CHAPTER 2

DATA TYPES

Interlisp is a system for the manipulation of various kinds of data; it provides a large set of built-in data types, which may be used to represent a variety of abstract objects, and the user can also define new data types which can be used exactly like built-in data types.

Each data type in Interlisp has an associated ''type name,'' a litatom. [1] Some of the type names of built-in data types are: LITATOM, LISTP, STRINGP, ARRAYP, STACKP, SMALLP, FIXP, and FLOATP. For user data types (page 3.14), the type name is specified when the data type is created.

(DATATYPES _)   [Function]
            Returns a list of all type names currently defined.

(TYPENAME DATUM)   [Function]
            Returns the type name for the data type of DATUM .

(TYPENAMEP DATUM TYPENAME)   [Function]
            Returns T if DATUM is an object with type name equal to TYPENAME , otherwise
            NIL.

Note: TYPENAME and TYPENAMEP distinguish the logical data types ARRAYP, CCODEP and HARRAYP, even though they may be implemented as ARRAYPs in some Interlisp implementations.

2.1 DATA TYPE PREDICATES

Interlisp provides seperate functions for testing whether objects are of certain commonly-used types:

(LITATOM x)   [Function]
            Returns T if x is a litatom, NIL otherwise. Note that a number is not a litatom.

(SMALLP x)   [Function]
            Returns x if x is a small integer; NIL otherwise. (Note that the range of small
            integers is implementation-dependent. See page 2.36.)

(FIXP x)   [Function]
            Returns x if x is a small or large integer (between MIN.FIXP and MAX.FIXP);
            NIL otherwise.

(FLOATP x)   [Function]
            Returns x if x is a floating point number; NIL otherwise.

---

[1] In Interlisp-10, each data type also has an associated ''type number.'' See page 22.2.

(NUMBERP x)   [Function]

> Returns x if x is a number of any type (FIXP or FLOATP), NIL otherwise.

(ATOM x)   [Function]

> Returns T if x is an atom (i.e. a litatom or a number); NIL otherwise.
>
> Warning: (ATOM x) is NIL if x is an array, string, etc. In many dialects of Lisp, the function ATOM is de ned equivalent to the Interlisp function NLISTP.

(LISTP x)   [Function]

> Returns x if x is a list cell, e.g., something created by CONS; NIL otherwise.

(NLISTP x)   [Function]

> (NOT (LISTP X)). Returns T if x is not a list cell, NIL otherwise.

(STRINGP x)   [Function]

> Returns x if x is a string, NIL otherwise.

(ARRAYP x)   [Function]

> Returns x if x is an array, NIL otherwise.
>
> Note: In some implementations of Interlisp, ARRAYP may also return x if it is of type CCODEP or HARRAYP.

(HARRAYP x)   [Function]

> Returns x if x is a hash array, NIL otherwise.

Note: The empty list, () or NIL, is considered to be a litatom, rather than a list. Therefore, (LITATOM NIL) = (ATOM NIL) = T and (LISTP NIL) = NIL. Care should be taken when using these functions if the object may be the empty list NIL.

2.2   DATA TYPE EQUALITY

A common operation when dealing with data objects is to test whether two objects are equal. In some cases, such as when comparing two small integers, equality can be easily determined. However, sometimes there is more than one type of equality. For instance, given two lists, one can ask whether they are exactly the same object, or whether they are two distinct lists which contain the same elements. Confusion between these two types of equality is often the source of program errors. Interlisp supplies an extensive set of functions for testing equality:

(EQ x y)   [Function]

> Returns T if x and y are identical pointers; NIL otherwise. EQ should not be used to compare two numbers, unless they are small integers; use EQP instead.

(NEQ x y)   [Function]

> (NOT (EQ x y))

(NULL x)   [Function]
(NOT x)   [Function]

> (EQ x NIL)

(EQP x y)   [Function]

> Returns `T` if `x` and `y` are `EQ`, or if `x` and `y` are numbers and are equal in value; `NIL` otherwise. For more discussion of `EQP` and other number functions, see page 2.36.

> Note: `EQP` also can be used to compare stack pointers (page 7.3) and compiled code (page 5.8).

(EQUAL x y)   [Function]

> `EQUAL` returns `T` if `x` and `y` are (1) `EQ`; or (2) `EQP`, i.e., numbers with equal value; or (3) `STREQUAL`, i.e., strings containing the same sequence of characters; or (4) lists and `CAR` of `x` is `EQUAL` to `CAR` of `y`, and `CDR` of `x` is `EQUAL` to `CDR` of `y`. `EQUAL` returns `NIL` otherwise. Note that `EQUAL` can be signi cantly slower than `EQ`.

> A loose description of `EQUAL` might be to say that `x` and `y` are `EQUAL` if they print out the same way.

(EQUALALL x y)   [Function]

> Like `EQUAL`, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-`EQ` arrays may be `EQUALALL` if their respective componants are `EQUALALL`.

2.3   "FAST" AND "DESTRUCTIVE" FUNCTIONS

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. The user should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by pre xing an existing function name with `F` indicates that the new function is a "fast" version of the old. These usually have the same de nitions as the slower versions, but they compile open and run without any "safety" error checks. For example, `FNTH` runs faster than `NTH`, however, it does not make as many checks (for lists ending with anything but `NIL`, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, the user should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by pre xing an existing function with `D` indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, `REMOVE` returns a copy of a list with a particular element removed, but `DREMOVE` actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of `APPEND` is `NCONC`.) The user should be careful when using destructive functions that they do not inadvertantly change data structures.

2.4   LITATOMS

A ''litatom'' (for ''literal atom'') is an object which conceptually consists of a print name, a value, a function de nition, and a property list. In some Lisp dialects, litatoms are also known as ''symbols.''

A litatom is read as any string of non- delimiting characters that cannot be interpreted as a number. The syntatic characters that delimit litatoms are called separator or break characters (see page 6.32) and normally are space, end- of-line, line-feed, ((left paren),) (right paren), " (double quote), [(left bracket), and] (right bracket). However, any character may be included in a litatom by preceding it with the escape character %. Here are some examples of litatoms:

```
A   wxyz   23SKIDDOO   %]   3.1415+17

Long% Litatom% With% Embedded% Spaces
```

Litatoms are printed by PRINT and PRIN2 as a sequence of characters with %'s inserted before all delimiting characters (so that the litatom will read back in properly). Litatoms are printed by PRIN1 as a sequence of characters without these extra %'s. For example, the litatom consisting of the ve characters A, B, C, (, and D will be printed as ABC%(D by PRINT and ABC(D by PRIN1.

Litatoms can also be constructed by PACK, PACK*, SUBATOM, MKATOM, and GENSYM (which uses MKATOM).

Litatoms are unique. In other words, if two litatoms print the same, they will *always* be EQ. Note that this is *not* true for strings, large integers, oating point numbers, and lists; they all can print the same without being EQ. Thus if PACK or MKATOM is given a list of characters corresponding to a litatom that already exists, they return a pointer to that litatom, and do *not* make a new litatom. Similarly, if the read program is given as input a sequence of characters for which a litatom already exists, it returns a pointer to that litatom. Note: Interlisp is di erent from other Lisp dialects which allow ''uninterned'' litatoms.

Note: Litatoms are limited to 255 characters in Interlisp- D; 127 characters in Interlisp- 10. Attempting to create a larger litatom either via PACK or by typing one in (or reading from a le) will cause an error, ATOM TOO LONG.

2.4.1   Using Litatoms as Variables

Litatoms are commonly used as variables. Each litatom has a ''top level'' variable binding, which can be an arbitrary Interlisp object. Litatoms may also be given special variable bindings within PROGs or function calls, which only exist for the duration of the function. When a litatom is evaluated, the ''current'' variable binding is returned. This is the most recent special variable binding, or the top level binding if the litatom has not been rebound. SETQ is used to change the current binding. For more information on variable bindings in Interlisp, see page 7.1.

Note: The compiler (page 12.1) treats variables somewhat di erently than the interpreter, and the user has to be aware of these di erences when writing functions that will be compiled. For example, variable references in compiled code are not checked for NOBIND, so compiled code will not generate unbound atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler o ers some facilities to increase the e ciency of variable use in compiled functions. Global variables (page 12.3) can be de ned so that the entire stack is not searched at each variable reference. Local variables (page 12.4) allow compiled functions to access variable bindings which are not on the stack,

which reduces variable con icts, and also makes variable lookup faster.

By convention, a litatom whose top level binding is to the litatom NOBIND is considered to have no top level binding. If a litatom has no local variable bindings, and its top level value is NOBIND, attempting to evaluate it will cause an unbound atom error.

The two litatoms T and NIL always evaluate to themselves. Attempting to change the binding of T or NIL with the functions below will generate the error ATTEMPT TO SET T or ATTEMPT TO SET NIL.

The following functions (except BOUNDP) will also generate the error ARG NOT LITATOM, if not given a litatom.

(BOUNDP VAR)   [Function]
> Returns T if VAR has a special variable binding (even if bound to NOBIND), or if VAR has a top level value other than NOBIND; otherwise NIL. In other words, if x is a litatom, (EVAL x) will cause an UNBOUND ATOM error if and only if (BOUNDP x) returns NIL.

(SET VAR VAL UE)   [Function]
> Sets the ''current'' variable binding of VAR to VAL UE, and returns VAL UE.
>
> Note that SET is a normal lambda spread function, so both VAR and VAL UE are evaluated before it is called. Thus, if the value of X is B, and the value of Y is C, then (SET X Y) would result in B being set to C, and C being returned as the value of SET.

(SETQ VAR VAL UE)   [NLambda NoSpread Function]
> Nlambda version of SET; VAR is not evaluated, VAL UE is.[2] Thus if the value of X is B and the value of Y is C, (SETQ X Y) would result in X (not B) being set to C, and C being returned.

(SETQQ VAR VAL UE)   [NLambda Function]
> Like SETQ except that neither argument is evaluated, e.g., (SETQQ X (A B C)) sets X to (A B C).

(GETTOPVAL VAR)   [Function]
> Returns the top level value of VAR (even if NOBIND), regardless of any intervening local bindings.

(SETTOPVAL VAR VAL UE)   [Function]
> Sets the top level value of VAR to VAL UE, regardless of any intervening bindings, and returns VAL UE.

A major di erence between various Interlisp implementations is the way that variable bindings are implemented. Interlisp- 10 and Interlisp- Jerico use what is called ''shallow'' binding. Interlisp- D and Interlisp- VAX use what is called ''deep'' binding.

---

[2]Since SETQ is an nlambda, *neither* argument is evaluated during the calling process. However, SETQ itself calls EVAL on its second argument. Note that as a result, typing (SETQ VAR FORM) and SETQ(VAR FORM) to the Interlisp executive is equivalent: in both cases VAR is not evaluated, and FORM is.

In a deep binding system, a variable is bound by saving on the stack the variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding. If the variable is not found on the stack, the top level binding is retrieved from a ''value cell'' associated with the variable.

In a ''shallow'' binding system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's value cell. When a variable is accessed, its value is always found in its value cell.

GETTOPVAL and SETTOPVAL are less efficient in a shallow binding system, because they have to search the stack for rebindings; it is more economical to simply rebind variables. In a deep binding system, GETTOPVAL and SETTOPVAL are very efficient since they do not have to search the stack, but can simply access the value cell directly.

GETATOMVAL and SETATOMVAL can be used to access a variable's value cell, in either a shallow or deep binding system.

(GETATOMVAL VAR)   [Function]
> Returns the value in the value cell of VAR. In a shallow binding system, this is the same as (EVAL ATM), or simply VAR. In a deep binding system, this is the same as (GETTOPVAL VAR).

(SETATOMVAL ATM VALUE)   [Function]
> Sets the value cell of VAR to VALUE. In a shallow binding system, this is the same as SET; in a deep binding system, this is the same as SETTOPVAL.

2.4.2   Function Definition Cells

Each litatom has a function definition cell, which is accessed when a litatom is used as a function. The mechanism for accessing and setting the function definition cell of a litatom is described on page 5.8.

