ORICLISP Reference Manual

Contents

I. Introducing LISP

The LISP language was created in 1964 by John McCarthy in order to process information organised as binary trees (math formulas, natural language sentences, knowledge bases etc.). LISP represents those trees as lists (by the way, LISP is an abbreviation for LISt Processing).

The ability of LISP to handle symbolic information has made it one of the preferred "machine languages" in such fields as Artificial Intelligence, Robotics, Formal Calculation, Programming Theory (e.g. program proving), Game Theory, Syntax Editors (e.g. Emacs) and more generally Programming Environments, etc.

To position LISP relative to other programming languages, a distinction is usually made between three language families (although some languages may belong to more than one family): imperative languages such as Fortran, Pascal, Cobol, Basic, Ada, C..., in which a sequence of instructions is used to reach a solution; declarative languages such as Prolog which merely describe the problem to be solved (a deductive logic is then used to reach the solution); and functional and application languages such as LISP, ML, FP, where you apply functions in the mathematical sense (i.e. there are no more "variables" in a program). Early object-oriented languages such as SmallTalk may be regarded as being derived from LISP. LISP also influenced the LOGO language.

Apart from this classification, LISP is comprised of a whole family of dialects (InterLisp, MacLisp, VLisp, LeLisp, MuLisp, FranzLisp...), which all share the following features:

- They compound recursive functions whereas traditional (imperative) languages repeat assignment sequences.
- They handle tree structures, and there are program trees as well as data trees (a program may therefore be processed by another program!)
- They are structured and modular (thanks to compound functions).
- They are interactive (they generally provide a language interpreter which lets you modify everything, including the interpreter itself!).
- Memory is managed dynamically and automatically (a "garbage collector" reclaims the storage space used by data which is no longer needed).

I.1 Syntax

OK, a LISP program is a binary tree, but how do you represent it as a sequence of ASCII characters? Of course, you can't use a diagram, therefore a notation using parentheses, spaces and dots has been selected to represent those trees. Once you've noticed that the first two branches of a binary tree are in turn trees, you will naturally accept the following notation, called a "dotted pair": (S1 . S2) where S1 and S2 are either dotted pairs or tree leaves called "atoms".

LISP atoms are basic components, just like in physics. There are two kinds of atoms: integer numbers (also sometimes real numbers) and "symbols" which are used as identifiers for abstract data. For example, two symbols are used to represent the classical Boolean values: NIL for "False" and T for "True" (although everything that is not NIL is in fact "True"). NIL is also used for "Null", i.e. to designate an empty list...

Talking of "lists", note that lists are particular trees (trees are called "S-expressions" in LISP). The left-hand side of a dotted pair has been chosen to hold the first item of a list, whilst the right-hand side points to the rest of the list. For example, a fruit list could be written as follows, using dotted pairs:

```
( apple . ( pear . ( apricot . nil ) ) )
```

Such lists are so common that the above notation can be abbreviated as:

```
( apple pear apricot . nil )
```

This simplified notation gives a special meaning to space separators inside parentheses. The notation can be further shortened by implying that the NIL symbol is the end-of-list marker, e.g.:

```
( apple pear apricot )
```

This is much simpler! However, you can still have more complex notations if you remember that, because a list is an S-expression, list items can be lists in turn... For example:

```
( marc (age 20) (children (lucy (age 2)) (allan (age 1)) ) )
```

is a list with 3 items. The first item is an atom; the second one is a list of two atoms; and the third one is a list comprised of one atom and two lists... Now you understand why it is so important to match parentheses properly in LISP!

I.2 Interpreter

What? We start a new section without having even mentioned the syntax of LISP programs! Well, this is because, if you can build lists, then you can write programs. A LISP program is just an S-expression or, should I say, an atom or a list, that will be "evaluated" by the interpreter. For example, if the interpreter comes across a number to be evaluated, it will return the same number (that one was easy...). If it comes across a symbol, it will return the "value" of that symbol if any. For example, MY_DOCTOR could evaluate to:

```
(Livingstone (23 Stanley Street))
```

And, as it comes across the list to be evaluated, the interpreter will regard the first item of the list as a function and subsequent items as arguments passed to that function. For example:

(+ 2 2) will return 4

(CAR MY_DOCTOR) will return Livingstone because, as we will see later, CAR is a function which returns the first item of a list.

The LISP interpreter can therefore be described as the successive invocation of the functions below:
- READ which reads an S-expression (or simply an expression for short) from the keyboard or from a file.
- EVAL which evaluates an expression.
- PRINT which prints an expression to the screen or to a file.

Depending on the LISP dialect you are using, this interpreter may come as a standard function (TOPLEVEL, DRIVER, etc.), whose definition contains a compound of the three functions above: (PRINT (EVAL (READ)))
Note that, since arguments are evaluated before they are passed to functions, the READ function is invoked first; its result is then passed to EVAL; and the result from EVAL is finally passed to PRINT! (We'll see later that most functions evaluate their arguments--this is known as 'call by value'; some functions don't, e.g. control structures--this is known as 'call by name').

The EVAL function is thus the core of the LISP language: when you enter (+ 2 2), EVAL will invoke the code for the addition function.

I.3 Predefined functions

All LISP dialects share the same basic kernel of functions which is enough to write any program. Dialects are differentiated by more or less comprehensive sets of extended functions, which make it possible to write more concise programs. Below is a description of the basic kernel:

a) QUOTE

The QUOTE function protects its argument from being evaluated. Therefore it will not attempt to evaluate its argument and will return it unchanged.
Example:

```
(QUOTE (+ 2 2))    -->    (+ 2 2)
```

Since this function is used very often, an abbreviation (the single quote character) has been assigned to it:

```
'A     -->   A
'(CAR MY_DOCTOR)    -->   (CAR MY_DOCTOR)
```

For inquisitive minds, the ' character is known as a "character macro", that is a function which is evaluated as soon it is found among the data stream input from the keyboard (the action of the ' character macro consists of invoking the READ function to read the next expression, building a list with QUOTE as the first list item and that expression as the second item, and finally returning that list!)

b) Selectors: CAR and CDR

Both these functions evaluate their arguments. CAR extracts the left-hand part of the dotted pair passed as an argument and CDR extracts the right-hand part (unfortunately the names of these functions are derived from the A and D registers used on the first machine where LISP was implemented...)
Example:

```
(CAR '(+ 2 3))    -->  +
(CAR '((A B) C))  -->  (A B)
(CDR '(+ 2 3))    -->  (2 3)
(CDR '((A B) C))  -->  (C)
```

Depending on the LISP dialect, applying the CAR function to an atom may return either an error or the value of that atom. Similarly, applying CDR to an atom may return either an error or a list of properties associated with that atom. For both functions, the latter makes it possible to apply CAR and CDR to any LISP expression, but it is best to use these functions with dotted pairs only (i.e. lists or trees) if you want to be able to port your programs to a different LISP dialect.

c) Constructors

CONS is the basic constructor function which allows you to build a dotted pair from any two expressions.
Example:

```
(CONS 'A 'B)        -->  (A.B)
(CONS 'A '(B C))    -->  (A B C)
(CONS '(A B) '(C)) -->  ((A B) C)
(CONS 'A NIL)       -->  (A)
(CONS NIL '(A))     -->  (NIL A)
```

