**Interlisp to Common Lisp Concordia**

**Chapter 2 IRM (Datatypes)**

| Interlisp Form | Common Lisp Form |
|---|---|
| (DATATYPES --) | ?? |
| (TYPENAME datum) | ??<br>(type-of datum) -- except for strings and arrays Note that the result types are different, however, and it is necessary to check for literals in the program, e.g., (IL:TYPENAME 123) => IL:SMALLP yet (IL:TYPE-OF 123) => LISP:FIXNUM. Also LISP:TYPE-OF is definitely non-portable except for structures. |
| (TYPENAMEP datum typename) | ??<br>(typep datum typename) -- except for strings and arrays and the problem of non-portability of the type names. |

**2.1 Datatype Predicates**

**For many of these, the translation should look at the value/effect context.**
**If used for effect only, no need to insert the (and (<test> x) x).**

| | |
|---|---|
| (LITATOM x) | (symbolp x) |
| (SMALLP x) | (and (typep x 'fixnum) x) |
| (FIXP x) | (and (integerp x) x) |
| (FLOATP x) | (and (floatp x) x) |
| (NUMBERP x) | (and (numberp x) x) -- but includes more sorts of numbers |
| (ATOM x) | (and (or (symbolp x) (numberp x)) x)<br>Often users wrote IL:ATOM when they meant the LISP:ATOM interpretation, however. |
| (LISTP x) | (and (consp x) x) |
| (NLISTP x) | (not (consp x)) or (atom x) |
| (STRINGP x) | (and (stringp x) x) |
| (ARRAYP x) | ?? How are arrays to be represented?<br>possibly (and (vectorp x) x)<br>BVM - "ARRAYP probably translates as vectorp. Again, the real question is how ARRAY translates, at least when the origin is 1 (the default). You could translate to make-array with a size one greater than specified (wasting the zero element), but then you can't translate ARRAYSIZE as length. Sigh." |
| (HARRYP x) | (and (hash-table-p x) x)<br>-- Not quite strong enough since Interlisp hash tables are more general than CL ones<br>BVM -- "hash-table-p is probably good enough; it's the translation of HASH-ARRAY that will need more strength." |

**2.2 Datatype Equality**

| | |
|---|---|
| (EQ x y) | (eq x y) |
| (NEQ x y) | (not (eq x y)) |
| (NULL x) | (null x) |
| (NOT x) | (not x) |
| (EQP x y) | (or (eq x y)<br>(and (numberp x) (number y)<br>(= x y)))<br>Probably (= x y) will suffice in most cases<br>BVM - "EQP also compares compiled code, but there's not much hope there." |
| (EQUAL x y) | ??<br>Probably (equal x y) will suffice in most cases (differ on number comparisons and the CL version descends more datatypes) |
| (EQUALALL x y) | ??<br>Probably (equalp x y) will suffice in most cases (differ on string comparisons) |

**2.3 Fast and Destructive Functions**

**2.4.1 Using Litatoms as Variables**

| | |
|---|---|
| (BOUNDP var) | (boundp var) |
| (SET x y) | (set x y)<br>Note that this is a place where free variable references might "sneak" in and ruin the automatic "only declare special things that are used free." algorithm. |
| (SETQ x y) | (setq x y) |
| (SETQQ x y) | (setq x 'y) |
| (GETTOPVAL var) | ??<br>(symbol-value atom)<br>-- no concept to top level value in CL<br>BVM -- "I would translate GETTOPVAL and SETTOPVAL as symbol-value and set (not identity and setq), with a warning that they're wrong." |
| (SETTOPVAL var value) | ??<br>(set var value) |
| (GETATOMVAL atom) | (symbol-value atom)<br>BVM - "{GET\|SET}ATOMVAL are exactly symbol-value and set, with the implicit declaration, irrelevant to common lisp, that the variable is not dynamically bound." |
| (SETATOMVAL atom value) | (set var value) |