2.4.3   Property Lists

Each litatom has a property list, which allows a set of named objects to be associated with the litatom. A property list associates a name, known as a ''property name'' or ''property'', with an abitrary object, the ''property value'' or simply ''value''. Sometimes the phrase ''to store on the property x'' is used, meaning to place the indicated information on a property list under the property name x.

Property names are usually litatoms or numbers, although no checks are made. However, the standard property list functions all use EQ to search for property names, so they may not work with non-atomic property names. Note that the same object can be used as both a property name and a property value.

Note: Many litatoms in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The variable SYSPROPS is a list of property names used by the system.

The functions below are used to manipulate the propert lists of litatoms. Except when indicated, they generate the error ARG NOT LITATOM, if given an object that is not a litatom.

DATA TYPES

(GETPROP ATM PROP) [Function]

Returns the property value for PROP from the property list of ATM. Returns NIL if ATM is not a litatom, or PROP is not found. Note that GETPROP also returns NIL if there is an occurrence of PROP but the corresponding property value is NIL; this can be a source of program errors.

Note: GETPROP used to be called GETP.

(PUTPROP ATM PROP VAL) [Function]

Puts the property PROP with value VAL on the property list of ATM. VAL replaces any previous value for the property PROP on this property list. Returns VAL.

(ADDPROP ATM PROP NEW FLG) [Function]

Adds the value NEW to the list which is the value of property PROP on the property list of ATM. If FLG is T, NEW is CONSed onto the front of the property value of PROP, otherwise it is NCONCed on the end (using NCONC1). If ATM does not have a property PROP, or the value is not a list, then the effect is the same as (PUTPROP ATM PROP (LIST NEW)). ADDPROP returns the (new) property value. Example:

```
_ (PUTPROP 'POCKET 'CONTENTS NIL)
NIL
_ (ADDPROP 'POCKET 'CONTENTS 'COMB)
(COMB)
_ (ADDPROP 'POCKET 'CONTENTS 'WALLET)
(COMB WALLET)
```

(REMPROP ATM PROP) [Function]

Removes all occurrences of the property PROP (and its value) from the property list of ATM. Returns PROP if any were found, otherwise NIL.

(REMPROPLIST ATM PROPS) [Function]

Removes all occurrences of all properties on the list PROPS (and their corresponding property values) from the property list of ATM. Returns NIL.

(CHANGEPROP X PROP1 PROP2) [Function]

Changes the property name of property PROP1 to PROP2 on the property list of X, (but does not affect the value of the property). Returns X, unless PROP1 is not found, in which case it returns NIL.

(PROPNAMES ATM) [Function]

Returns a list of the property names on the property list of ATM.

(DEFLIST L PROP) [Function]

Used to put values under the same property name on the property lists of several litatoms. L is a list of two-element lists. The first element of each is a litatom, and the second element is the property value for the property PROP. Returns NIL. For example,

(DEFLIST '((FOO MA) (BAR CA) (BAZ RI)) 'STATE)

puts MA on FOO's STATE property, CA on BAR's STATE property, and RI on BAZ's

STATE property.

Property lists are conventionally implemented as lists of the form

(NAME $_1$ VALUE$_1$ NAME $_2$ VALUE$_2$)

although the user can store anything as the property list of a litatom. However, the functions which manipulate property lists observe this convention by searching down the property lists two CDRs at a time. Most of these functions also generate an error, ARG NOT LITATOM, if given an argument which is not a litatom, so they cannot be used directly on lists. (LISTPUT, LISTPUT1, LISTGET, and LISTGET1 are functions similar to PUTPROP and GETPROP that work directly on lists. See page 2.26.) The property lists of litatoms can be directly accessed with the following functions:

(GETPROPLIST ATM)   [Function]
              Returns the property list of ATM .

(SETPROPLIST ATM LST)   [Function]
              If ATM is a non-NIL litatom, sets the property list of ATM to be LST , and returns LST
              as its value. If ATM is NIL, generates the error, ATTEMPT TO RPLAC NIL (unless
              LST is also NIL).

(GETLIS X PR OPS)   [Function]
              Searches the property list of x, and returns the property list as of the rst property
              on PR OPS that it nds. For example,

              _ (GETPROPLIST 'X)
              (PROP1 A PROP3 B A C)
              _ (GETLIS 'X '(PROP2 PROP3))
              (PROP3 B A C)

              Returns NIL if no element on PR OPS is found. x can also be a list itself, in which
              case it is searched as described above. If x is not a litatom or a list, returns NIL.

2.4.4   Print Names

Each litatom has a print name, a string of characters that uniquely identi es that litatom. The term ''print name'' has been extended, however, to refer to the characters that are output when any object is printed. In Interlisp, all objects have print names, although only litatoms and strings have their print name explicitly stored. This section describes a set of functions which can be used to access and manipulate the print names of any object, though they are primarily used with the print names of litatoms.

The print name of an object is those characters that are output when the object is printed using PRIN1, e.g., the print name of the litatom ABC%(D consists of the ve characters ABC(D. The print name of the list (A B C) consists of the seven characters (A B C) (two of the characters are spaces).

Sometimes we will have occasion to refer to a ''PRIN2-name.'' The PRIN2-name of an object is those characters output when the object is printed using PRIN2. Thus the PRIN2-name of the litatom ABC%(D is the *six* characters ABC%(D. Note that the PRIN2-name depends on what readtable is being used (see page 6.32), since this determines where %'s will be inserted. Many of the functions below allow either print names or PRIN2-names to be used, as speci ed by FLG and RDTBL arguments. If FLG is NIL, print names are used. Otherwise, PRIN2-names are used, computed with respect to the readtable RDTBL (or

the current readtable, if RDTBL = NIL).

Note: The print name of an integer depends on the setting of RADIX (page 6.19). The functions described in this section (UNPACK, NCHARS, etc.) de ne the print name of an integer as though the radix was 10, so that (PACK (UNPACK 'X9)) will always be X9 (and not sometimes X11) regardless of the setting of RADIX. However, integers will still be *printed* by PRIN1 using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable PRXFLG (see page 6.20).

(MKATOM x) [Function]

        Creates and returns an atom whose print name is the same as that of the string x or, if x isn't a string, the same as that of (MKSTRING x). Examples:

        (MKATOM '(A B C)) => %(A% B% C%)

        (MKATOM "1.5") => 1.5

        Note that the last example returns a number, not a litatom. It is a deeply-ingrained feature of Interlisp that no litatom can have the print name of a number.

(SUBATOM x N M) [Function]

        Equivalent to (MKATOM (SUBSTRING x N M)), but does not make a string pointer (see page 2.29). Returns an atom made from the Nth through Mth characters of the print name of x. If N or M are negative, they specify positions counting backwards from the end of the print name. Examples:

        (SUBATOM "FOO1.5BAR" 4 6) => 1.5

        (SUBATOM '(A B C) 2 -2) => A% B% C

(PACK x) [Function]

        If x is a list of atoms, PACK returns a single atom whose print name is the concatenation of the print names of the atoms in x. If the concatenated print name is the same as that of a number, PACK will return that number. For example,

        (PACK '(A BC DEF G)) => ABCDEFG

        (PACK '(1 3.4)) => 13.4

        (PACK '(1 E -2)) => .01

        Although x is usually a list of atoms, it can be a list of arbitrary Interlisp objects. The value of PACK is still a single atom whose print name is the concatenation of the print names of all the elements of x, e.g.,

        (PACK '((A B) "CD")) => %(A% B%)CD

        If x is not a list or NIL, PACK generates an error, ILLEGAL ARG.

(PACK* x_1 x_2 x_N) [NoSpread Function]

        Nospread version of PACK that takes an arbitrary number of arguments, instead of a list. Examples:,

```
(PACK* 'A 'BC 'DEF 'G)  =>  ABCDEFG

(PACK* 1 3.4)  =>  13.4
```

(UNPACK X FLG RDTBL)   [Function]
> Returns the print name of x as a list of single-characters atoms, e.g.,

```
(UNPACK 'ABC5D)  =>  (A B C 5 D)

(UNPACK "ABC(D")  =>  (A B C %( D)
```

> If FLG= T, the PRIN2-name of x is used (computed with respect to RDTBL), e.g.,

```
(UNPACK "ABC(D" T)  =>  (%" A B C %( D %")

(UNPACK 'ABC%(D" T)  =>  (A B C %% %( D)
```

> Note: (UNPACK x) performs N CONSes, where N is the number of characters in the print name of x.

(DUNPACK X SCRA TCHLIST FLG RDTBL)   [Function]
> A destructive version of UNPACK that does not perform any CONSes but instead reuses the list SCRA TCHLIST . If the print name is too long to t in SCRA TCHLIST , DUNPACK will extend it. If SCRA TCHLIST is not a list, DUNPACK returns (UNPACK X FLG RDTBL).

(NCHARS X FLG RDTBL)   [Function]
> Returns the number of characters in the print name of x. If FLG= T, the PRIN2-name is used. For example,

```
(NCHARS "ABC")  =>  3

(NCHARS "ABC" T)  =>  5
```

(NTHCHAR X N FLG RDTBL)   [Function]
> Returns the Nth character of the print name of x as an atom. N can be negative, in which case it counts from the end of the print name, e.g., -1 refers to the last character, -2 next to last, etc. If N is greater than the number of characters in the print name, or less than minus that number, or 0, NTHCHAR returns NIL. Examples:

```
(NTHCHAR 'ABC 2)  =>  B

(NTHCHAR 15.6 2)  =>  5

(NTHCHAR 'ABC%(D -3 T)  =>  %%

(NTHCHAR "ABC" 2)  =>  B

(NTHCHAR "ABC" 2 T)  =>  A
```

Note: NTHCHAR and NCHARS work much faster on objects that actually have an internal representation of their print name, i.e., litatoms and strings, than they do on numbers and lists, as they do not have to simulate printing.

DATA TYPES

(L-CASE X FLG) [Function]

      Returns a lower case version of x. If FLG is T, the rst letter is capitalized. If x is a string, the value of L-CASE is also a string. If x is a list, L-CASE returns a new list in which L-CASE is computed for each corresponding element and non-NIL tail of the original list. Examples:

```
(L-CASE 'FOO)  =>  foo

(L-CASE 'FOO T)  =>  Foo

(L-CASE "FILE NOT FOUND" T)  =>  "File not found"

(L-CASE '(JANUARY FEBRUARY (MARCH "APRIL")) T)
          =>  '(January February (March "April"))
```

(U-CASE x) [Function]

      Similar to L-CASE, except returns the upper case version of x.

(U-CASEP x) [Function]

      Returns T if x contains no lower case letters; NIL otherwise.

(GENSYM CHAR) [Function]

      Returns a litatom of the form Xnnnn, where X= CHAR (or A if CHAR is NIL) and nnnn is an integer. Thus, the rst one generated is A0001, the second A0002, etc. GENSYM provides a way of generating litatoms for various uses within the system.

GENNUM [Variable]

      The value of GENNUM, initially 10000, determines the next GENSYM, e.g., if GENNUM is set to 10023, (GENSYM)= A0024.

The term ''gensym'' is used to indicate a litatom that was produced by the function GENSYM. Litatoms generated by GENSYM are the same as any other litatoms: they have property lists, and can be given function de nitions. Note that the litatoms are not guaranteed to be new. For example, if the user has previously created A0012, either by typing it in, or via PACK or GENSYM itself, when GENNUM gets to 10011, the next litatom returned by GENSYM will be the A0012 already in existence.

(MAPATOMS FN) [Function]

      Applies FN (a function or lambda expression) to every litatom in the system. Returns NIL

      For example,

```
(MAPATOMS (FUNCTION (LAMBDA(X)
                        (if (GETD X) then (PRINT X)]
```

      will print every litatom with a function de nition.

      Note: In some implementations of Interlisp, unused litatoms may be garbage collected, which can e ect the action of MAPATOMS.

2.4.5 Character Code Functions

Characters may be represented in two ways: as single-character atoms, or as integer character codes.[3] In many situations, it is more e cient to use character codes, so Interlisp provides parallel functions for both representations.