Alternately, the LIST function is always available to build lists of any size more easily. LIST accepts any numbers of arguments.
Example:

```
(LIST 'a 'b 'c 'd)    -->  (a b c d)
(LIST 'a '(b c) 'd)   -->  (a (b c) d)
```

d) Predicates

Predicates are functions which return a Boolean value (where NIL stands for "False" and any other result is "True"). They all evaluate their arguments.

```
ATOM    returns T if its argument is an atom, otherwise it returns NIL.
NUMBERP returns T if its argument is a number, otherwise it returns NIL.
CONSP   returns T if its argument is a list, otherwise it returns NIL.
NULL    returns T if its argument is NIL, otherwise it returns NIL.
EQUAL   returns T if its arguments are equal, otherwise it returns NIL.
EQ      returns T if its arguments are the same, otherwise it returns NIL.
```

The last two deserve some explanation: EQ checks if the two objects passed as arguments are one and the same in memory (physical equality) whereas EQUAL compares every branch of either object to check whether they have the same external representation.
Example:

```
(ATOM 'A)       -->  T
(ATOM '(A))     -->  NIL
(ATOM '())      -->  T   because () is the empty list represented by
                         the NIL atom!
(NULL 'A)       -->  NIL
(NULL '(A))     -->  NIL
(NULL NIL)      -->  T
(EQUAL '(A (B C) D)  '(A (B C) D))      -->  T
(EQ    '(A (B C) D)  '(A (B C) D))      -->  NIL
(EQUAL 'A 'A)   -->  T
(EQ    'A 'A)   -->  T because atoms are stored uniquely in memory
```

e) CONDitional

COND accepts as parameters any number of (non-evaluated) clauses in the form "(predicate expression)". COND does not evaluate its arguments as a whole, but it evaluates the predicates inside the clauses in sequence until it finds one which is "True" (not NIL), then it evaluates the associated expression in the clause and returns the value of that expression. Otherwise (if all predicates evaluated to NIL), COND returns NIL.
Example:

```
(COND ((EQ 'A 'A) 'YES))  -->  YES
(COND ((EQ 'A 'B) 'YES))  -->  NO
(COND   ((EQ 'A 'B) 'YES)
        (T          'NO))       -->  NO
                This demonstrates the use of T as the last predicate,
                but you can do without it as shown below:
(COND   ((EQ X 1) 'FIRST)
        ((EQ X 2) 'SECOND)
        ((EQ X 3) 'THIRD)
        (         'OTHER)     --> FIRST, SECOND, THIRD or OTHER
                                  depending on the value of X...
```

f) Arithmetic

Function names may vary from one LISP dialect to the next, but you always have the four basic operations as well as order predicates.
Example:

```
(- 239 (* 7 (/ 239 7)))    -->  1
(< 3 7)                    -->  T
```

g) Assignment

For LISP purists, assignment must be eschewed just like a GOTO statement in a structured language. In fact, assignment is often reserved to the higher level of a program, when you have to assign a name to a value, like in maths. But then this assignment must not be changed (symbols are not variables!) in order to avoid side effects.
Example:

```
(SETQ A '(1 2 3))       -->  (1 2 3)
A                       -->  (1 2 3)
```

SETQ does not evaluate its first argument (the symbol) and returns its second argument after assigning it to the symbol. SET is even more dangerous because it also evaluates its first argument:

```
(SETQ B 'A)     -->  A
B               -->  A
(SET B '(A B))  -->  (A B)
B               -->  A
A               -->  (A B)
```

h) Defining Functions

A LISP function is an expression which has the generic form below:

```
(LAMBDA (param1 param2 ... paramN) expr)
```

where param1, param2 ... paramN are the names of formal parameters and expr is the body of the function.

Example:

```
(LAMBDA (X) X)  is the identity function (it returns its argument)
(LAMBDA (X Y)
  (COND ((ATOM X) (CONS X Y))
        (T        (CONS (CAR X) Y))))
                is a function which adds X to the start of list Y if
                X is an atom, otherwise it adds the CAR of X to the
                start of list Y.
```

There is no standard way to associate a function definition to a function name: LeLisp uses the function DE (for example: "(DE IDENTITY(X) X)"), VLisp stores the definition in the property list of an atom, other dialects associate the definition to the value of an atom. In the rest of this manual, we'll use the MuLisp function PUTD.

Example:

```
(PUTD 'IDENTITY '(LAMBDA(X) X))
```

In MuLisp, the default value of atoms are the atoms themselves, which makes it possible to remove the quote (') from before a function name when no value is assigned to it. Now, a few classics:

```
(PUTD LENGTH
  '(LAMBDA (X)
     (COND ((NULL X) 0)
           (T        (+ 1 (LENGTH (CDR X))))
)  ) )
        calculates the length of a list: (LENGTH '(A B C)) --> 3
(PUTD MEMBER
  '(LAMBDA (X L)
     (COND ((NULL L)          NIL)
           ((EQUAL X (CAR L)) T)
           (T                 (MEMBER X (CDR L)))
)  ) )
        looks for an expression in a list:
                (MEMBER '(A B) '(A B (A (A B)) (B A) (A B))  --> T
        because (A B) is found in fourth position.
(PUTD FACT
  '(LAMBDA (N)
     (COND ((EQ N 0) 1)
           (T        (* N (FACT (- N 1))))
)  ) )
        calculates the factorial of a number...
```

II. Introducing OricLisp

OricLisp is a LISP dialect derived from LeLisp (by Jerome Chailloux) for user-defined functions (LAMBDA, FLAMBDA, MLAMBDA, see below) and from MuLisp (by Albert D.Rich, David R.Stoutemyer, and Roy Feldman) for the rest, although OricLisp is implemented quite differently (as MuLisp was implemented on Intel 8080 and 8086 processors).
Relatively few functions have been predefined (over 60 of them, however!) in order to leave as much as memory space as possible available to the user. Nevertheless the powerful features of OricLisp make it possible to develop sizeable applications.

OricLisp was written in 1986 (but this manual only in 1995!).

II.1 Main Features

* You can save a memory dump so you can resume a session at the exact stage where you left it, or extend the language with custom definitions, or produce Lisp applications which run automatically.
* Full 32-bit arithmetic, using any number base (radix) from 2 to 36.
* Variable-length symbols (up to 256 characters) which may contain any ASCII character (including the space character). Symbols can thus be used as character strings.
* Segmented virtual stack so the processor can access a 3K stack at the same speed as the usual 256-byte stack.
* Large user RAM with a standard 16K for dotted pairs and character strings (i.e. over 4000 dotted pairs), and almost 12K for atoms. This memory layout may be changed.
* The garbage collector compresses all user workspaces (strings, pairs, numbers, symbols) transparently.
* Eval and non-eval functions, plus function and character macros!
* Closed pointer space and address-typed data for greater efficiency.

II.2 Getting Started

Enter CLOAD"ORICLISP" from the Oric Basic 1.1 command line to load and run the software from tape (or LOAD"ORICLISP" to load it from disk, but memory dumps can only be saved to tape). The top line of the screen displays the copyright text, and a question mark prompt is displayed to indicate that the system is ready for character input from the keyboard. Enter a few expressions to verify that the interpreter is running:
Example:

```
? 'HELLO
=HELLO
? (+ 2 2)
=4
```

a) Display and Keyboard

For increased speed, display routines have been redesigned, so you can use 38 or 40 columns and the full 28 rows of the screen with a faster scroll routine.