**2.4.3 Property Lists**

| | |
|---|---|
| (GETPROP atom prop) | (get atom prop)<br>BVM -- "GETPROP is really (and (symbolp atom) (get atom prop)), though you'll usually want it translated directly as get. Fortunately, PUTPROP does not suffer this brain damage." |
| (PUTPROP atom prop val) | (setf (get atom prop) val) |

| | |
|---|---|
| (ADDPROP atom prop new flg) | ??<br>-- no direction translation (runtime?) |
| (REMPROP atom prop) | (remprop atom prop) |
| (REMPROPLIST atom prop) | ??<br>-- no direction translation (runtime?) |
| (CHANGEPROP x prop1 prop2) | ??<br>-- no direction translation (runtime?) |
| (PROPNAMES atom) | ??<br>-- no direction translation (runtime?) |
| (DEFLIST l prop) | ??<br>-- no direction translation (runtime?)<br>LMM -- "Surely obsolete and not necessary." |
| (GETPROPLIST atom) | (symbol-plist atom) |
| (SETPROPLIST atom list) | (setf (symbol-plist atom) list) |
| (GETLIS x props) | ??<br>(multiple-value-bind (prop value tail)<br>(get-properties (symbol-plist x) props)<br>tail) |

**2.4.4 Print Names**

**Most of this section is extremely problematic -- especially since, although functions may be written that capture much of the semantic content, they tend to much more cons'y then their Interlisp counterparts, hence will disrupt the performance profile of any translated program that exploits these features.**

AD -- "I'd be tempted to leave most of the atom-building functions untranslated and flag them as something that the programmer should deal with himself. Except for very simple things, you will probably want to do whatever you were doing with atoms in some other way in CL."

BVM -- "I tend to agree with Andy. However, some of these are common enough that it might be worthwhile having approximate definitions in the library. E.g., write a version of MKATOM that does ordinary strings and numbers (the definition I wrote is close; slightly better might be one that did read-from-string while binding *readtable* to a table in which all the special characters have been given alphabetic syntax). Translate SUBATOM, PACK, PACK* as (MKATOM something), and then just flag all the MKATOMs uniformly.

It doesn't seem worth even trying for UNPACK, as any use is highly likely to need manual intervention anyway."

BVM - "Given that IL is so willing, and CL so unwilling, to coerce to strings, you might introduce a coerce-to-string macro to make some "translations" more palatable. If the translator knows how to evaluate it for constant forms (such as strings), so much the better."

(MKATOM x)

```
?? This is hard to capture exactly -- but here's one attempt
(defun mkatom (arg)
        (if (numberp arg)
                arg
                (values
                  (intern
                        (typecase arg
                          (symbol (string arg))
                          (string arg)
                          (otherwise
                                (prin1-to-string arg)))
                        ))
))
```

and another (due to BVM)

```
(defun mkatom (arg)
  (let ((string (typecase arg
                  (symbol (string arg))
                  (string arg)
                  (otherwise (prin1-to-string arg)))))
    (multiple-value-bind (n end)
        (parse-integer string :junk-allowed t)
      (if (and n (= end (length string)))
          n
          (values (intern string
                          ))))
```

)) BVM -- "Of course, this still doesn't do (mkatom "123Q") or (mkatom "12E3") correctly (yecch)."

(SUBATOM x n m)

??

Again here's a (long and cons'y) attempt at translation

```
(defun subatom (x n &optional  (m -1))
(let* ((string (symbol-name x))
       (start (if (< n 0)
                  (+ (length string) n)
                  (1- n)))
       (end (if (< m 0)
                  (+ 1 (length string) m)
                  m)))
       (values (intern (subseq string start end)))
))
```

or

```
(MKATOM (subseq
                    (string x)
                    (if (< n 0 )
                      (+ (length string) n)
                      (1- n))
                    (if (< m 0 )
                      (+1 (length string) m)
                      m)))
```

(PACK x)

??

But try

```
(defun pack (arglist)
  (let ((new-arglist
           (mapcar
                  #'(lambda (arg)
                          (typecase arg
                                  (symbol (string arg))
                                  (string arg)
                                  (otherwise
                                   (prin1-to-string arg))))
                  arglist)))
    (values (intern
       (apply #'concatenate 'string new-arglist)))
))
```

or

```
(MKATOM
(apply
  #'concatenate
  'string
  (mapcar
       #'(lambda (arg)
                  (typecase arg
                          (symbol (string arg))
                          (string arg)
                          (otherwise
```

```
                                                    (prin1-to-string arg))))
            x)))
```

(PACK* x1 x2 .. xn)

??
But try

```
(values
  (intern (apply #'concatenate
            'string
            (mapcar #'princ-to-string
                    (list x1 x2 .. xn)))))
```

or

```
(MKATOM
  (apply
            #'concatenate
            'string
            (mapcar #'princ-to-string
                    (list x1 x2 ".." xn))))
```

(UNPACK x flg rdtbl)