(PACKC x) [Function]

>Similar to PACK except x is a list of character codes. For example,
>
>(PACKC '(70 79 79)) => FOO

(CHCON X FLG RDTBL) [Function]

>Like UNPACK, except returns the print name of x as a list of character codes. If FLG = T, the PRIN2-name is used. For example,
>
>(CHCON 'FOO) => (70 79 79)

(DCHCON X SCRA TCHLIST FLG RDTBL) [Function]

>Similar to DUNPACK.

(NTHCHARCODE X N FLG RDTBL) [Function]

>Similar to NTHCHAR, except returns the character code of the N th character of the print name of x. If N is negative, it is interpreted as a count backwards from the end of x. If the absolute value of N is greater than the number of characters in x, or 0, then the value of NTHCHARCODE is NIL.
>
>If FLG is T, then the PRIN2-name of x is used, computed with respect to the readtable RDTBL

(CHCON1 x) [Function]

>Returns the character code of the rst character of the print name of x; equal to (NTHCHARCODE x 1).

(CHARACTER N) [Function]

>N is a character code. Returns the atom having the corresponding single character as its print name.
>
>(CHARACTER 70) => F

(FCHARACTER N) [Function]

>Fast version of CHARACTER that compiles open.

The following function makes it possible to gain the e ciency that comes from dealing with character codes without losing the symbolic advantages of character atoms:

(CHARCODE C) [NLambda Function]

>Returns the character code structure speci ed by C (unevaluated). If C is a 1-character atom or string, the corresponding character code is simply returned.

---

[3]Interlisp- D uses an 8-bit character set, so the legal character codes range from 0 to 255. Interlisp- 10 uses standard 7-bit ASCII, so the range is 0-127.

Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If C is a list structure, the value is a copy of C with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67))

CHARCODE permits easy speci cation of non- printable ASCII character codes: A multi- character litatom or string whose rst character is ^ is interpreted as the control- character corresponding to its second character. Thus, (CHARCODE ^A) is 1, the code for control- A.

Also, if a multi- character litatom or string begins with #, this signi es a ''meta- character'', with a code between 128 to 255. # and ^ may be combined, so (CHARCODE #^A) is 129. (Note: Interlisp- 10 cannot directly represent meta- characters as character litatoms, because it only supports 7-bit characters.)

The following key litatoms are mapped into the indicated codes: CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). The litatom EOL maps into the appropriate End- Of-Line character code in the di erent Interlisp implementations (31 in Interlisp- 10, 13 in Interlisp- D, 10 in Interlisp- VAX).

Finally, CHARCODE maps NIL into NIL. This is included because some character-code producing functions sometimes return NIL (e.g. NTHCHARCODE); a test for that value can be included in a CHARCODE list along with true character- code values.

Charcode of litatomic arguments can be used wherever a structure of character codes would be appropriate. For example:

```
(FMEMB (NTHCHARCODE X 1) (CHARCODE (CR LF SPACE)))
(EQ (BIN FOO) (CHARCODE ^C))
```

There is a macro for CHARCODE which causes the character- code structure to be constructed at compile- time. Thus, the compiled code for these examples is exactly as e cient as the less readable:

```
(FMEMB (NTHCHARCODE X 1) (QUOTE (13 10 32)))
(EQ (BIN FOO) 3)
```

(SELCHARQ E CLAUSE$_1$ CLAUSE$_N$ DEFAULT) [NLambda NoSpread Function]
Similar to SELECTQ (page 4.2), except that the selection keys are determined by applying CHARCODE (instead of QUOTE) to the key-expressions. If the value of E is a character code or NIL and it is EQ or MEMB to the result of applying CHARCODE to the rst element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

```
(SELCHARQ (BIN FOO)
    ((SPACE TAB) (FUM))
    ((^D NIL) (BAR))
    (a (BAZ))
    (ZIP))
```

is exactly equivalent to

```
(SELECTQ (BIN FOO)
    ((32 9) (FUM))
    ((4 NIL) (BAR))
    (97 (BAZ))
    (ZIP))
```

Furthermore, `SELCHARQ` has a macro such that it always compiles as an equivalent `SELECTQ`.

2.5   LISTS

One of the most useful datatypes in Interlisp is the list cell, a data structure which contains pointers to two other objects, known as the `CAR` and the `CDR` of the list cell (after the accessing functions). Very complicated structures can be built out of list cells, including lattices and trees, but list cells are most frequently used for representing simple linear lists of objects.

The following functions are used to manipulate list cells:

`(CONS X Y)`   [Function]

> `CONS` is the primary list construction function. It creates and returns a new list cell containing pointers to `x` and `y`. If `y` is a list, this returns a list with `x` added at the beginning of `y`.

`(CAR X)`   [Function]

> Returns the rst element of the list `x`. `CAR` of `NIL` is always `NIL`. For all other nonlists (e.g., litatoms, numbers, strings, arrays), the value is unde ned (and in some implementations may generate an error).

`(CDR X)`   [Function]

> Returns all but the rst element of the list `x`. `CDR` of `NIL` is always `NIL`. The value of `CDR` is unde ned for other nonlists.

Often, combinations of the `CAR` and `CDR` functions are used to extract various components of complex list structures. Functions of the form C R may be used for some of these combinations:

`(CAAR X)  ==>  (CAR (CAR X))`

`(CADR X)  ==>  (CAR (CDR X))`

`(CDDDDR X)  ==>  (CDR (CDR (CDR (CDR X))))`

All 30 combinations of nested `CAR`s and `CDR`s up to 4 deep are included in the system.

`(RPLACD X Y)`   [Function]

> Replaces the `CDR` of the list cell `x` with `y`. This physically changes the internal structure of `x`, as opposed to `CONS`, which creates a new list cell. It is possible to construct a circular list by using `RPLACD` to place a pointer to the beginning of a list in a spot at the end of the list.

The value of RPLACD is x. An attempt to RPLACD NIL will cause an error, ATTEMPT TO RPLAC NIL (except for (RPLACD NIL NIL)). An attempt to RPLACD any other non-list will cause an error, ARG NOT LIST.

(RPLACA X Y)   [Function]
Similar to RPLACD, but replaces the CAR of x with Y. The value of RPLACA is x. An attempt to RPLACA NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for (RPLACA NIL NIL)). An attempt to RPLACA any other non-list will cause an error, ARG NOT LIST.

(RPLNODE X A D)   [Function]
Performs (RPLACA X A), (RPLACD X D), and returns x.

(RPLNODE2 X Y)   [Function]
Performs (RPLACA X (CAR Y)), (RPLACD X (CDR Y)) and returns x.

(FRPLACD X Y)   [Function]
(FRPLACA X Y)   [Function]
(FRPLNODE X A D)   [Function]
(FRPLNODE2 X Y)   [Function]
Faster versions of RPLACD, etc.

Warning: In Interlisp-10 and Interlisp-VAX, these functions compile open with no error checks on the type of x, so a compiled FRPLACD can produce unpredictable e ects.

Usually, single list cells are not manipulated in isolation, but in structures known as ''lists''. By convention, a list is represented by a list cell whose CAR is the rst element of the list, and whose CDR is the rest of the list (usually another list cell or the ''empty list,'' NIL). List elements may be any Interlisp objects, including other lists.

The input syntax for a list is a sequence of Interlisp data objects (litatoms, numbers, other lists, etc.) enclosed in parentheses or brackets. Note that ( ) is read as the litatom NIL. A right bracket can be used to match all left parenthesis back to the last left bracket, or terminate the lists, e.g. (A (B (C].

If there are two or more elements in a list, the nal element can be preceded by a period delimited on both sides, indicating that CDR of the nal list cell in the list is to be the element immediately following the period, e.g. (A . B) or (A B C . D), otherwise CDR of the last list cell in a list will be NIL. Note that a list does not have to end in NIL. It is simply a structure composed of one or more list cells. The input sequence (A B C . NIL) is equivalent to (A B C), and (A B . (C D)) is equivalent to (A B C D). Note however that (A B . C D) will create a list containing the ve litatoms A, B, %., C, and D.

Lists are printed by printing a left parenthesis, and then printing the rst element of the list, then printing a space, then printing the second element, etc. until the nal list cell is reached. The individual elements of a list are printed by PRIN1 if the list is being printed by PRIN1, and by PRIN2 if the list is being printed by PRINT or PRIN2. Lists are considered to terminate when CDR of some node is not a list. If CDR of this terminal node is NIL (the usual case), CAR of the terminal node is printed followed by a right parenthesis. If CDR of the terminal node is *not* NIL, CAR of the terminal node is printed, followed by a space, a period, another space, CDR of the terminal node, and then the right parenthesis. Note that a list input as (A B C . NIL) will print as (A B C), and a list input as (A B . (C D)) will print as (A B C D). Note also that PRINTLEVEL a ects the printing of lists (page 6.18), and that carriage

returns may be inserted where dictated by LINELENGTH (page 6.8).

Note: One must be careful when testing the equality of list structures. EQ will be true only when the two lists are the *exact* same list. For example,

```
_ (SETQ A '(1 2))
(1 2)
_ (SETQ B A)
(1 2)
_ (EQ A B)
T
_ (SETQ C '(1 2))
(1 2)
_ (EQ A C)
NIL
_ (EQUAL A C)
T
```

In the example above, the values of A and B are the exact same list, so they are EQ. However, the value of C is a totally di erent list, although it happens to have the same elements. EQUAL should be used to compare the elements of two lists. In general, one should notice whether list manipulation functions use EQ or EQUAL for comparing lists. This is a frequent source of errors.

Interlisp provides an extensive set of list manipulation functions:

2.5.1   Creating Lists

(MKLIST x)   [Function]
      ''Make List.'' If x is a list or NIL, returns x; Otherwise, returns (LIST x).

(LIST x_1 x_2 x_N)   [NoSpread Function]
     Returns a list of its arguments, e.g.

     (LIST 'A 'B '(C D)) => (A B (C D))

(APPEND x_1 x_2 x_N)   [NoSpread Function]
     Copies the top level of the list x_1 and appends this to a copy of the top level of the list x_2 appended to   appended to x_N, e.g.,

     (APPEND '(A B) '(C D E) '(F G)) => (A B C D E F G)

     Note that only the rst N-1 lists are copied. However N=1 is treated specially; (APPEND X) copies the top level of a single list. To copy a list to all levels, use COPY.

     The following examples illustrate the treatment of non-lists:

     (APPEND '(A B C) 'D) => (A B C . D)

     (APPEND 'A '(B C D)) => (B C D)

```
(APPEND '(A B C . D) '(E F G))  =>  (A B C E F G)

(APPEND '(A B C . D))  =>  (A B C . D)
```

(NCONC x_1 x_2 x_N)   [NoSpread Function]

Returns the same value as APPEND, but actually modies the list structure of x_1
x_{n-1}.

Note that NCONC cannot change NIL to a list:

```
_(SETQ FOO NIL)
NIL
_(NCONC FOO '(A B C))
(A B C)
_FOO
NIL
```

Although the value of the NCONC is (A B C), FOO has *not* been changed. The
''problem'' is that while it is possible to alter list structure with RPLACA and
RPLACD, there is no way to change the non-list NIL to a list.

(NCONC1 LST x)   [Function]

```
(NCONC LST (LIST x))
```

(ATTACH X L)   [Function]

''Attaches'' x to the front of L by doing a RPLACA and RPLACD. The value is
EQUAL to (CONS x L), but EQ to L, which it physically changes (except if L is
NIL). (ATTACH X NIL) is the same as (CONS X NIL). Otherwise, if L is not
a list, an error is generated, ARG NOT LIST.

2.5.2   Building Lists From Left to Right

(TCONC PTR x)   [Function]

TCONC is similar to NCONC1; it is useful for building a list by adding elements one
at a time at the end. Unlike NCONC1, TCONC does not have to search to the end
of the list each time it is called. Instead, it keeps a pointer to the end of the list
being assembled, and updates this pointer after each call. This can be considerably
faster for long lists. The cost is an extra list cell, PTR . (CAR PTR) is the list being
assembled, (CDR PTR) is (LAST (CAR PTR)). TCONC returns PTR , with its
CAR and CDR appropriately modied.