The keyboard scan rate varies based on the user activity: the keyboard is scanned every 3 hundredths of a second during character input, but only every second during calculations and every tenth of a second during display. This combines performance and user comfort, because a program can always be stopped during calculations by holding down Ctrl-C for 1 second, whereas it runs about 15 percent faster.

b) Line Editor

The OricLisp line editor is quite different from the Basic one and uses a 128-character buffer (i.e. over 3 screen rows). The last entry (terminated by the RETURN key) is stored in the buffer, making it easier to edit the previous entry.

Two edit modes are available: Insert and Overwrite. The default mode is Overwrite. Press Ctrl-I to toggle to Insert mode, or back to Overwrite mode.

In Overwrite mode, any ASCII character you type replaces the character located at the same position in the buffer (hence in the previous entry).

In Insert mode, any ASCII character you type is inserted at the current position in the buffer, and the rest of the buffer is shifted to the right.

The highest ASCII character (DEL) has a special meaning because it deletes the last character you typed (thus moving one position to the left in the buffer). Apart from CTRL-I, two more control characters are available to use the previous entry from the buffer:

- Ctrl-A copies the current character from the buffer, so holding down Ctrl-A copies the previous entry up to the character you want.
- Ctrl-D deletes the next character in the buffer. (You will not see the effect of this keystroke until you press Ctrl-A, however).

For example: suppose you have entered the following line:

```
? (PUTD FACT '(LAMBDA(N) COND ((EQ N 0) 1)) (T (* N (FACT (- N 1))))))
=(LAMBDA(N) COND ((EQ N 0) 1))
```

Then you realise that you forgot an open parenthesis before COND and typed a extra close parenthesis after the first clause.
To edit your entry, you just have to hold down Ctrl-A until

```
? (PUTD FACT '(LAMBDA(N)
```

is displayed, then press Ctrl-I to toggle to Insert mode, type an open parenthesis, then hold down Ctrl-A again until

```
? (PUTD FACT '(LAMBDA(N) (COND ((EQ N 0) 1)
```

is displayed, then delete the extra close parenthesis using Ctrl-D, and copy the rest of the line using Ctrl-A...

II.3 OricLisp Primitives

a) QUOTE

This is the classic QUOTE function, which protects its argument from being evaluated.

```
? (QUOTE (+ 2 2))
=(+ 2 2)
```

Note: the character macro ' has the same effect.

```
? '(+ 2 2)
=(+ 2 2)
```

b) Selectors

* Both CAR and CDR selectors are available, as well as compounds of 2 or 3 CARs or CDRs (for example, "(CADR X)" is equivalent to "(CAR (CDR X))"). OricLisp uses a closed pointer space, so you can always apply CAR or CDR without any error; for an atom, CAR returns the value associated with that atom and CDR returns the list of properties for it. The value of a symbol defaults to the symbol itself (until a different value is assigned to it); the value of a number is always the number itself. The property list of a symbol is empty by default (i.e. it is NIL); the property list of a number reflects the sign of the number: T if the number is greater than or equal to zero, or NIL if the number is less than zero.

Example:

```
? (CAR '(A.B))
=A
? (CADR '(A B C))
=B
? (CDDR '(A B C D))
=(C D)
```

* LAST returns the last first-level dotted pair of its argument (rather than the last item), or NIL if the argument is an atom. Returning the last dotted pair rather than the last item makes it possible to modify this dotted pair (for example to add to the end of list using RPLACD, but be careful with the effects of such physical modifications). And you can easily get the last item by using an additional CAR.

Example:

```
? (LAST '(A B C D))
=(D)
? (LAST '(A B.C))
=(B.C)
? (CAR (LAST '(A B C D)))
=D
```

* ASSOC looks for the first argument (called the key) in the A-list supplied by the second argument. An A-list (for associative list) is a list in the form

```
((key1.val1) (key2.val2) ... (keyN.valN))
```

ASSOC compares (using EQUAL) the first argument with each of the keys (skipping any atom items in the A-list) and returns the whole item corresponding to the first match, or NIL if no match is found. Some Lisp dialects only return the value part of the dotted pair, then you just need to add a CDR to get the same result (whereas returning the whole pair makes it possible to modify the value associated with the key).

Example:

```
        ? (ASSOC 'MARTIN '((SMITH JOHN 61586273) (MARTIN JAMES 61483922)
                           (SMITH ALLAN 61289019)))
        =(MARTIN JAMES 61483922)
```

* MEMBER looks (using EQUAL) for the first argument in the list supplied by the second argument et returns the end of the list starting from the found item, or NIL if no match is found. Whereas some Lisp dialects only return a Boolean value (i.e. T or NIL), the actual result can be used here (for example, in order to delete a list item).

Example:

```
        ? (MEMBER '(MARTIN JAMES 61483922)
                  '((SMITH JOHN 61586273) (MARTIN JAMES 61483922)
                                          (SMITH ALLAN 61289019)))
        =((MARTIN JAMES 61483922) (SMITH ALLAN 61289019)))
```

c) Constructors

---

* CONS builds a dotted pair from the two supplied arguments.

Example:

```
        ? (CONS 'A 'B)
        =(A.B)
        ? (CONS 'A '(B C))
        =(A B C)
```

* LIST builds a list from all supplied arguments.

```
        ? (LIST 'a '(b c) 'd)
        =(a (b c) d)
```

* OBLIST builds a list of all symbols.

```
        ? (OBLIST)
        =(LAST OBLIST DIV MOD ..... LAMBDA T NIL)
```

* APPEND builds a list by concatenating the two lists passed as arguments. Unlike NCONC, APPEND creates new dotted pairs for the beginning of that list. Note: you can use APPEND with a single argument to duplicate a list.

```
        ? (APPEND '(A (B C) D) '(E F (G H)))
        =(A (B C) D E F (G H))
```

* REVERSE builds a new list by reversing its first argument. (In this implementation a second parameter is used to hold the intermediate result. Therefore, if a second argument is supplied, it will be appended to the reversed list.)

```
        ? (REVERSE '(A (B C) D))
        =(D (B C) A)
        ? (REVERSE '(A (B C) D) '(E F))
        =(D (B C) A E F)
```

* GC is not a constructor like CONS, LIST, OBLIST, APPEND or REVERSE, but it is also used to manage the memory space. All constructors call CONS to allocate a dotted pair. When no space is left to allocate a new pair, the Garbage Collector is automatically invoked. You can also invoke it explicitly using the GC function.

d) Predicates
---

* EQUAL returns T if its two arguments are equal. All branches are compared.

```
        ? (EQUAL '(A (B C) D) '(A (B C) D))
        =T
```

* EQ returns T if its two arguments are one and the same object in memory. No two symbols can have the same name. There may be several numbers with the same value but EQ regards them as being identical. When you share dotted pairs, EQ is a fast way of comparing them because it does not look at branches.

```
        ? (EQ '(A (B C) D) '(A (B C) D))
        =NIL
```

* ATOM returns T if its argument is an atom (i.e. not a dotted pair)

```
        ? (ATOM 'A)
        =T
        ? (ATOM '(A B))
        =NIL
        ? (ATOM '())
        =T
```

* NULL returns T if its argument is NIL (i.e. the empty list).

```
        ? (NULL 'A)
        =NIL
        ? (NULL '(A B))
        =NIL
        ? (NULL '())
        =T
```

