is compiled and then executed with !*FLINK = T it cannot be reliably traced afterwards. Any call which is converted to a fast call will bypass the trace code while those which have not been converted will reach the trace code. This leads to arbitrary results.

2. Function calls within the interpreter are not traceable because they are all fast links to start with.

3. Precompiled "fast load" files are traceable provided that the !*FLINK flag is set to NIL before the functions in it are evaluated.

4. Once a compiled function has the trace code wrapped around it, the !*FLINK flag may be set to T to speed up execution. Since the defined function is now interpreted (TR sets the !*COMP flag to NIL before embedding the function to be traced) slow links will not be converted to fast ones.

7.3 EXAMPLE

The following is an example function which is compiled, and then traced during its execution. Finally the trace is removed and the function is executed again. Input provided by user is underlined.

```
(FLOAD "TRACE")  (FLOAD "COMP")
NIL

NIL

(DE FACT (N)
  (COND ((LESSP N 2) 1)
        (T (TIMES2 N (FACT (SUB1 N))))))
(FACT USED 46 BYTES)
FACT

(SETQ !*FLINK NIL)
NIL

(TR FACT)
HOW MANY ARGUMENTS FOR FACT*1
(FACT REDEFINED)
T

(FACT 3)
("ENTERING " FACT (3))
("ENTERING " FACT (2))
("ENTERING " FACT (1))
("LEAVING " FACT 1)
("LEAVING " FACT 2)
("LEAVING " FACT 6)
6

(UNTR FACT)
(FACT REDEFINED)
T

(FACT 6)
720

(DE TEST1 (X Y)
   (PROG (A)
     (SETQ A 12)
     (BREAK "HELLO FROM TEST1:")
     (PRINT (LIST A X Y))
     (RETURN A) ))
TEST1

(TEST1 "STRING" 'IDENTIFIER)
BREAK AT: HELLO FROM TEST1:
*A
12
*(SETQ A 34)
```

```
34
*EXIT
(34 "STRING" IDENTIFIER)
34
```

CHAPTER 8

MISCELLANEOUS PACKAGES

8.1 THE VECTOR PACKAGE

This package supports the complete vector operations designated by Standard LISP including vector input and output.

8.1.1 Functions

(GETV V:vector INDEX:integer):any
Type: EVAL, SPREAD.
Returns the value stored at position INDEX of vector V. An error occurs if INDEX does not lie within 0 .. (UPBV V) inclusive.

***** INDEX subscript out of range

(MKVECT UPLIM:integer):vector
Type: EVAL, SPREAD.
Defines and allocates a vector with UPLIM + 1 elements accessed as 0..UPLIM. Each element is initialized to NIL. An error will occur if UPLIM is less than 0.

***** (UPLIM invalid vector size)

(PUTV V:vector INDEX:integer VALUE:any):any
Type: EVAL, SPREAD.
PUTV stores VALUE into the vector V at position INDEX. VALUE is returned. If INDEX does not line in the range 0..(UPBV V) an error occurs:

***** INDEX subscript out of range

To use these, you must
(DE LEQ (X Y) (OR (LESSP X Y) (EQ X Y)))
(DE NEQ (X Y) (NOT (EQ X Y)))

(UPBV U:any):{NIL,integer}
Type: EVAL, SPREAD.
Returns the upper limit of U if U is a vector, or NIL if it is not.

(VECTORP U:any):boolean
Type: EVAL, SPREAD.
Returns T if U is a vector and NIL if not.

8.1.2 Implementation

Vectors are implemented as lists. Consequently, linear search is used to access vector locations. The list structure of vectors is always:

```
(!$vector!$ nnn v[0] ... v[nnn])
```

The !$vector!$ tag is actually an FEXPR type function which returns itself and arguments without evaluation. In this way vectors are treated like constants. The elements of a vector can be of any type and can even be of mixed types. Vectors of vectors are even possible.

8.1.3 Input

Vectors may be read from RLISP source code in the format of the standard LISP report only if the host computer supports square brackets. Thus:

```
[1, 2, "Here a string", "There a string"]
[[1, 2], [2, 3], [3, 4]]
```

are possible. On machines without square brackets the left square bracket is @ and the right square bracket is #. Output is similar to input.

If vector input is not needed, vectors may be used from RLISP by loading the vector package "VECTORS". In RLISP the vector package is enabled by entering ON VECTORS;

8.2 PRETTY PRINTING

The pretty printing package attempts to format LISP S-expressions by indenting them and keeping them within the boundaries specified by LINELENGTH. The pretty print package can be loaded directly by entering:

```
(FLOAD "PRETTY")
```

or in RLISP:

```
ON PRETTY;
```

The pretty printer is interfaced to RLISP and the structure editor. In RLISP, the result of each evaluation will be pretty printed whenever the package is loaded. In the editor, all output is pretty printed.

The interface function is PRETTYPRINT.

(PRETTYPRINT U:any):any
Type: EVAL, SPREAD.
Pretty prints the S-expression U within the boundaries set by the LINELENGTH function. The value of U is returned.

8.3 LAPP MODULE

This small package is an extra for the compiler and does a better job of displaying the result of the LAP assembler than the smaller resident version. All values are displayed in hexadecimal rather than decimal. Output is formatted into columns for easier reading. The module is loaded by:

```
(FLOAD "LAPP")
```

The module is automatically enabled without further communication.

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

3. Allen, J., "Anatomy of LISP", McGraw Hill, New York.

4. Winston, P. H., Horn, B. K. P, "LISP", Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

5. Weissman, Clark, "LISP 1.5 Primer", Dickenson Publishing Company, Belmont, California, 1967.

6. Siklossy, Laurent, "Lets Talk LISP", Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

BEST→ Touretsky, David S. LISP: A Gentle Introduction to Symbolic Computation.