PTR can be initialized in two ways. If PTR is NIL, TCONC will create and return a
PTR . In this case, the program must set some variable to the value of the rst call
to TCONC. After that, it is unnecessary to reset the variable, since TCONC physically
changes its value. Example:

```
_(SETQ FOO (TCONC NIL 1))
((1) 1)
_(for I from 2 to 5 do (TCONC FOO I))
NIL
_FOO
```

```
((1 2 3 4 5) 5)
```

If PTR is initially (NIL), the value of TCONC is the same as for PTR = NIL. but TCONC changes PTR . This method allows the program to initialize the TCONC variable before adding any elements to the list. Example:

```
_(SETQ FOO (CONS))
(NIL)
_(for I from 1 to 5 do (TCONC FOO I))
NIL
_FOO
((1 2 3 4 5) 5)
```

(LCONC PTR X)   [Function]

Where TCONC is used to add *elements* at the end of a list, LCONC is used for building a list by adding *lists* at the end, i.e., it is similar to NCONC instead of NCONC1. Example:

```
_(SETQ FOO (CONS))
(NIL)
_(LCONC FOO '(1 2))
((1 2) 2)
_(LCONC FOO '(3 4 5))
((1 2 3 4 5) 5)
_(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

LCONC uses the same pointer conventions as TCONC for eliminating searching to the end of the list, so that the same pointer can be given to TCONC and LCONC interchangeably. Therefore, continuing from above,

```
_(TCONC FOO NIL)
((1 2 3 4 5 NIL) NIL)
_(TCONC FOO '(3 4 5))
((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

The functions DOCOLLECT and ENDCOLLECT also permit building up lists from left-to-right like TCONC, but without the overhead of an extra list cell. The list being maintained is kept as a circular list. DOCOLLECT adds items; ENDCOLLECT replaces the tail with its second argument, and returns the full list.

(DOCOLLECT ITEM LST)   [Function]

''Adds'' ITEM to the end of LST . Returns the new circular list. Note that LST is modi ed, but it is not EQ to the new list. The new list should be stored and used as LST to the next call to DOCOLLECT.

(ENDCOLLECT LST TAIL)   [Function]

Takes LST , a list returned by DOCOLLECT, and returns it as a non-circular list, adding TAIL as the terminating CDR.

Here is an example using DOCOLLECT and ENDCOLLECT. HPRINT is used to print the results because they are circular lists. Notice that FOO has to be set to the value of DOCOLLECT as each element is

added.

```
_(SETQ FOO NIL]
NIL
_(HPRINT (SETQ FOO (DOCOLLECT 1 FOO]
^(1 . {1})
_(HPRINT (SETQ FOO (DOCOLLECT 2 FOO]
^(2 1 . {1})
_(HPRINT (SETQ FOO (DOCOLLECT 3 FOO]
^(3 1 2 . {1})
_(HPRINT (SETQ FOO (DOCOLLECT 4 FOO]
^(4 1 2 3 . {1})
_(SETQ FOO (ENDCOLLECT FOO 5]
(1 2 3 4 . 5)
```

2.5.3   Copying Lists

(COPY x)   [Function]
>　Creates and returns a copy of the list x. All levels of x are copied down to non-lists, so that if x contains arrays and strings, the copy of x will contain the same arrays and strings, not copies. COPY is recursive in the CAR direction only, so very long lists can be copied.
>
>　Note: To copy just the *top level* of x, do (APPEND x).

(COPYALL x)   [Function]
>　Like COPY except copies down to atoms. Arrays, hash-arrays, strings, user data types, etc., are all copied. Analagous to EQUALALL (page 2.3). Note that this will not work if given a data structure with circular pointers; in this case, use HCOPYALL.

(HCOPYALL x)   [Function]
>　Similar to COPYALL, except that it will work even if the data structure contains circular pointers.

2.5.4   Extracting Tails of Lists

(TAILP x y)   [Function]
>　Returns x, if x is a *tail* of the list y; otherwise NIL. x is a tail of y if it is EQ to 0 or more CDRs of y.
>
>　Note: If x is EQ to 1 or more CDRs of y, x is called a ''proper tail.''

(NTH x n)   [Function]
>　Returns the tail of x beginning with the nth element. Returns NIL if x has fewer than n elements. Examples:
>
>　(NTH '(A B C D) 1) => (A B C D)

```
(NTH '(A B C D) 3)  =>  (C D)

(NTH '(A B C D) 9)  =>  NIL

(NTH '(A . B) 2)  =>  B
```

For consistency, if N=0, NTH returns (CONS NIL x):

```
(NTH '(A B) 0)  =>  (NIL A B)
```

(FNTH x N) [Function]
Faster version of NTH that terminates on a null-check.

(Interlisp-10) Interpreted, generates an error, BAD ARGUMENT - FNTH, if x ends in other than NIL.

(LAST x) [Function]
Returns the last list cell in the list x. Returns NIL if x is not a list. Examples:

```
(LAST '(A B C))  =>  (C)

(LAST '(A B . C))  =>  (B . C)

(LAST 'A)  =>  NIL
```

(FLAST x) [Function]
Faster version of LAST that terminates on a null-check.

(Interlisp-10) Interpreted, generates an error, BAD ARGUMENT - FLAST, if x ends in other than NIL.

(NLEFT L N TAIL) [Function]
NLEFT returns the tail of L that contains N more elements than TAIL. If L does not contain N more elements than TAIL, NLEFT returns NIL. If TAIL is NIL or not a tail of L, NLEFT returns the last N list cells in L. NLEFT can be used to work backwards through a list. Example:

```
_(SETQ FOO '(A B C D E))
(A B C D E)
_(NLEFT FOO 2)
(D E)
_(NLEFT FOO 1 (CDDR FOO))
(B C D E)
_(NLEFT FOO 3 (CDDR FOO))
NIL
```

(LASTN L N) [Function]
Returns (CONS X Y), where Y is the last N elements of L, and X is the initial segment, e.g.,

```
(LASTN '(A B C D E) 2)  =>  ((A B C) D E)

(LASTN '(A B) 2)  =>  (NIL A B)
```

Returns `NIL` if `L` is not a list containing at least `N` elements.

2.5.5   Counting List Cells

`(LENGTH x)`   [Function]

>Returns the length of the list `x`, where ''length'' is de ned as the number of `CDR`s required to reach a non-list. Examples:
>
>`(LENGTH '(A B C))  =>  3`
>
>`(LENGTH '(A B C . D))  =>  3`
>
>`(LENGTH 'A)  =>  0`

`(FLENGTH x)`   [Function]

>Faster version of `LENGTH` that terminates on a null-check.
>
>(Interlisp-10) Interpreted, generates an error, `BAD ARGUMENT - FLENGTH`, if `x` ends in other than `NIL`.

`(EQLENGTH x N)`   [Function]

>Equivalent to `(EQUAL (LENGTH x) N)`, but more e cient, because `EQLENGTH` stops as soon as it knows that `x` is longer than `N`. Note that `EQLENGTH` is safe to use on (possibly) circular lists, since it is ''bounded'' by `N`.

`(COUNT x)`   [Function]

>Returns the number of list cells in the list `x`. Thus, `COUNT` is like a `LENGTH` that goes to all levels. `COUNT` of a non-list is 0. Examples:
>
>`(COUNT '(A))  =>  1`
>
>`(COUNT '(A . B))  =>  1`
>
>`(COUNT '(A (B) C))  =>  4`
>
>In this last example, the value is 4 because the list `(A x C)` uses 3 list cells for any object `x`, and `(B)` uses another list cell.

`(COUNTDOWN x N)`   [Function]

>Counts the number of list cells in `x`, decrementing `N` for each one. Stops and returns `N` when it nishes counting, or when `N` reaches 0. `COUNTDOWN` can be used on circular structures since it is ''bounded'' by `N`. Examples:
>
>`(COUNTDOWN '(A) 100)  =>  99`
>
>`(COUNTDOWN '(A . B) 100)  =>  99`
>
>`(COUNTDOWN '(A (B) C) 100)  =>  96`
>
>`(COUNTDOWN '(DOCOLLECT 1 NIL) 100)  =>  0`

(EQUALN X Y DEPTH)   [Function]
> Similar to EQUAL, for use with (possibly) circular structures. Whenever the depth
> of CAR recursion plus the depth of CDR recursion exceeds DEPTH , EQUALN does
> not search further along that chain, and returns the litatom ?. If recursion never
> exceeds DEPTH , EQUALN returns T if the expressions X and Y are EQUAL; otherwise
> NIL.
>
> (EQUALN ’(((A)) B) ’(((Z)) B) 2)   =>   ?
>
> (EQUALN ’(((A)) B) ’(((Z)) B) 3)   =>   NIL
>
> (EQUALN ’(((A)) B) ’(((A)) B) 3)   =>   T

2.5.6   Logical Operations

(LDIFF X Y Z)   [Function]
> Y must be a tail of X, i.e., EQ to the result of applying some number of CDRs to
> X. (LDIFF X Y) returns a list of all elements in X up to Y.
>
> If Z is not NIL, the value of LDIFF is e ectively (NCONC Z (LDIFF X Y)),
> i.e., the list di erence is added at the end of Z.
>
> If Y is not a tail of X, LDIFF generates an error, LDIFF: NOT A TAIL. LDIFF
> terminates on a null- check, so it will go into an in nite loop if X is a circular list
> and Y is not a tail.
>
> Example:
>
> ```
> _(SETQ FOO ’(A B C D E F))
> (A B C D E F)
> _(CDDR FOO)
> (C D E F)
> _(LDIFF FOO (CDDR FOO))
> (A B)
> _(LDIFF FOO (CDDR FOO) ’(1 2))
> (1 2 A B)
> _(LDIFF FOO ’(C D E F))
> LDIFF: not a tail
> (C D E F)
> ```
>
> Note that the value of LDIFF is always new list structure unless Y= NIL, in which
> case the value is X itself.

(LDIFFERENCE X Y)   [Function]
> ‘‘List Di erence.’’   Returns a list of those elements in X that are not members of
> Y.

(INTERSECTION X Y)   [Function]
> Returns a list whose elements are members of both lists X and Y. Note that
> (INTERSECTION X X) gives a list of all members of X without any duplications.

`(UNION x Y)` [Function]

Returns a (new) list consisting of all elements included on either of the two original lists. It is more e cient to make x be the shorter list.

The value of `UNION` is `Y` with all elements of x not in `Y` `CONS`ed on the front of it. Therefore, if an element appears twice in `Y`, it will appear twice in `(UNION x Y)`. Since `(UNION '(A) '(A A))` = `(A A)`, while `(UNION '(A A) '(A))` = `(A)`, `UNION` is non-commutative.

2.5.7   Searching Lists

`(MEMB x Y)` [Function]

Determines if x is a member of the list `Y`. If there is an element of `Y` `EQ` to x, returns the tail of `Y` starting with that element. Otherwise, returns `NIL`. Examples:

```
(MEMB 'A '(A (W) C D))   =>   (A (W) C D)

(MEMB 'C '(A (W) C D))   =>   (C D)

(MEMB 'W '(A (W) C D))   =>   NIL

(MEMB '(W) '(A (W) C D))   =>   NIL
```

`(FMEMB x Y)` [Function]

Faster version of `MEMB` that terminates on a null-check.

(Interlisp-10) Interpreted, `FMEMB` gives an error, `BAD ARGUMENT – FMEMB`, if `Y` ends in a non-list other than `NIL`.

`(MEMBER x Y)` [Function]

Identical to `MEMB` except that it uses `EQUAL` instead of `EQ` to check membership of x in `Y`. Examples:

```
(MEMBER 'C '(A (W) C D))   =>   (C D)

(MEMBER 'W '(A (W) C D))   =>   NIL

(MEMBER '(W) '(A (W) C D))   =>   ((W) C D)
```

`(EQMEMB x Y)` [Function]

Returns `T` if either x is `EQ` to `Y`, or else `Y` is a list and x is an `FMEMB` of `Y`.