* PLUSP returns T if its argument is a number greater than or equal to zero.

```
        ? (PLUSP 0)
        =T
        ? (PLUSP 'A)
        =NIL
```

* MINUSP returns T if its argument is a number less than zero.

```
        ? (MINUSP -3)
        =T
        ? (MINUSP '(A B))
        =NIL
```

* ZEROP returns T if its argument is zero.

```
        ? (ZEROP NIL)
        =NIL
        ? (ZEROP 0)
        =T
```

Note: Any expression can be used as a predicate since any non-NIL value is regarded as "True". For example, T is the "always true" predicate, which is useful for the last clause of a conditional.

e) Conditional and Control Structures
---

* COND is the classic conditional in the form:

```
        (COND (pred1 exp11 exp12 ... exp1N)
              (pred2 exp21 exp22 ... exp2M)
                ....
        )
```

Each predicate pred1, pred2 ... is evaluated in sequence until a non-NIL value is found among them, whereas the associated expressions expI1, expI2... are also evaluated and the last one provides the return value of COND (if no expression is associated with a predicate, the value of the predicate itself is returned). Having more than one expression after a predicate is useful only if the expressions have side effects).

Example:

```
        ? (COND ((< A B) A) (B)) returns the minimum of two numbers A and B.
```

* PROGN evaluates all its arguments in sequence and returns the value of the last. PROGN is useful only for expressions which have side effects and it can generally be omitted because PROGN is implied in function definitions and in the conditional (COND).

Example:

```
        ? (PROGN (PRIN '(FACT 5)) (FACT 5))
        (FACT 5)=120
```

* PROG1 evaluates all its arguments in sequence and returns the value of the first. Like PROGN, its use is related to side effects.

Example:

```
        ? (SETQ A (PROG1 B (SETQ B A))) swaps the values assigned to A and B.
```

* AND is both the classic logical "and" and a control structure (equivalent to embedded "if...then..." statements). Arguments are evaluated in sequence until a NIL value is found, in which case the result is NIL and the remaining arguments are "skipped". If all arguments evaluate to a non-NIL value, the value of the last argument is returned, thus the result is "True".

Example:

```
        ? (AND (ZEROP 1) (PRINT 'ARGH))
        =NIL    Note that the second argument is not evaluated in this case
        ? (AND A B C) is equivalent to (COND (A (COND (B (COND (C))))))
```

* OR is both the classic logical "or" and a control structure (equivalent to embedded "if...then...else if" statements). Arguments are evaluated in sequence until a NIL value is found, in which case this value is returned and the remaining arguments are "skipped". If all arguments evaluate to NIL, NIL is returned.

Example:

```
        ? (OR NIL '() 'A (PRINT 'ARGH))
        =A
        ? (OR A B C) is equivalent to (COND (A) (B) (C))
```

* NOT is the logical "not" (it is not a control structure but it can be used to negate predicates). NOT returns T if its argument is NIL, otherwise it returns NIL. NOT is therefore identical to the NULL predicate.

Example:

```
        ? (NOT NIL)
        =T
```

* WHILE is an imperative control structure in the form below:

```
        (WHILE pred exp1 exp2 ... expN)
```

While pred evaluates to "True" (not NIL), WHILE evaluates the expressions exp1, exp2 ... expN iteratively (so there must be some side effect before the value of pred becomes NIL). WHILE is implemented non-recursively, therefore long (even endless) loops can be performed without any stack overflow.
Example:

```
        ? (WHILE T (PRINT (EVAL (READ)))
                an endless loop for a new interpreter!
        ? (WHILE (NOT (ZEROP N)) (SETQ N (- N 1)))
        =NIL
                decrements N down to 0
```

f) Arithmetic
---

* RADIX changes the number base for the notation of numbers entered from the keyboard and displayed on screen. The new base is passed as an argument and must range from 2 to 36 (beyond 10, up to 26 letters can be used as digits). RADIX returns the new base as the result (but you will note that the current base is always printed as "10"). If a nonnumeric argument is supplied, the current base remains unchanged and is returned.
Example:

```
        ? (RADIX 8)
        =10
        ? (+ 6 7)
        =15
        ? (RADIX 12)
        =10
        ? (+ 6 7)
        =13
```

* The four integer operations are +,-,*,/. They take two numbers as arguments and return the result. A NONNUMERIC error is generated if an operand is not a number, and a DIVBYZERO is generated if you attempt to divide by 0. / returns the quotient of the integer division whereas MOD returns the remainder. As both the quotient and the remainder are often needed, DIV carries out the division once and returns a dotted pair (quotient.remainder).
Example:

```
        ? (DIV 10 3)
        =(3.1)
```

* Order predicates are < and >. They return T if the order of the two arguments is true, and NIL otherwise. Comparisons in a broader sense (including equality) are not primitives, so the test has to be reversed.
Example:

```
        ? (NOT (< A B))
                returns T if A>=B, and NIL otherwise.
```

g) Modifier Functions and Side-Effect Functions

* SETQ assigns the value supplied by the second argument to the symbol specified by the first argument (which is not evaluated). If the symbol was local to a function, the assignment is lost when exiting the function. For example:

```
? (SETQ A '(1 2))
=(1 2)
? A
=(1 2)
? (SETQ A 'B)
=B
? A
=B
```

* SET assigns the value supplied by the second argument to the symbol specified by the first argument (both arguments are evaluated). Example:

```
? (SETQ A 'B)
=B
? (SET A 3)
=3
? B
=3
? A
=B
```

* RPLACA and RPLACD replace respectively the CAR and the CDR of the dotted pair supplied as the first argument by the second argument, and returns the updated dotted pair. Example:

```
? (SETQ A '(1 2))
=(1 2)
? (RPLACA A 0)
=(0 2)
? A
(0 2)
? (RPLACD A '(3 4))
=(0 3 4)
? A
(0 3 4)
```

* NCONC concatenates two lists like APPEND, except that no dotted pair is used up. NCONC modifies the end of the list supplied by its first argument to append the second list to it. Example:

```
? (SETQ A '(1 2))
=(1 2)
? (NCONC A '(3 4 5))
=(1 2 3 4 5)
? A
=(1 2 3 4 5)
```

* MEMORY is used to read or write a byte from or to memory. If a single argument is supplied, it is considered as a memory location and MEMORY returns its contents. If two arguments are supplied, MEMORY sets the memory location to the byte value of the second argument and returns the value of the previous contents.
Example:

```
? (RADIX 16)
=10
? (MEMORY 26B 1)
=7      sets the paper colour to white...
```

* TIME returns the value of a timer in 1/100th seconds which can be used to measure the execution time of a program.
Example:

```
? (SETQ T1 (TIME)) (GC) (- (TIME) T1)
=5
```

* PRINT displays the expression passed as a parameter and returns this expression as the result. PRIN does the same but it does not print a carriage return at the end of the expression.
Example:

```
? (PRINT 'HELLO)
HELLO
=HELLO
? (PRINT '(A (B C) D))
(A (B C) D)
=(A (B C) D)
? (PRIN 0)
0=0
```

* READ reads an expression from the keyboard and returns it as the result. If more than one expression is supplied, the remaining expressions are kept in the keyboard buffer for possible use by a further READ.
Example:

```
? (PRIN '(+ 2 2)) (+ 2 2)
(+ 2 2)=(+ 2 2)
=4
? (SETQ A (READ)) (+ 2 2)
=(+ 2 2)
        Note that (+ 2 2) was not read by the interpreter
        but actually by READ.
```

h) Defining Functions

* An OricLisp function which evaluates its arguments is an expression in the form

```
(LAMBDA (param1 ... paramN) exp1 exp2 ... expM)
```

where param1 ... paramN are the names of the formal parameters and exp1, exp2 ... expM is the body of the function (a LAMBDA evaluation therefore contains an implicit PROGN).
Examples:

```
(LAMBDA () (/ (TIME) 100))
        a function without parameters which returns the time in seconds
(LAMBDA (N) (+ N 1))
        a function which returns the value of its argument plus one
```

Caution! The second item of the function list (formal parameters) must be a list (which may be empty, like in the first example above).