2.5.8   Substitution Functions

`(SUBST NEW  OLD  EXPR )` [Function]

Returns the result of substituting `NEW` for all occurrences of `OLD` in the expression `EXPR` . Substitution occurs whenever `OLD` is `EQUAL` to `CAR` of some subexpression of `EXPR` , or when `OLD` is atomic and `EQ` to a non-`NIL` `CDR` of some subexpression of `EXPR` . For example:

```
(SUBST 'A 'B '(C B (X . B)))  =>  (C A (X . A))

(SUBST 'A '(B C) '((B C) D B C))
          =>  (A D B C)  not  (A D . A)
```

SUBST returns a copy of EXPR with the appropriate changes. Furthermore, if NEW is a list, it is copied at each substitution.

(DSUBST NEW OLD EXPR) [Function]
> Similar to SUBST, except it does not copy EXPR, but changes the list structure EXPR itself. Like SUBST, DSUBST substitutes with a copy of NEW. More e cient than SUBST.

(LSUBST NEW OLD EXPR) [Function]
> Like SUBST except NEW is substituted as a segment of the list EXPR rather than as an element. For instance,

```
(LSUBST '(A B) 'Y '(X Y Z))  =>  (X A B Z)
```

> Note that if NEW is not a list, LSUBST returns a copy of EXPR with all OLD's deleted:

```
(LSUBST NIL 'Y '(X Y Z))  =>  (X Z)
```

(SUBLIS ALST EXPR FLG) [Function]
> ALST is a list of pairs:

```
((OLD 1 . NEW 1) (OLD 2 . NEW 2)   (OLD N . NEW N))
```

> Each OLD_i is an atom. SUBLIS returns the result of substituting each NEW_i for the corresponding OLD_i in EXPR, e.g.,

```
(SUBLIS '((A . X) (C . Y)) '(A B C D))  =>  (X B Y D)
```

> If FLG = NIL, new structure is created only if needed, so if there are no substitutions, the value is EQ to EXPR. If FLG = T, the value is always a copy of EXPR.

(DSUBLIS ALST EXPR FLG) [Function]
> Similar to SUBLIS, except it does not copy EXPR, but changes the list structure EXPR itself.

(SUBPAIR OLD NEW EXPR FLG) [Function]
> Similar to SUBLIS, except that elements of NEW are substituted for corresponding atoms of OLD in EXPR, e.g.,

```
(SUBPAIR '(A C) '(X Y) '(A B C D))  =>  (X B Y D)
```

> As with SUBLIS, new structure is created only if needed, or if FLG = T, e.g., if FLG = NIL and there are no substitutions, the value is EQ to EXPR.

> If OLD ends in an atom other than NIL, the rest of the elements on NEW are substituted for that atom. For example, if OLD = (A B . C) and NEW = (U V X Y Z), U is substituted for A, V for B, and (X Y Z) for C. Similarly, if OLD itself

is an atom (other than NIL), the entire list NEW is substituted for it. Examples:

```
(SUBPAIR '(A B . C) '(W X Y Z) '(C A B B Y))  =>  ((Y Z) W X
X Y)
```

Note that SUBST, DSUBST, and LSUBST all substitute copies of the appropriate expression, whereas
SUBLIS, and DSUBLIS, and SUBPAIR substitute the identical structure (unless FLG= T). For example:

```
_ (SETQ FOO '(A B))
(A B)
_ (SETQ BAR '(X Y Z))
(X Y Z)
_ (DSUBLIS (LIST (CONS 'X FOO)) BAR)
((A B) Y Z)
_ (DSUBLIS (LIST (CONS 'Y FOO)) BAR T)
((A B) (A B) Z)
_ (EQ (CAR BAR) FOO)
T
_ (EQ (CADR BAR) FOO)
NIL
```

2.5.9   Association Lists and Property Lists

(ASSOC KEY ALST)   [Function]

> ALST is a list of lists. ASSOC returns the rst sublist of ALST whose CAR is EQ to
> KEY . If such a list is not found, ASSOC returns NIL. Example:
>
> (ASSOC 'B '((A . 1) (B . 2) (C . 3))) => (B . 2)

(FASSOC KEY ALST)   [Function]

> Faster version of ASSOC that terminates on a null-check.
>
> (Interlisp-10) Interpreted, FASSOC gives an error if ALST ends in a non-list other
> than NIL, BAD ARGUMENT - FASSOC.

(SASSOC KEY ALST)   [Function]

> Same as ASSOC but uses EQUAL instead of EQ when searching for KEY .

(PUTASSOC KEY VAL ALST)   [Function]

> Searches ALST for a sublist CAR of which is EQ to KEY . If one is found, the CDR is
> replaced (using RPLACD) with VAL . If no such sublist is found, (CONS KEY VAL)
> is added at the end of ALST . Returns VAL . If ALST is not a list, generates an error,
> ARG NOT LIST.

Note that the argument order for ASSOC, PUTASSOC, etc. is di erent from that of LISTGET, LISTPUT,
etc.

(LISTGET LST PR OP)   [Function]

> Similar to GETPROP (page 2.7) but works on lists using property list format.
> Searches LST two elements at a time, by CDDR, looking for an element EQ to
> PR OP . If one is found, returns the next element of LST , otherwise NIL. Returns

NIL if LST is not a list. Example:

```
(LISTGET '(A 1 B 2 C 3) 'B)  =>  2

(LISTGET '(A 1 B 2 C 3) 'W)  =>  NIL
```

(LISTPUT LST PR OP VAL)   [Function]
> Similar to PUTPROP. Searches LST two elements at a time, by CDDR, looking for an element EQ to PR OP. If PR OP is found, replaces the next element of LST with VAL. Otherwise, PR OP and VAL are added to the end of LST. If LST is a list with an odd number of elements, or ends in a non-list other than NIL, PR OP and VAL are added at its beginning. Returns VAL. If LST is not a list, generates an error, ARG NOT LIST.

(LISTGET1 LST PR OP)   [Function]
> Like LISTGET, but searches LST one CDR at a time, i.e., looks at each element. Returns the next element after PR OP. Examples:

```
(LISTGET1 '(A 1 B 2 C 3) 'B)  =>  2

(LISTGET1 '(A 1 B 2 C 3) '1)  =>  B

(LISTGET1 '(A 1 B 2 C 3) 'W)  =>  NIL
```

> Note: LISTGET1 used to be called GET.

(LISTPUT1 LST PR OP VAL)   [Function]
> Like LISTPUT, except searches LST one CDR at a time. Returns the modi ed LST. Example:

```
_(SETQ FOO '(A 1 B 2))
(A 1 B 2)
_(LISTPUT FOO 'B 3)
(A 1 B 3)
_(LISTPUT FOO 'C 4)
(A 1 B 3 C 4)
_(LISTPUT FOO 1 'W)
(A 1 W 3 C 4)
_FOO
(A 1 W 3 C 4)
```

> Note that if LST is not a list, no error is generated. However, since a non-list cannot be changed into a list, LST is not modi ed. In this case, the value of LISTPUT1 should be saved. Example:

```
_(SETQ FOO NIL)
NIL
_(LISTPUT FOO 'A 5)
(A 5)
_FOO
NIL
```

2.5.10   Other List Functions

(REMOVE X L)   [Function]

> Removes all top-level occurrences of x from list L, returning a copy of L with all elements EQUAL to x removed. Example:
>
> (REMOVE 'A '(A B C (A) A)) => (B C (A))
>
> (REMOVE '(A) '(A B C (A) A)) => (A B C A)

(DREMOVE X L)   [Function]

> Similar to REMOVE, but uses EQ instead of EQUAL, and actually modi es the list L when removing x, and thus does not use any additional storage. More e cient than REMOVE.
>
> Note that DREMOVE cannot *change* a list to NIL:
>
> ```
> _(SETQ FOO '(A))
> (A)
> _(DREMOVE 'A FOO)
> NIL
> _FOO
> (A)
> ```
>
> The DREMOVE above returns NIL, and does not perform any CONSes, but the value of FOO is *still* (A), because there is no way to change a list to a non-list. See NCONC.

(REVERSE L)   [Function]

> Reverses (and copies) the top level of a list, e.g.,
>
> (REVERSE '(A B (C D))) => ((C D) B A)
>
> If L is not a list, REVERSE just returns L.

(DREVERSE L)   [Function]

> Value is the same as that of REVERSE, but DREVERSE destroys the original list L and thus does not use any additional storage. More e cient than REVERSE.

2.6   STRINGS

A string is an object which represents a sequence of characters. Interlisp provides functions for creating strings, concatenating strings, and creating sub-strings of a string.

The input syntax for a string is a double quote ("), followed by a sequence of any characters except double quote and %, terminated by a double quote. The % and double quote characters may be included in a string by preceding them with the escape character %.

Strings are printed by PRINT and PRIN2 with initial and nal double quotes, and %s inserted where

necessary for it to read back in properly. Strings are printed by PRIN1 without the delimiting double quotes and extra %s.

A ''null string'' containing no characters is input as "". The null string is printed by PRINT and PRIN2 as "". (PRIN1 "") doesn't print anything.

Strings are created by MKSTRING, ALLOCSTRING, SUBSTRING, and CONCAT.

Internally a string is stored in two parts; a ''string pointer'' and the sequence of characters. Several string pointers may reference the same character sequence, so a substring can be made by creating a new string pointer, without copying any characters. It is not possible to directly access a character sequence, so functions that refer to ''strings'' actually manipulate string pointers. In most cases, the user does not have to be aware of string pointers, but there are some situations where it is important to understand them. For example, suppose that x is a string pointer to a sequence of characters, and Y is another string pointer to a substring of x's characters. If the characters of Y are modied (with RPLSTRING or RPLCHARCODE), the corresponding characters of x will be modied too.

(STREQUAL x Y)   [Function]
> Returns T if x and Y are both strings and they contain the same sequence of characters, otherwise NIL. EQUAL uses STREQUAL. Note that strings may be STREQUAL without being EQ. For instance,
>
> (STREQUAL "ABC" "ABC") => T
>
> (EQ "ABC" "ABC") => NIL
>
> STREQUAL returns T if x and Y are the same string pointer, or two dierent string pointers which point to the same character sequence, or two string pointers which point to dierent character sequences which contain the same characters. Only in the rst case would x and Y be EQ.

(ALLOCSTRING N INITCHAR OLD)   [Function]
> Creates a string of length N charaters of INITCHAR (which can be either a character code or something coercible to a character). If INITCHAR is NIL, it defaults to character code 0. if OLD is supplied, it must be a string pointer, which is re-used.

(MKSTRING x FLG RDTBL)   [Function]
> If x is a string, returns x. Otherwise, creates and returns a string containing the print name of x. Examples:
>
> (MKSTRING "ABC") => "ABC"
>
> (MKSTRING '(A B C)) => "(A B C)"
>
> (MKSTRING NIL) => "NIL"
>
> Note that the last example returns the string "NIL", not the atom NIL.
>
> If FLG is T, then the PRIN2-name of x is used, computed with respect to the readtable RDTBL . For example,
>
> (MKSTRING "ABC" T) => "%"ABC%""

(SUBSTRING X N M OLDPTR) [Function]

Returns the substring of x consisting of the Nth through Mth characters of x. If M is NIL, the substring contains the Nth character thru the end of x. N and M can be negative numbers, which are interpreted as counts back from the end of the string, as with NTHCHAR (page 2.10). SUBSTRING returns NIL if the substring is not well de ned, e.g., N or M specify character positions outside of x, or N corresponds to a character in x to the right of the character indicated by M). Examples:

(SUBSTRING "ABCDEFG" 4 6) => "DEF"

(SUBSTRING "ABCDEFG" 3 3) => "C"

(SUBSTRING "ABCDEFG" 3 NIL) => "CDEFG"

(SUBSTRING "ABCDEFG" 4 -2) => "DEF"

(SUBSTRING "ABCDEFG" 6 4) => NIL

(SUBSTRING "ABCDEFG" 4 9) => NIL

If x is not a string, it is converted to one. For example,

(SUBSTRING '(A B C) 4 6) => "B C"

SUBSTRING does not actually copy any characters, but simply creates a new string pointer to the characters in x. If OLDPTR is a string pointer, it is modi ed and returned.

(GNC x) [Function]

''Get Next Character.'' Returns the next character of the string x (as an atom); also removes the character from the string, by changing the string pointer. Returns NIL if x is the null string. If x isn't a string, a string is made. Used for sequential access to characters of a string. Example:

```
_(SETQ FOO "ABCDEFG")
"ABCDEFG"
_(GNC FOO)
A
_(GNC FOO)
B
_FOO
"CDEFG"
```

Note that if A is a substring of B, (GNC A) does not remove the character from B. GNC doesn't physically change the string of characters, just the string pointer.

(GLC x) [Function]

''Get Last Character.'' Returns the last character of the string x (as an atom); also removes the character from the string. Similar to GNC. Example:

```
_(SETQ FOO "ABCDEFG")
"ABCDEFG"
_(GLC FOO)
```

Strings

```
G
_(GLC FOO)
F
_FOO
"ABCDE"
```

(CONCAT x₁ x₂ ... x_N)   [NoSpread Function]

Returns a new string which is the concatenation of (copies of) its arguments. Any arguments which are not strings are transformed to strings. Examples:

```
(CONCAT "ABC" 'DEF "GHI")  =>  "ABCDEFGHI"

(CONCAT '(A B C) "ABC")  =>  "(A B C)ABC"
```

(CONCAT) returns the null string, "".

(CONCATLIST x)   [Function]

x is a list of strings and/or other objects. The objects are transformed to strings if they aren't strings. Returns a new string which is the concatenation of the strings. Example:

```
(CONCATLIST '(A B (C D) "EF"))  =>  "AB(C D)EF"
```

(RPLSTRING X N Y)   [Function]

Replaces the characters of string X beginning at character position N with string Y. X and Y are converted to strings if they aren't already. N may be positive or negative, as with SUBSTRING. Characters are smashed into (converted) X. Returns the string X. Examples:

```
(RPLSTRING "ABCDEF" -3 "END")  =>  "ABCEND"

(RPLSTRING "ABCDEFGHIJK" 4 '(A B C))  =>  "ABC(A B C)K"
```

Generates an error if there is not enough room in X for Y, i.e., the new string would be longer than the original. If Y was not a string, X will already have been modied since RPLSTRING does not know whether Y will ''t'' without actually attempting the transfer.

Note that if X is a substring of Z, Z will also be modied by the action of RPLSTRING. Example:

```
_ (SETQ FOO "ABCDEFG")
"ABCDEFG"
_ (SETQ BAR (SUBSTRING FOO 4 6)
"DEF"
_ (RPLSTRING BAR 2 "XY")
"DXY"
_ FOO
"ABCDXYG"
```

(RPLCHARCODE X N CHAR CODE)   [Function]

Replaces the Nth character of the string X with the character code CHAR CODE . N may be positive or negative. Returns the new X. Similar to RPLSTRING. Example:

```
              (RPLCHARCODE "ABCDE" 3 (CHARCODE F))  =>  "ABFDE"
```

(STRPOS PAT STRING START SKIP ANCHOR   TAIL)   [Function]

        STRPOS is a function for searching one string looking for another. PAT and STRING are both strings (or else they are converted automatically). STRPOS searches STRING beginning at character number START, (or 1 if START is NIL) and looks for a sequence of characters equal to PAT. If a match is found, the character position of the rst matching character in STRING is returned, otherwise NIL. Examples:

```
(STRPOS "ABC" "XYZABCDEF")  =>  4

(STRPOS "ABC" "XYZABCDEF" 5)  =>  NIL

(STRPOS "ABC" "XYZABCDEFABC" 5)  =>  10
```

        SKIP can be used to specify a character in PAT that matches any character in STRING. Examples:

```
(STRPOS "A&C&" "XYZABCDEF" NIL '&)  =>  4

(STRPOS "DEF&" "XYZABCDEF" NIL '&)  =>  NIL
```

        If ANCHOR is T, STRPOS compares PAT with the characters beginning at position START (or 1 if START is NIL). If that comparison fails, STRPOS returns NIL without searching any further down STRING. Thus it can be used to compare one string with some *portion* of another string. Examples:

```
(STRPOS "ABC" "XYZABCDEF" NIL NIL T)  =>  NIL

(STRPOS "ABC" "XYZABCDEF" 4 NIL T)  =>  4
```

        Finally, if TAIL is T, the value returned by STRPOS if successful is not the starting position of the sequence of characters corresponding to PAT, but the position of the rst character after that, i.e., the starting position plus (NCHARS PAT). Examples:

```
(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T)  =>  7

(STRPOS "A" "A" NIL NIL NIL T)  =>  2
```

        If TAIL= NIL, STRPOS returns NIL, or a character position within STRING which can be passed to SUBSTRING. In particular, (STRPOS "" "") => NIL. However, if TAIL= T, STRPOS may return a character position outside of STRING. For instance, note that the second example above returns 2, even though "A" has only one character.

(STRPOSL A STR START NEG)   [Function]

        STR is a string (or else it is converted automatically to a string), A is a list of characters or character codes. STRPOSL searches STR beginning at character number START (or else 1 if START= NIL) for one of the characters in A. If one is found, STRPOSL returns as its value the corresponding character position, otherwise NIL. Example:

```
(STRPOSL '(A B C) "XYZBCD")  =>  4
```

If NEG = T, STRPOSL searches for a character *not* on A. Example:

```
(STRPOSL '(A B C) "ABCDEF" NIL T)  =>  4
```

If any element of A is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via CHCON1. Therefore, it is more e cient to call STRPOSL with A a list of character *codes.*

If A is a bit table, it is used to specify the characters (see MAKEBITTABLE below)

STRPOSL uses a ''bit table'' data structure to search e ciently. If A is not a bit table, it is converted it to a bit table using MAKEBITTABLE. If STRPOSL is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to STRPOSL as its rst argument.

(MAKEBITTABLE L NEG A) [Function]
> Returns a bit table suitable for use by STRPOSL. L is a list of characters or character codes, NEG is the same as described for STRPOSL. If A is a bit table, MAKEBITTABLE modi es and returns it. Otherwise, it will create a new bit table.

Note: if NEG = T, STRPOSL must call MAKEBITTABLE whether A is a list *or* a bit table. To obtain bit table e ciency with NEG = T, MAKEBITTABLE should be called with NEG = T, and the resulting ''inverted'' bit table should be given to STRPOSL with NEG = NIL.

2.7 ARRAYS

An array in Interlisp is an object representing a one-dimensional vector of objects. Arrays do not have input syntax; they can only be created by the function ARRAY. Arrays are printed by PRINT, PRIN2, and PRIN1 as # followed by an integer.

Note: Interlisp-10 and Interlisp-Vax provide a much more primitive version of arrays than other implementations of Interlisp. See page 2.33.

(ARRAY SIZE TYPE INIT ORIG) [Function]
> Creates and returns a new array capable of containing SIZE objects of type TYPE. TYPE may be one of BIT, BYTE, WORD, FIXP, FLOATP, POINTER, or DOUBLEPOINTER.[4] ARRAY also accepts any ''type'' which is legal in DATATYPE records (such as (BITS 7), FLAG, see page 3.7). (Note: DATATYPE types are coerced into the next ''enclosing'' array type. Therefore, users should not rely on truncation of values stored in arrays of these types.)

---

[4]For backward compatibility with Interlisp-10 arrays, TYPE can be NIL or 0 (meaning to create an array of type DOUBLEPOINTER) or SIZE (meaning an array of type FIXP). For arrays of type DOUBLEPOINTER, the functions ELTD and SETD are de ned the same as in Interlisp-10 (page 2.34). For arrays of any other type, ELTD and SETD are the same as ELT and SETD. Combined POINTER/FIXP arrays are not supported. Interlisp-D users should avoid using Interlisp-10 arrays.

INIT is the initial value in each element of the new array. If not speciﬁed, the array elements will be initialized with 0 (for number arrays) or NIL (all other types).

Arrays can have either 0-origin or 1-origin indexing, as speciﬁed by the ORIG argument; if ORIG is not speciﬁed, the default is 1.

(ELT A N) [Function]

Returns the Nth element of the array A.

(SETA A N V) [Function]

Sets the Nth element of the array A to V. SETA returns V.

(ARRAYTYP A) [Function]

Returns a value corresponding to the second argument to ARRAY.

Note: If ARRAY coerced the array type as described above, ARRAYTYP will return the *new* type.

(ARRAYSIZE A) [Function]

Returns the size of array A. Generates the error, ARG NOT ARRAY, if A is not an array.

(ARRAYORIG A) [Function]

Returns the origin of array A, which may be 0 or 1. Generates an error, ARG NOT ARRAY, if A is not an array.

(COPYARRAY A) [Function]

Returns a new array of the same size and type as A, and with the same contents as A. Generates an ARG NOT ARRAY error, if A is not an array.

2.7.1 Interlisp-10 Arrays

Interlisp-10 and Interlisp-Vax have a more primitive array facility than the other implementations of Interlisp. In Interlisp-10, arrays are partitioned into four sections: a header, a section containing unboxed numbers, a section containing list cells (each with a CAR and CDR), and a section containing relocation information. The last three sections can each be of arbitrary length (including 0); the header is two words long and contains the length of the other sections. The unboxed number region of an array is used to store 36 bit quantities that are not Interlisp pointers, and therefore are not to be chased during garbage collections, e.g. machine instructions. The relocation informaion is used when the array contains the deﬁnition of a compiled function, and speciﬁes which locations in the *unboxed* region of the array must be changed if the array is moved during a garbage collection.

ARRAY returns an ''array pointer'' to the beginning of the array, but it is also possible to create a pointer into the middle of an array. ARRAYP will accept a pointer into the middle of an array, but ELT, SETA, ELTD, and SETD generate an error, ARG NOT ARRAY, if A is not an array pointer to the beginning of an array.

Array-pointers print as #NNNN , where NNNN is the octal representation of the pointer. Note that #NNNN will be read as a literal atom, and not an array pointer.

The following functions are used to manipulate Interlisp-10 arrays:

(ARRAY N P V)   [Function]
> Allocates a block of N+2 words, of which the rst two are header information. The next P (N) words contain unboxed numbers, and are initialized to unboxed 0. The last N-P (0) words are list cells; both CAR and CDR are available for storing information, and each is initialized to V. If P is NIL, 0 is used (i.e., an array containing all Interlisp pointers). ARRAY returns an ''array pointer'' to the array.
>
> If su cient space is not available for the array, a garbage collection of array space is initiated. If this is unsuccessful in obtaining su cient space, an error is generated, ARRAYS FULL.

(ELT A N)   [Function]
> Returns the Nth element of the array A. (ELT A 1) is the rst element of the array (actually corresponds to the 3rd cell because of the 2 word header).
>
> If N corresponds to the unboxed number region of A, ELT returns the full 36 bit word as a boxed integer. If N corresponds to the list cell region of A, ELT returns the CAR of the corresponding element.

(SETA A N V)   [Function]
> Sets the Nth element of the array A to V. If N corresponds to the unboxed number region of A, V must be a number, and is unboxed and stored as a full 36 bit word into the Nth element of A. If N corresponds to the list cell region of A, V replaces the CAR of the Nth element. SETA returns V.

(ELTD A N)   [Function]
> Same as ELT for the unboxed number region of A, but returns the CDR of the Nth element, if N corresponds to the list cell region of A.

(SETD A N V)   [Function]
> Same as SETA for the unboxed number region of A, but sets the CDR half of the Nth element, if N corresponds to the list cell region of A. SETD returns V.

(ARRAYTYP A)   [Function]
> Returns the number of unboxed number words of array A. This value corresponds to the second argument to ARRAY.

(ARRAYP X)   [Function]
> Returns X if X is an array pointer, otherwise NIL. No check is made to ensure that X actually addresses the *beginning* of an array.

(ARRAYBEG A)   [Function]
> If A is a pointer into the middle of an array, returns the pointer to its beginning. Otherwise returns NIL.

(ARRAYORIG A)   [Function]
> Returns 1. A dummy function provided for compatibility with other Interlisp arrays.

DATA TYPES

2.8   HASH ARRAYS

Hash arrays provide a mechanism for associating arbitrary lisp objects (''hash keys'') with other objects (''hash values''), such that the hash value associated with a particular hash key can be quickly obtained. A set of associations could be represented as a list or array of pairs, but these schemes are very ine cient when the number of associations is large. There are functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key.