LAMBDAs are "unnamed" functions which can be used in the same way
as OricLisp primitives.
Example:

```
        ? ( (LAMBDA(N)(+ N 1)) 3)
        =4
```

To assign a name to a LAMBDA, use the PUTD primitive.

```
        ? (PUTD 'INCR '(LAMBDA(N)(+ N 1)))
        =(LAMBDA (N) (+ N 1))
        ? (INCR 3)
        =4
```

Note: A symbol can have at the same time a value, a property list and
a function definition associated together.
For example:

```
        ? (SETQ INCR '(1 2))
        =(1 2)
        ? (INCR 3)
        =4
        ? INCR
        =(1 2)
```

* GETD is used to get the function definition of a symbol. GETD returns the
LAMBDA of a function defined in LISP, T for a function in machine code, or
NIL if the function is undefined.
Example:

```
        ? (GETD 'T)
        =NIL
        ? (GETD 'EVAL)
        =T
        ? (GETD 'INCR)
        =(LAMBDA (N) (+ N 1))
```

* MOVD copies the function definition from a symbol to another symbol,
thus defining a synonym for a function without consuming any extra memory
space (apart from the target symbol itself).
Example:

```
        ? (MOVD 'APPEND 'COPY)
        =T
        ? (COPY '(A B (C D) E))
        =(A B (C D) E)
```

II.4 Advanced Concepts

a) Non-Eval Functions

OricLisp allows you to define functions which do not evaluate their arguments (called 'non-eval functions'). Such a function has the following format:

```
(FLAMBDA param exp1 exp2 ... expM)
```

A single symbol is supplied as a formal parameter; it will be linked to the list of all non-evaluated parameters.
Non-eval functions are used to defined new control structures (but also sometimes to define functions with an undetermined number of parameters).
Example:

```
(FLAMBDA L
  (COND ((EVAL (CAR L)) (EVAL (CADR L)))
        (T              (EVAL (CADDR L)))
) )                              defines an IF X THEN Y ELSE Z !!
```

which will be linked as follows:

```
? (PUTD 'IF '(FLAMBDA L
?              (COND ((EVAL (CAR L)) (EVAL (CADR L)))
?                    (T              (EVAL (CADDR L))) )))
=(FLAMBDA L (COND ((EVAL (CAR L))(EVAL (CADR L))) (T (EVAL (CADDR L)))))
? (IF NIL (PRINT 'ARGH) (PRINT 'OK))
OK
=OK     and you note that, indeed, the THEN clause is not evaluated,
        unlike what would have happened if the IF structure had been
        defined by (LAMBDA (X Y Z) (COND (X Y) (Z)))
```

Example:

```
(PUTD 'PLUS '(FLAMBDA L
  (COND ((NULL L) 0)
        ((+ (EVAL (CAR L)) (PLUS (CDR L))))
)))
        defines a PLUS operation which sums all its arguments
? (PLUS 1 2 3 4 5 6 7)
=28
```

b) Character macros

Character macros are special characters with an associated function which is called when the respective character is input from the keyboard. The quote is the only predefined character macro. READ looks up a table (stored at $8000) to determine whether a character is a character macro. To define a character macro, you just have to store the corresponding byte in this table and associate it with the function you want.

Example:

```
? (PUTD '"#" '(LAMBDA () (EVAL (READ))))
=(LAMBDA NIL (EVAL (READ)))
? (RADIX 16) (MEMORY (+ 8000 3) FF)
=10
=0      # has the ASCII value 23h; the table starts at character 20h...
? (SETQ N 8) (PUTD 'TEST '(LAMBDA (X) (/ X #(* N N))))
=8
=(LAMBDA (X) (/ X 64))
        Note that the expression preceded with # has been evaluated...
? (QUOTE # (+ 2 2))
=4      ... although inside a protected expression.
```

c) Function Macros

Function macros are dual-evaluation functions defined as follows:

```
(MLAMBDA param exp1 exp2 ... expN)
```

A single symbol is supplied as a formal parameter; at runtime it will be linked to the whole list which invokes the macro (without any evaluation), including the function at the beginning of the list. The body of the macro (the expressions exp1 ... expN) is executed and the result is fed back to evaluation!
The purpose of a function macro is therefore to replace the macro call with another expression.

Example:

```
(PUTD 'IF '(MLAMBDA L
  (LIST 'COND
        (LIST (CADR L) (CADDR L))
        (LIST (CAR (CDDDR L)))
)))
```

defines an "IF" macro which builds a COND expression containing the arguments from the IF.

Thus,

```
(IF (PLUSP N) N (- 0 N))
```

will build the list

```
(COND ((PLUSP N) N) ((- 0 N)))
```

which will be evaluated in turn, yielding the absolute value of N.

This powerful mechanism may seem inefficient and cumbersome for defining an IF function (i.e. invoking the macro, reading through the arguments and allocating dotted paris to build a COND expression, before finally evaluating this expression. But its value shows when you compare it to the IF version obtained using FLAMBDA, in the context of "overriding" function macros. To verify this, let's slightly modify the IF definition as follows:

```
        (PUTD 'IF '(MLAMBDA L
          (RPLACA L 'COND)
          (RPLACD L (LIST (LIST (CADR L) (CADDR L))
                          (LIST (CAR (CDDDR L)))))
        ))
                Now the IF will be replaced once for all by a COND
                when it is first executed.
```

Example:

```
        ? (PUTD 'FACT '(LAMBDA (N)
        ?    (IF (ZEROP N) 1 (* N (FACT (- N 1))))))
        =(LAMBDA (N) (IF (ZEROP N) 1 (* N (FACT (- N 1)))))
        ? (FACT 3)
        =6
        ? (GETD 'FACT)
        =(LAMBDA (N) (COND ((ZEROP N) 1) ((* N (FACT (- N 1))))))
```

When the IF is first executed, the code for FACT gets modified! The code is then more efficient than when using an IF implemented with an FLAMBDA!

This kind of manipulation is useful when porting LISP programs from one machine to another (in our example, suppose that IF is not available on the target machine) or for defining functions efficiently. Although macros themselves are not very easy to read and more difficult to write than standard LAMBDAs, using macros often makes the overall program more readable while preserving high performance.