Hash keys can be any lisp object, but is should be noted that the hash array functions use EQ for comparing hash keys. Therefore, if non-atoms are used as hash keys, the exact same object (not a copy) must be used to retrieve the hash value.

In the description of the functions below, the argument HARRA Y has one of three forms: NIL, in which case a hash array provided by the system, SYSHASHARRAY, is used; a hash-array created by the function HARRAY; or a list, CAR of which is a hash array. The latter form is used for specifying what is to be done on over ow, as described below.

(HARRAY LEN)   [Function]
          Creates a hash array containing at least LEN hash keys.

(HARRAYSIZE HARRA Y)   [Function]
          Returns the size of HARRA Y; the number of hash keys it can hold before becoming ''full''.

(CLRHASH HARRA Y)   [Function]
          Clears all hash keys/values from HARRA Y. Returns HARRA Y.

(PUTHASH KEY VAL HARRA Y)   [Function]
          Associates the hash value VAL with the hash key KEY in HARRA Y. Replaces the previous hash value, if any. If VAL is NIL, any old association is removed (hence a hash value of NIL is not allowed). If HARRA Y is full when PUTHASH is called with a key not already in the hash array, the function HASHOVERFLOW is called, and the PUTHASH is done to the value returned (see below). Returns VAL .

(GETHASH KEY HARRA Y)   [Function]
          Returns the hash value associated with the hash key KEY in HARRA Y. Returns NIL, if KEY is not found.

(REHASH OLDHARRA Y NEWHARRA Y)   [Function]
          Hashes all hash keys and values in OLDHARRA Y into NEWHARRA Y. The two hash arrays do not have to be (and usually aren't) the same size. Returns NEWHARRA Y.

(MAPHASH HARRA Y MAPHFN)   [Function]
          MAPHFN is a function of two arguments. For each hash key in HARRA Y, MAPHFN will be applied to (1) the hash value, and (2) the hash key. For example,

```
[MAPHASH A
    (FUNCTION (LAMBDA (VAL KEY)
                        (if (LISTP KEY) then (PRINT VAL)]
```

          will print the hash value for all hash keys that are lists. MAPHASH returns HARRA Y.

(DMPHASH HARRA Y_1 HARRA Y_2 HARRA Y_N)   [NLambda NoSpread Function]

         Prints on the primary output ﬁle LOADable forms which will restore the hash-arrays contained as the values of the atoms HARRA Y_1, HARRA Y_2,   HARRA Y_N. Example: (DMPHASH SYSHASHARRAY) will dump the system hash-array.

         Note: all EQ identities except atoms and small integers are lost by dumping and loading because READ will create new structure for each item. Thus if two lists contain an EQ substructure, when they are dumped and loaded back in, the corresponding substructures while EQUAL are no longer EQ. The HORRIBLEVARS ﬁle package command (page 11.25) provides a way of dumping hash tables such that these identities are preserved.

2.8.1   Hash Overﬂow

When a hash array becomes full, attempting to add another hash key will cause the function HASHOVERFLOW to be called. This will either automatically enlarge the hash array, or cause the error HASH TABLE FULL. How hash overﬂow is handled is determined by the form that was passed to PUTHASH:

HARRA Y   If a plain hash array is passed to a hash function, and it overﬂows, the error HASH ARRAY FULL is generated.

NIL   If a hash function is passed NIL as its HARRA Y argument, the system hash array SYSHASHARRAY is used. This array is not used by the system, but is provided for the user. If SYSHASHARRAY overﬂows, it is automatically enlarged by 1.5.

(HARRA Y . N)   N is a positive integer. This form speciﬁes that upon hash overﬂow, a new hash-array is created with N more cells than the current hash-array.

(HARRA Y . F)   F is a ﬂoating point number. This form speciﬁes that upon hash overﬂow, the new hash array will be F times the size of the current hash-array.

(HARRA Y . FN)   FN is a function name or a lambda expression. This form speciﬁes that upon hash overﬂow, FN is called with (HARRA Y . FN) as its argument. If FN returns a number, the number will be the size of the new hash array. Otherwise, the new size defaults to 1.5 times the size of the old hash array. FN could be used to print a message, or perform some monitor function.

(HARRA Y)   Equivalent to (HARRA Y . 1.5).

If a list form is used, upon hash overﬂow the new hash-array is RPLACAed into the dotted pair, and HASHOVERFLOW returns it.

2.9   NUMBERS AND ARITHMETIC FUNCTIONS

Numerical atoms, or simply numbers, do not have value cells, function deﬁnition cells, property lists, or explicit print names. There are three diﬀerent types of numbers in Interlisp: small integers, large integers, and ﬂoating point numbers. Small integers are those integers that can be directly stored within a

pointer value. The range of small integers is implementation- dependent. Since a large integer or oating point number can be (in value) any full word quantity (and vice versa), it is necessary to distinguish between those full word quantities that represent large integers or oating point numbers, and other Interlisp pointers. We do this by ''boxing'' the number: When a large integer or oating point number is created (via an arithmetic operation or by READ), Interlisp gets a new word from ''number storage'' and puts the large integer or oating point number into that word. Interlisp then passes around the pointer to that word, i.e., the ''boxed number'', rather than the actual quantity itself. Then when a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the ''value'' of the number. This latter process is called ''unboxing''. Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, if a computation creates many large integers or oating point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of oating point number space. Di erent implementations of Interlisp may use di erent boxing strategies. Thus, while lots of arithmetic operations *may* lead to garbage collections, this is not necessarily always the case.

The following functions can be used to distinguish the di erent types of numbers:

(SMALLP x)   [Function]
> Returns x, if x is a small integer; NIL otherwise. Does *not* generate an error if x is not a number.

(FIXP x)   [Function]
> Returns x, if x is an integer (between MIN.FIXP and MAX.FIXP); NIL otherwise. Note that FIXP is true for both large and small integers. Does *not* generate an error if x is not a number.

(FLOATP x)   [Function]
> Returns x if x is a oating point number; NIL otherwise. Does *not* give an error if x is not a number.

(NUMBERP x)   [Function]
> Returns x, if x is a number of any type (FIXP or FLOATP); NIL otherwise. Does *not* generate an error if x is not a number.
>
> Note that if (NUMBERP x) is true, then either (FIXP x) or (FLOATP x) is true.

Each small integer has a unique representation, so EQ may be used to check equality. Note that EQ should not be used for large integers or oating point numbers, EQP, IEQP, or EQUAL must be used instead.

(EQP x y)   [Function]
> Returns T, if x and y are EQ, or equal numbers; NIL otherwise. Note that EQ may be used if x and y are known to be *small* integers. EQP does not convert x and y to integers, e.g., (EQP 2000 2000.3) => NIL, but it can be used to compare an integer and a oating point number, e.g., (EQP 2000 2000.0) => T. EQP does *not* generate an error if x or y are not numbers.
>
> Note: EQP can also be used to compare stack pointers (page 7.3) and compiled code objects (page 5.8).

Integer Arithmetic

2.9.1   Integer Arithmetic

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of digits, followed by an optional Q, and terminated by a delimiting character. If the Q is present, the digits are interpreted in octal, otherwise in decimal, e.g. 77Q and 63 both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of how integers were created.

The setting of RADIX (page 6.19), determines how integers are printed: signed or unsigned, octal or decimal.

Integers are created by PACK and MKATOM when given a sequence of characters observing the above syntax, e.g. (PACK '(1 2 Q)) => 10. Integers are also created as a result of arithmetic operations.

The range of integers of various types is implementation-dependent. This information is accessable to the user through the following variables:

MIN.SMALLP   [Variable]
MAX.SMALLP   [Variable]
    The smallest/largest possible small integer.

MIN.FIXP   [Variable]
MAX.FIXP   [Variable]
    The smallest/largest possible large integer.

MIN.INTEGER   [Variable]
MAX.INTEGER   [Variable]
    The smallest/largest possible integer representable. Currently, these variables are equal to MIN.FIXP and MAX.FIXP; they may be di erent in future implementations with other methods for representing integers.

In Interlisp-D, the action taken on integer over ow is determined with the following function:

(OVERFLOW FLG)   [Function]
    Sets a ag that determines the system response to integer over ow; returns the previous setting. If FLG= T, an error occurs on integer over ow. If FLG= NIL, the largest (or smallest) integer is returned as the result of the over owed computation. If FLG= 0, the result is returned modulo 2^{32} (the default action).

All of the functions described below work on integers. Unless speci ed otherwise, if given a oating point number, they rst convert the number to an integer by truncating the fractional bits, e.g., (IPLUS 2.3 3.8)= 5; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

(IPLUS x_1 x_2 x_N)   [NoSpread Function]
    Returns the sum $x_1 + x_2 + \dots + x_N$. (IPLUS)= 0.

(IMINUS x)   [Function]
    -x

(IDIFFERENCE x Y)   [Function]
    X - Y

DATA TYPES

(ADD1 x)   [Function]

        $x + 1$

(SUB1 x)   [Function]

        $x - 1$

(ITIMES x_1 x_2 x_N)   [NoSpread Function]

        Returns the product $x_1 * x_2 *$ $* x_N$. (ITIMES)= 1.

(IQUOTIENT x y)   [Function]

        x / y truncated. Examples:

        (IQUOTIENT 3 2) => 1

        (IQUOTIENT -3 2) => -1

(IREMAINDER x y)   [Function]

        Returns the remainder when x is divided by y. Example:

        (IREMAINDER 3 2) => 1

(IMOD x y)   [Function]

        Computes the integer modulus; this di ers from IREMAINDER in that the result
        is always a non-negative integer in the range [0,y).

(IGREATERP x y)   [Function]

        T, if $x > y$; NIL otherwise.

(ILESSP x y)   [Function]

        T, if $x < y$; NIL otherwise.

(IGEQ x y)   [Function]

        T, if x y; NIL otherwise.

(ILEQ x y)   [Function]

        T, if x y; NIL otherwise.

(IMIN x_1 x_2 x_N)   [NoSpread Function]

        Returns the minimum of x_1, x_2, , x_N. (IMIN) returns the largest possible large
        integer, the value of MAX.FIXP.

(IMAX x_1 x_2 ...x_N)   [NoSpread Function]

        Returns the maximum of x_1, x_2, , x_N. (IMAX) returns the smallest possible
        large integer, the value of MIN.FIXP.

(IEQP N M)   [Function]

        Returns T if N and M are EQ or equal integers; NIL otherwise. Note that EQ
        may be used if N and M are known to be *small* integers. IEQP converts N and M
        to integers, e.g., (IEQP 2000 2000.3) => T. Causes NON-NUMERIC ARG
        error if either N or M are not numbers.

(ZEROP x)   [Function]

        (EQ x 0).

Note: ZEROP should not be used for oating point numbers because it uses EQ. Use (EQP x 0) instead.

(MINUSP x) [Function]

Returns T if x is negative; NIL otherwise. Does not convert x to an integer, but simply checks the sign bit.

(FIX x) [Function]

If x is an integer, returns x. Otherwise, converts x to an integer by truncating fractional bits, e.g., (FIX 2.3) => 2, (FIX -1.7) => -1.

Since FIX is also a programmer's assistant command (page 8.10), typing FIX directly to Interlisp will not cause the function FIX to be called.

(GCD x Y) [Function]

Returns the greatest common divisor of x and Y, e.g., (GCD 72 64)= 8.

2.9.2   Logical Arithmetic Functions

(LOGAND x_1 x_2 x_N) [NoSpread Function]

Returns the logical AND of all its arguments, as an integer. Example:

(LOGAND 7 5 6) => 4

(LOGOR x_1 x_2 x_N) [NoSpread Function]

Returns the logical OR of all its arguments, as an integer. Example:

(LOGOR 1 3 9) => 11

(LOGXOR x_1 x_2 x_N) [NoSpread Function]

Returns the logical exclusive OR of its arguments, as an integer. Example:

(LOGXOR 11 5) => 14

(LOGXOR 11 5 9) <=> (LOGXOR 14 9) => 7

(LSH x N) [Function]

(arithmetic) ''Left Shift.'' Returns x shifted left N places, with the sign bit una ected. x can be positive or negative. If N is negative, x is shifted *right* -N places.

(RSH x N) [Function]

(arithmetic) ''Right Shift.'' Returns x shifted right N places, with the sign bit una ected, and copies of the sign bit shifted into the leftmost bit. x can be positive or negative. If N is negative, x is shifted *left* -N places.

Warning: Be careful if using RSH to simulate division; RSHing a negative number is not generally equivalent to deviding by a power of two.

(LLSH x N) [Function]

''Logical Left Shift.''

(LRSH X N)   [Function]

        ''Logical Right Shift.''

(INTEGERLENGTH N)   [Function]

        Returns the number of bits needed to represent N (coerced to a FIXP). This is equivalent to: $1+ \text{oor}[\log2[\text{abs}[N]]]$. (INTEGERLENGTH 0) = 0.

(POWEROFTWOP N)   [Function]

        Returns non-NIL if N (coerced to a FIXP) is a power of two.

(EVENP X Y)   [NoSpread Function]

        If Y is not given, equivalent to (ZEROP (IMOD X 2)); otherwise equivalent to (ZEROP (IMOD X Y)).

(ODDP X Y)   [NoSpread Function]

        Equivalent to (NOT (EVENP X Y)).

The difference between a logical and arithmetic right shift lies in the treatment of the sign bit. Logical shifting treats it just like any other bit; arithmetic shifting will not change it, and will ''propagate'' rightward when actually shifting rightwards. Note that shifting (arithmetic) a negative number ''all the way'' to the right yields -1, not 0.

The following ''logical'' arithmetic functions are derived from Common Lisp, and have both macro and function definitions (the macros are for speed in running of compiled code). The following code equivalences are primarily for definitional purposes, and should not be considered an implementation (especially since the real implementation tends to be faster and less ''consy'' than would be apparent from the code here).

Note: The following logical functions are currently only implemented in Interlisp-D.

(LOGNOT N)   [Function]

        (LOGXOR N -1)

(BITTEST N MASK)   [Function]

        (NOT (ZEROP (LOGAND N MASK)))

(BITCLEAR N MASK)   [Function]

        (LOGAND N (LOGNOT MASK))

(BITSET N MASK)   [Function]

        (LOGOR N MASK)

(MASK.1'S POSITION SIZE)   [Function]

        (LLSH (SUB1 (EXPT 2 SIZE))
            POSITION)

(MASK.0'S POSITION SIZE)   [Function]

        (LOGNOT (MASK.1'S POSITION SIZE))

(LOADBYTE N POSITION SIZE)   [Function]

        (LOGAND (LRSH N POSITION)

```
                    (MASK.1'S 0 SIZE))
```

(DEPOSITBYTE N POSITION SIZE BYTE)   [Function]

```
            (LOGOR (BITCLEAR N (MASK.1'S POSITION SIZE))
                   (LLSH (LOGAND BYTE (MASK.1'S 0 SIZE))
                         POSITION ))
```

(ROT X N FIELDSIZE)   [Function]
> ''Rotate bits in eld''. This is a slight extension of the CommonLisp ROT function. It performs a bitwise left-rotation of the integer X, by N places, within a eld of FIELDSIZE bits wide. Bits being shifted out of the position selected by (EXPT 2 (SUB1 FIELDSIZE)) will ow into the ''units'' position.
>
> The optional argument FIELDSIZE defaults to the ''cell'' size (the integerlength of the current maximum FIXP), and must either be a positive integer, or else be one of the litatoms CELL or WORD. In the latter two cases the appropriate numerical values are respectively substituted. A macro optimizes the case where FIELDSIZE is WORD and N is 1.

The notions of position and size can be combined to make up a ''byte speci er'', which is constructed by the macro BYTE [note reversal of arguments as compare with above functions]:

(BYTE SIZE POSITION)   [Macro]
> Constructs and returns a ''byte speci er'' containing SIZE and POSITION .

(BYTESIZE BYTESPEC)   [Macro]
> Returns the SIZE componant of the ''byte speci er'' BYTESPEC .

(BYTEPOSITION BYTESPEC)   [Macro]
> Returns the POSITION componant of the ''byte speci er'' BYTESPEC .

(LDB BYTESPEC VAL)   [Macro]

```
            (LOADBYTE VAL
                      (BYTEPOSITION BYTESPEC )
                      (BYTESIZE BYTESPEC ))
```

(DPB N BYTESPEC VAL)   [Macro]

```
            (DEPOSITBYTE VAL
                         (BYTEPOSITION BYTESPEC )
                         (BYTESIZE BYTESPEC )
                         N )
```

2.9.3   Floating Point Arithmetic

A oating point number is input as a signed integer, followed by a decimal point, followed by another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer) and terminated by a delimiter.

Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a oating point number from an integer. For example, the following will be recognized as oating point numbers:

```
5.       5.00    5.01    .3
5E2      5.1E2   5E-3    -5.2E+6
```

Floating point numbers are printed using the format control speci ed by the function FLTFMT (page 6.20). FLTFMT is initialized to T, or free format. For example, the above oating point numbers would be printed free format as:

```
5.0      5.0     5.01    .3
500.0    510.0   .005    -5.2E6
```

Floating point numbers are created by the read program when a "." or an E appears in a number, e.g., 1000 is an integer, 1000. a oating point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms. Floating point numbers are also created by PACK and MKATOM, and as a result of arithmetic operations.

PRINTNUM (page 6.21) permits greater controls on the printed appearance of oating point numbers, allowing such things as left-justi cation, suppression of trailing decimals, etc.

The oating point number range is stored in the following variables:

MIN.FLOAT   [Variable]
        The smallest possible oating point number.

MAX.FLOAT   [Variable]
        The largest possible oating point number.

All of the functions described below work on oating point numbers. Unless speci ed otherwise, if given an integer, they rst convert the number to a oating point number, e.g., (FPLUS 1 2.3) <=> (FPLUS 1.0 2.3) => 3.3; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

(FPLUS x_1 x_2 x_N)   [NoSpread Function]
        $x_1 + x_2 +$ $+ x_N$

(FMINUS x)   [Function]
        $- x$

(FDIFFERENCE x y)   [Function]
        $x - y$

(FTIMES x_1 x_2 x_N)   [NoSpread Function]
        $x_1 * x_2 *$ $* x_N$

(FQUOTIENT x y)   [Function]
        x / y

(FREMAINDER x y)   [Function]
        Returns the remainder when x is divided by y. Equivalent to:

```
                    (FDIFFERENCE x (FTIMES y (FIX (FQUOTIENT x y)))))
```

Example:

```
(FREMAINDER 7.5 2.3)  =>  0.6
```

(MINUSP x)   [Function]

        T, if x is negative; NIL otherwise. Works for both integers and oating point
numbers.

(FGREATERP x y)   [Function]

        T, if $x > y$, NIL otherwise.

(FLESSP x y)   [Function]

        T, if $x < y$, NIL otherwise.

(FEQP x y)   [Function]

        Returns T if N and M are equal oating point numbers; NIL otherwise. FEQP
converts N and M to oating point numbers.Causes NON-NUMERIC ARG error if
either N or M are not numbers.

(FMIN x_1 x_2 x_N)   [NoSpread Function]

        Returns the minimum of x_1, x_2, , x_N. (FMIN) returns the largest possible
oating point number, the value of MAX.FLOAT.

(FMAX x_1 x_2 ...x_N)   [NoSpread Function]

        Returns the maximum of x_1, x_2, , x_N. (FMAX) returns the smallest possible
oating point number, the value of MIN.FLOAT.

(FLOAT x)   [Function]

        Converts x to a oating point number. Example:

```
(FLOAT 0)  =>  0.0
```

2.9.4   Mixed Arithmetic

The functions in this section are ''generic'' oating point arithmetic functions. If any of the arguments
are oating point numbers, they act exactly like oating point functions, and oat all arguments, and
return a oating point number as their value. Otherwise, they act like the integer functions. If given a
non-numeric argument, they generate an error, NON-NUMERIC ARG.

(PLUS x_1 x_2 x_N)   [NoSpread Function]

        x_1 + x_2 + + x_N.

(MINUS x)   [Function]

        - x

(DIFFERENCE x y)   [Function]

        x - y

(TIMES x_1 x_2 x_N)   [NoSpread Function]

x_1 * x_2 *   * x_N

(QUOTIENT x y)   [Function]

If x and y are both integers, returns (IQUOTIENT x y), otherwise (FQUOTIENT x y).

(REMAINDER x y)   [Function]

If x and y are both integers, returns (IREMAINDER x y), other wise (FREMAINDER x y).

(GREATERP x y)   [Function]

T, if x > y, NIL otherwise.

(LESSP x y)   [Function]

T if x < y, NIL otherwise.

(GEQ x y)   [Function]

T, if x   y, NIL otherwise.

(LEQ x y)   [Function]

T, if x   y, NIL otherwise.

(MIN x_1 x_2 x_N)   [NoSpread Function]

Returns the minimum of x_1, x_2,   , x_N. (MIN) returns the value of MAX.INTEGER.

(MAX x_1 x_2 x_N)   [NoSpread Function]

Returns the maximum of x_1, x_2,   , x_N. (MAX) returns the value of MIN.INTEGER.

(ABS x)   [Function]

x if x > 0, otherwise -x. ABS uses GREATERP and MINUS, (not IGREATERP and IMINUS).

2.9.5   Special Functions

(EXPT m n)   [Function]

Returns m^n. If m is an integer and n is a positive integer, returns an integer, e.g, (EXPT 3 4) => 81, otherwise returns a oating point number. If m is negative and n fractional, an error is generated, ILLEGAL EXPONENTIATION. If n is oating and either too large or too small, an error is generated, VALUE OUT OF RANGE EXPT.

(SQRT n)   [Function]

Returns the square root of n as a oating point number. n may be xed or oating point. Generates an error if n is negative.

(LOG x)   [Function]

Returns the natural logarithm of x as a oating point number. x can be integer or oating point.

Special Functions

(ANTILOG x)   [Function]
> Returns the oating point number whose logarithm is x. x can be integer or oating point. Example:
>
>   (ANTILOG 1) = e => 2.71828...

(SIN x RADIANSFL G)   [Function]
> Returns the sine of x as a oating point number. x is in degrees unless RADIANSFL G = T.

(COS x RADIANSFL G)   [Function]
> Similar to SIN.

(TAN x RADIANSFL G)   [Function]
> Similar to SIN.

(ARCSIN x RADIANSFL G)   [Function]
> x is a number between -1 and 1 (or an error is generated). The value of ARCSIN is a oating point number, and is in degrees unless RADIANSFL G = T. In other words, if (ARCSIN x RADIANSFL G) = z then (SIN z RADIANSFL G) = x. The range of the value of ARCSIN is -90 to +90 for degrees, - /2 to /2 for radians.

(ARCCOS x RADIANSFL G)   [Function]
> Similar to ARCSIN. Range is 0 to 180, 0 to .

(ARCTAN x RADIANSFL G)   [Function]
> Similar to ARCSIN. Range is 0 to 180, 0 to .

(ARCTAN2 Y X RADIANSFL G)   [Function]
> Computes (ARCTAN (FQUOTIENT Y X) RADIANSFL G), and returns a corresponding value in the range -180 to 180 (or - to), i.e. the result is in the proper quadrant as determined by the signs of x and Y.

(RAND LOWER UPPER)   [Function]
> Returns a pseudo-random number between LOWER and UPPER inclusive, i.e., RAND can be used to generate a sequence of random numbers. If both limits are integers, the value of RAND is an integer, otherwise it is a oating point number. The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET described below.

(RANDSET x)   [Function]
> Returns the internal state of RAND. If x = NIL, just returns the current state. If x = T, RAND is initialized using the clocks, and RANDSET returns the new state. Otherwise, x is interpreted as a previous internal state, i.e., a value of RANDSET, and is used to reset RAND. For example,
>
>   _ (SETQ OLDSTATE (RANDSET))
>   ...
>   _ (for X from 1 to 10 do (PRIN1 (RAND 1 10)))
>   2847592748NIL
>   _ (RANDSET OLDSTATE)

```
...
_ (for X from 1 to 10 do (PRIN1 (RAND 1 10)))
2847592748NIL
```

Special Functions