Example:

The LET construct is available in many LISP dialects for defining local symbols in the block. Using LET, the above IF definition could be written in a more legible way:

```
        (PUTD 'IF '(MLAMBDA L
          (LET (
                (X (CADR L))
                (Y (CADDR L))
                (Z (CAR (CDDDR L)))
               )
             (RPLACA L 'COND)
             (RPLACD L (LIST (LIST X Y) (LIST Z)))
        )))
```

LET can be defined efficiently with a macro which replaces it as a LAMBDA:

```
        (PUTD 'LET '(MLAMBDA L
          (RPLACA L (CONS 'LAMBDA
                          (CONS (ALLCAR (CADR L))
                                (CDDR L))))
          (RPLACD L (ALLCADR (CADR L)))
        ))
```

where ALLCAR and ALLCADR are used respectively to get the formal parameters and the actual arguments:

```
        (PUTD 'ALLCAR '(LAMBDA (L)
         (COND ((NULL L) NIL)
               ((CONS (CAAR L) (ALLCAR (CDR L)))))))
        (PUTD 'ALLCADR '(LAMBDA (L)
         (COND ((NULL L) NIL)
               ((CONS (CADAR L) (ALLCADR (CDR L)))))))
```

d) Higher-Level Functions

OricLisp allows you to pass a function as a parameter to another function. But, since the first item in a list (here the function) is never evaluated by the EVAL function, the function obtained as an argument cannot be invoked directly. For example, the classic functional MAP (which applies a function to all the items of a list and returns the list of results) cannot be written as follows in OricLisp:

```
(PUTD 'MAP '(LAMBDA (F L)
  (COND ((NULL L) NIL)
        ((CONS (F (CAR L))
               (MAP F (CDR L)))))))
```

because the symbol F in the position of a function will not be evaluated. (Note: in many LISP dialects, the above definition of MAP will only work if F has no associated definition.)
In addition, OricLisp has no APPLY function available but such constructs as (APPLY F ARGS) can easily be replaced with (EVAL (CONS F ARGS)).
MAP can thus be defined:

```
(PUTD 'MAP '(LAMBDA (F L)
  (COND ((NULL L) NIL)
        ((CONS (APPLY F (LIST (CAR L)))
               (MAP F (CDR L)))))))
```

and this definition will work without any restriction. APPLY can be defined using just (LAMBDA (F L) (EVAL (CONS F L))) or using a macro:

```
(MLAMBDA L
  (RPLACA L 'EVAL)
  (RPLACD L (LIST (LIST 'CONS (EVAL (CADR L)) (EVAL (CADDR L))))))).
```

Example:

```
? (MAP '(LAMBDA(X)(* X X)) '(1 2 3 4 5))
=(1 4 9 16 25)
? (MAP 'PLUSP '(1 8 -2 6 -3))
=(T T NIL T NIL)
```

A. Appendix A: Copyright

B. Appendix B: Formal Definitions

For a "LISPian", a LISP function is worth a thousand words, so how could the behaviour of a primitive be better understood than by examining its code written in LISP? Many functions implemented in machine code in OricLisp could also be written in LISP, but then they would be less efficient and use up much stack space (because of their recursivity).

The real minimum kernel consists of the definitions LAMBDA, FLAMBDA, MLAMBDA and the primitives READ, EVAL, PRIN, COND, CAR, CDR, CONS, PUTD, GETD, NULL, ATOM, RPLACA, RPLACD, EQ, TIME, MEMORY, RADIX, <, >, +, -, *, /, MOD. All other functions can be written based on those core primitives.
The following definitions behave exactly the same as the corresponding machine-code routines (they return the same results under any circumstances), even though the actual implementation is different for the sake of performance. The only primitives which are implemented recursively are LIST and APPEND (they shouldn't...). Finally, APPLY is not a symbol defined in OricLisp, but it exists internally and can be defined by

```
(PUTD 'APPLY '(LAMBDA (F ARGS) (EVAL (CONS F ARGS))))
```

even though the actual implementation does not use up any dotted pair.

```
(PUTD 'AND
  '(FLAMBDA L
        (COND ((ATOM L) T)
              ((ATOM (CDR L)) (EVAL (CAR L)))
              ((EVAL (CAR L)) (APPLY 'AND (CDR L))))))

(PUTD 'APPEND
  '(LAMBDA (X Y)
      (COND ((ATOM X) Y)
            ((CONS (CAR X) (APPEND (CDR X) Y))))))

(PUTD 'ASSOC '(LAMBDA (X L)
                (COND   ((ATOM L) NIL)
                        ((ATOM (CAR L)) (ASSOC X (CDR L))
                        ((EQUAL X (CAAR L)) (CAR L))
                        ((ASSOC X (CDR L))))))

(PUTD 'CAAAR '(LAMBDA (X) (CAAR (CAR X))))
(PUTD 'CAADR '(LAMBDA (X) (CAAR (CDR X))))
(PUTD 'CADAR '(LAMBDA (X) (CADR (CAR X))))
(PUTD 'CADDR '(LAMBDA (X) (CADR (CDR X))))
(PUTD 'CDAAR '(LAMBDA (X) (CDAR (CAR X))))
(PUTD 'CDADR '(LAMBDA (X) (CDAR (CDR X))))
(PUTD 'CDDAR '(LAMBDA (X) (CDDR (CAR X))))
(PUTD 'CDDDR '(LAMBDA (X) (CDDR (CDR X))))
(PUTD 'CAAR '(LAMBDA (X) (CAR (CAR X))))
(PUTD 'CADR '(LAMBDA (X) (CAR (CDR X))))
(PUTD 'CDAR '(LAMBDA (X) (CDR (CAR X))))
(PUTD 'CDDR '(LAMBDA (X) (CDR (CDR X))))

(PUTD 'DIV '(LAMBDA (X Y) (CONS (/ X Y) (MOD X Y))))

(PUTD 'EQUAL
  '(LAMBDA (X Y)
       (COND ((ATOM X) (EQ X Y))
             ((ATOM Y) NIL)
             ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y))))))
```

```
(PUTD 'LAST '(LAMBDA (L)
                (COND   ((ATOM L) NIL)
                        ((ATOM (CDR L)) L)
                        ((LAST (CDR L))))))

(PUTD 'LENGTH '(LAMBDA (L)
                 (COND  ((ATOM L) 0)
                        ((+ 1 (LENGTH (CDR L)))))))

(PUTD 'LIST
  '(FLAMBDA L
      (COND ((ATOM L) NIL)
            ((CONS (EVAL (CAR L)) (APPLY 'LIST (CDR L)))))))

(PUTD 'MEMBER
  '(LAMBDA (X L)
        (COND ((ATOM L) NIL)
              ((EQUAL X (CAR L)) L)
              ((MEMBER X (CDR L))))))

(PUTD 'NCONC '(LAMBDA (X Y)
                (COND ((ATOM X) Y)
                      (T (RPLACD (LAST X) Y) X))))

(PUTD 'OR '(FLAMBDA L
                (COND   ((ATOM L) NIL)
                        ((EVAL (CAR L)))
                        ((APPLY 'OR (CDR L))))))

(PUTD 'PROG1
  '(FLAMBDA L
        (COND ((ATOM L) NIL)
              (T ((LAMBDA (X) (APPLY 'PROGN (CDR L)) X) (EVAL (CAR L)))))))

(PUTD 'PROGN
  '(FLAMBDA L
        (COND ((ATOM L) NIL)
              ((ATOM (CDR L)) (EVAL (CAR L)))
              (T (EVAL (CAR L)) (APPLY 'PROGN (CDR L))))))

(PUTD 'QUOTE '(FLAMBDA L (CAR L)))

(PUTD 'REVERSE
  '(LAMBDA (X Y)
        (COND ((ATOM X) NIL)
              ((ATOM (CDR X)) (CONS (CAR X) Y))
              ((REVERSE (CDR X) (CONS (CAR X) Y))))))

(PUTD 'SET '(LAMBDA (X Y) (RPLACA X Y) Y)))

(PUTD 'SETQ '(FLAMBDA L (COND ((NAME (CAR L)) (SET (CAR L) (EVAL (CADR L)))))))

(PUTD 'WHILE
  '(FLAMBDA L
      (COND ((EVAL (CAR L))
                (APPLY 'PROGN (CDR L))
                (APPLY 'WHILE L)))))
```

B. Appendix B: Porting MuLisp Programs

Apart from user function evaluation (LAMBDA, FLAMBDA, MLAMBDA), OricLisp primitives are quite close to MuLisp primitives, therefore MuLisp programs can be easily ported to OricLisp (and vice versa). To save space for user programs, not all functions available in MuLisp are defined in OricLisp. But the following definitions can be used to provide exact substitutes:

```
(PUTD 'NTH '(LAMBDA (N L)
        (COND   ((ZEROP N) (CAR L))
                ((ATOM L) NIL)
                ((NTH (- N 1) (CDR L)))))) for a MacLisp-like version

(PUTD 'NTH '(LAMBDA (L N)
        (COND   ((EQ N 1) L)
                ((ATOM L) NIL)
                ((NTH (CDR L) (- N 1)))))) for an InterLisp-like version

(PUTD 'TCONC '(LAMBDA (PTR OBJ)
        (SETQ OBJ (LIST OBJ))
        (COND   ((ATOM PTR) (CONS OBJ OBJ))
                ((ATOM (CDR PTR)) (RPLACA PTR OBJ) (RPLACD PTR OBJ))
                (T (RPLACD (CDR PTR) OBJ) (RPLACD PTR OBJ)))))

(PUTD 'LCONC '(LAMBDA (PTR L)
        (COND   ((ATOM L) PTR)
                ((ATOM PTR) (CONS L (LAST L)))
                ((ATOM (CDR PTR)) (RPLACA PTR L) (RPLACD PTR (LAST L)))
                (T (RPLACD (CDR PTR) L) (RPLACD PTR (LAST L))))))

(PUTD 'EVENP '(LAMBDA (N) (COND ((NUMBERP N) (ZEROP (MOD N 2))))))

(PUTD 'POP '(FLAMBDA P
        (COND   ((NOT (NAME (CAR P))) NIL)
                ((ATOM (CAAR P)) NIL)
                ((PROG1 (CAAAR P) (SET (CAR P) (CDAAR P)))))))

(PUTD 'PUSH '(FLAMBDA P
        (COND   ((NULL (CADR P)) NIL)
                ((NAME (CADR P)) (SET (CADR P) (CONS (EVAL (CAR P))
                                                     (CADR P)))))))

(PUTD 'PUT '(LAMBDA (NAM KEY OBJ)
  ((LAMBDA(ELEM)
        (COND ((NULL ELEM) (RPLACD NAM (CONS (CONS KEY OBJ) (CDR NAM))) OBJ)
              (T (RPLACD ELEM OBJ) OBJ)))  (ASSOC KEY (CDR NAM)))))

(PUTD 'GET '(LAMBDA (NAM KEY) (CDR (ASSOC KEY (CDR NAM)))))
```

```
(PUTD 'REMPROP '(LAMBDA (NAM KEY)
        (COND ((ATOM (CDR NAM)) NIL)
              ((EQUAL (CAADR NAM) KEY)
                        (SETQ KEY (CDADR NAM))
                        (RPLACD NAM (CDDR NAM))
                       KEY)
              ((REMPROP (CDR NAM) KEY)))))

(PUTD 'FLAGP '(LAMBDA (NAM ATT) (MEMBER ATT (CDR NAM))))

(PUTD 'FLAG '(LAMBDA (NAM ATT)
        (COND ((FLAGP NAM ATT) ATT)
              (T (RPLACD NAM (CONS ATT (CDR NAM))) ATT))))

(PUTD 'REMFLAG '(LAMBDA (NAM ATT)
        (COND ((ATOM (CDR NAM)) NIL)
              ((EQUAL ATT (CADR NAM)) (RPLACD NAM (CDDR NAM)) T)
              ((REMFLAG (CDR NAM) ATT)))))

(PUTD 'GCD '(LAMBDA (X Y)
        (COND   ((NOT (ZEROP Y)) (GCD Y (MOD X Y)))
                ((PLUSP X) X)
                ((- 0 X)))))

(PUTD 'COMMENT '(FLAMBDA L NIL))
```

The following differences may be found when porting programs:

* The body of a MuLisp function and an OricLisp function are evaluated differently. MuLisp implements an implicit COND/PROGN instead of an implicit PROGN inside function bodies (and COND clauses), which results in more compact code but also in some ambiguity for LAMBDAs which are then regarded as predicates. COND is therefore mandatory in the body of OricLisp functions.

* PLUSP returns NIL for 0 in MuLisp and T in OricLisp. Also, the CDR of a number (its property list) has the opposite meaning: its value is T for a nonnegative number in OricLisp, whereas it is T for a negative number in MuLisp. Whilst GREATERP and LESSP accept more than two arguments in MuLisp, such expressions must be written using several > and < in OricLisp. Identically, PLUS, DIFFERENCE, and TIMES accept more than two arguments in MuLisp, but those expressions can be easily rewritten using multiple +, - or *.

* MuLisp string functions (SUBSTRING, FINDSTRING, PACK, UNPACK, LENGTH, ASCII) cannot be implemented in OricLisp.

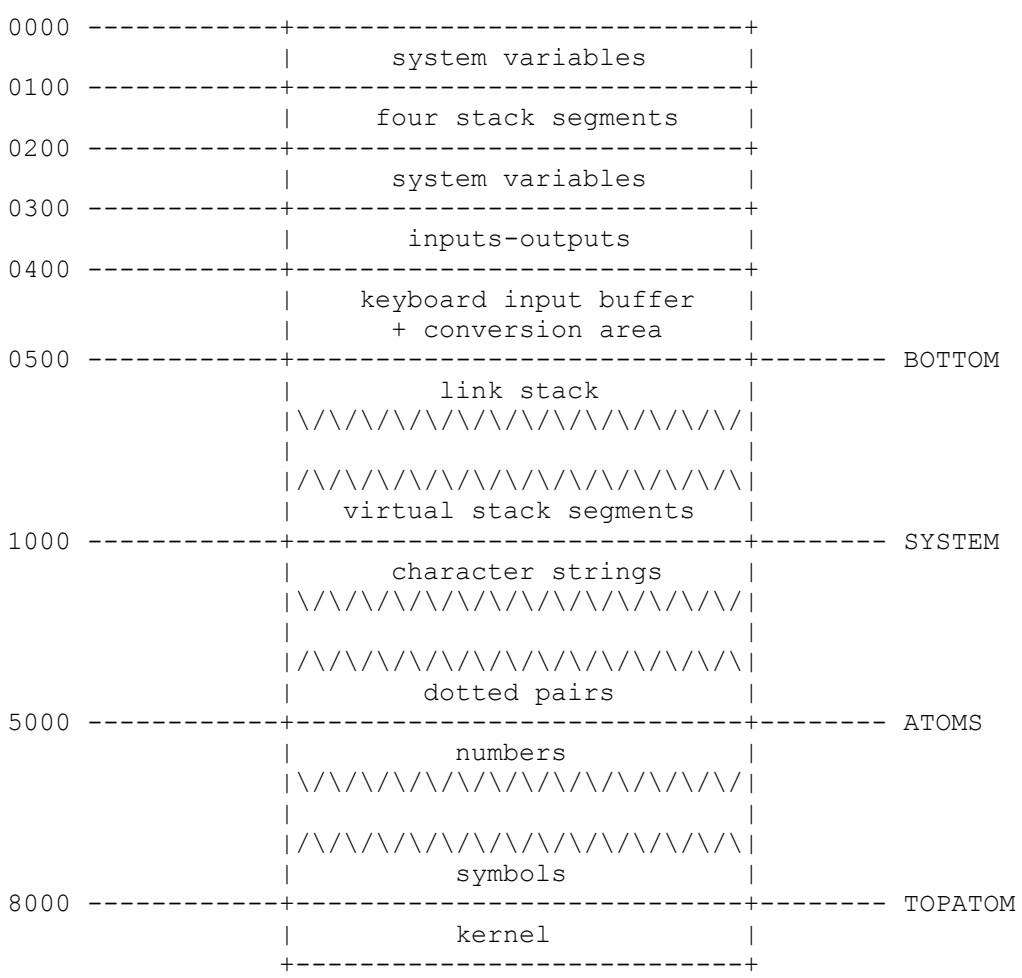
* MuLisp escapes (CATCH, THROW) cannot be implemented in OricLisp, neither can error handling.

* File read/write functions are not available in OricLisp.

* The powerful OricLisp function macros do not exist in MuLisp. Therefore, if an OricLisp program makes intensive use of them, it is better to port it to a different dialect, such as LeLisp.

C. Appendix D: Implementation Details

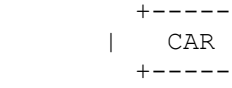
a) Memory Map



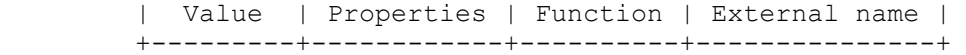
The addresses BOTTOM, SYSTEM and ATOMS may be changed to be adapted to different consumption in numbers and symbols, dotted pairs or recursivity by an application. Unfortunately, any change must be made by modifying kernel addresses BEFORE the kernel is executed...
The BOTTOM page may be changed by modifying location 9AA6, the SYSTEM page by modifying location 9AAA and the ATOMS page by modifying location 9AB0.

b) Data Structures

Dotted pairs have the following structure:



Symbols have the following structure:



Thus the CAR of symbol yields its value, and a symbol which has not be
given any value defaults to itself (i.e. the value points to the same symbol).
The CDR of a symbol holds the list de properties associated with the symbol;
this list defaults to NIL and may be changed by RPLACD.
The function field of the symbol is NIL if no function is associated;
it points to the list LAMBDA (or FLAMBDA, or MLAMBDA) for a user function;
and it contains the start address for a machine-code function (the last
two are differentiated by the most significant bit, which is set to 1 for
a machine-code routine stored above 8000h and cleared to 0 for a user function).
The external name field (or Print-name) points to the character string
which represents the symbol.

Numbers have the following structure:

Value	Properties	32-bit representation

The CAR of a number (its value) always points to itself.
The CDR of a number (its property) is T if the number is nonnegative,
or NIL if it is negative.

c) Virtual Stack

LISP is by essence a recursive language and this poses a problem when
implementing it for the 6502 processor which only has an 8-bit stack pointer.
OricLisp implements an unprecedented virtual stack which avoids having to
simulate a 16-bit stack and makes it possible to use standard 6502 stack
instructions 6502: JSR, RTS, PHA, PLA...
The virtual stack consists of 64-byte segments, therefore 4 segments are
present in the physical stack of the 6502. From time to time (when the EVAL
routine is invoked, for example), the stack pointer position is checked
to find if a segment boundary has been crossed. If so, this may result in
saving a segment of the physical stack to the virtual stack or retrieving
a segment from the virtual stack. However, the algorithm used is optimised
for handling recursive binary trees, so segments seldom need to be transferred
(no transfer occurs when the stack pointer oscillates between either side of a
segment boundary).

d) Garbage Collector

The first pass marks those pairs and atoms which can be accessed from the
value, property and function fields of symbols -- marking does not apply
to any symbol whose value points to itself and has no associated function
definition -- or from the link stack.
The second pass removes and compresses all unmarked symbols, numbers and pairs.
The third pass unmarks objects and adjusts pointers to objects relocated due
to compression.
The last pass compresses the strings for all symbols which have not been
removed by the garbage collector.

E. Appendix E: Sample Program: Knight's Ride

The Knight's Ride is a classical poser which consists of going through each square of a chessboard once and only once, while following the rules for a Knight's movement. LISP is well suited for this kind of problem-solving. The program below provides a quick solution for chessboards of any size:

```
; a list of the 8 possible moves for the Knight, sorted using a heuristic
method:
(SETQ DIR '((-2 1)(-2 -1)(-1 -2)(1 -2)(2 -1)(2 1)(1 2)(-1 2)))

; first a short function builds the list of integers M,M-1...1
(PUTD COUNTD '(LAMBDA (M) (COND ((ZEROP M) NIL) ((CONS M (COUNTD (- M 1)))))))

; then another function checks whether a square is outside the board:
(PUTD TEST '(LAMBDA (X Y) (AND (< -1 X) (< X N) (< -1 Y) (< Y N))))

; so as to build the list of locations which can be reached from M
(PUTD MOVES '(LAMBDA (STEPS)
  (COND ((NULL STEPS) NIL)
        ((TEST (+ (CAAR STEPS) (MOD (- M 1) N)) (+ (CADAR STEPS) (/ (- M 1) N)))
           (CONS (+ (+ M (CAAR STEPS)) (* N (CADAR STEPS))) (MOVES (CDR
STEPS))))
        ((MOVES (CDR STEPS))))))

; build a table of such lists for all squares
(PUTD TABLE '(LAMBDA (M)
  (COND ((ZEROP M) NIL)
        ((CONS (CONS M (MOVES DIR)) (TABLE (- M 1))))))

; now the program proper: look for a way through each and every square
(PUTD WAY '(LAMBDA (SQUARES REMAINING RIDE)
  (COND ((ZEROP REMAINING) RIDE)                ; found!
        ((NULL SQUARES) NIL)                    ; dead end...
        ((MEMBER (CAR SQUARES) RIDE)            ; already been here...
                (WAY (CDR SQUARES) REMAINING RIDE))
        ((WAY (CDR (ASSOC (CAR SQUARES) TAB))
                 (- REST 1)
                 (CONS (CAR SQUARES) PARC)))    ; let's try this way...
        ((WAY (CDR SQUARES) REST PARC)))))      ; and other squares too...

; the main program to launch the whole thing, and that's it!
(PUTD RIDE '(LAMBDA (N TAB)
  (SETQ TAB (TABLE (* N N)))                    ; calculate table only once
  (WAY (COUNTD (* N N)) (* N N))))              ; then start searching
```

Note 1: To find all solutions, you just have to replace the first clause of the WAY function with:

```
(COND ((ZEROP REMAINING) (PRINT RIDE) NIL)
```

Note 2: A memory dump supplied with OricLisp contains an enhanced version of this program, which can be used to reach a solution more quickly.