??
But try

```
(defun unpack (arg)
 (let ((string (typecase arg
                    (symbol (string arg))
                    (string arg)
                    (otherwise (prin1-to-string arg))))
            (result nil)
            (ch nil))
   (with-collection
     (dotimes (i (length string))
            (setq ch (char string i))
            (collect
                    (or (digit-char-p ch)
                       (intern
                              (string (char string i))))
            )))
))
```

A more Common Lisp'y version is:

```
(defun unpack (arg)
(let ((string
            (typecase arg
                    (symbol (string arg))
                    (string arg)
                    (otherwise (prin1-to-string arg))))
      )
(coerce string 'list)             ))
```

(DUNPACK x scatchlist flg rdtbl)

"
BVM -- "I see no need for DCHCON and DUNPACK to translate differently than CHCON and UNPACK, though the translations may want to be flagged (but then, you need to flag them anyway)"

(NCHARS x flg rdtbl)

??

```
(defun nchars
            (arg &optional (flg nil)
                    (*readtable* *readtable*))
            (length
                    (if flg
                       (prin1-to-string arg)
                    (princ-to-string arg)))
)
```

If flg is nil, this can be optimized to cut down on the consing.

(NTHCHAR x n flg rdtbl)

```
??
(let ((*readtable* (or rdtbl *readtable*)))
  (if flg
        (values (intern
                        (aref (prin1-to-string x) (1- n))))
        (values (intern
                        (aref (princ-to-string x) (1- n))))
)
```

Use of this function almost surely indicates a stylistic problem -- single letter symbols being used as character objects

(L-CASE x flg)

```
??
(typecase x
        (string (if flg
                        (string-capitalize x) ;;not quite
                        (string-downcase x)))
        (symbol
                (values (intern
                        (if flg
                                (string-capitalize x)
                                (string-downcase x)))))
        (cons
                (mapcar #'L-CASE x)))
```

(U-CASE x)

```
??
(typecase x
        (string (string-upcase x))
        (symbol
                (values (intern
                        (string-upcase x))))
        (cons
                (mapcar #'U-CASE x)))
```

(GENSYM char)

```
(gensym (if char (string char)))
```

Although this translation may well in subtle ways

GENNUM

?? -- no corresponding var in Common Lisp

(MAPATOMS fn)

```
(do-all-symbols (dummy-var)
        (funcall fn dummy-var))
```

Although do-all-symbols is not guaranteed to touch each symbol only once.

### 2.4.5 Character Code Functions

**This section forces to face squarely the problem of Interlisp's penchant of representing character objects as symbols with single letter p-names.**

(PACKC x)

```
??
(MKATOM (coerce
                (mapcar #'code-char x) 'string))
```

(CHCON x flg rdtbl)

```
??
(mapcar #'(lambda (sym)
                (char (symbol-name sym) 0))
        (UNPACK x flg rdtbl))
```

(DCHCON x scatchlist flg rdtbl)

"

(NTHCHARCODE x n flg rdtbl)

```
??
(char-code (char (symbol-name
                (NTHCHAR x n flg rdtbl)) 0))
```

Not quite right since NTHCHARCODE may return NIL in some circumstances

(CHCON1 x)

??
(char-code (char (symbol-name x) 0))
BVM - "Your translation of CHCON1 oddly assumes the arg is a symbol, rather than an arbitrary printable object"

(CHARACTER n)

??
(MKATOM (string (code-char x)))

(FCHARACTER n)

"

(CHARCODE c)

??

```
(defun charcode-1 (c)
  (etypecase c
        (symbol
              (case symbol
                (CR 13)
                ...
                (otherwise
                  (char-code (char
                                    (symbol-name c)
                                    0)))))
        (string
                (char-code (char c 0)))
        (cons
              (cons (charcode-1 (car c))
                    (charcode-1 (cdr c)))))
)
(defmacro charcode (c)
        (charcode-1 c ))
```

or in many cases
(char-code "some character object")
BVM - "CHARCODE should probably *always* translate as (char-code #\somechar), to facilitate conversion to the character idiom."

(SELCHARQ e c1 .. cn default)

```
(defmacro (e &rest args)
  (let ((default (car (last args)))
        (clauses (butlast args 1)))
    '(SELECTQ ,e
        ,@(mapcar
              #'(lambda (clause)
                    '(, (CHARCODE (car clause)) .
                          ,@(cdr clause))) clauses)
        ,default))
)
```

**2.5 Lists**

(CONS x y)

(cons x y)

(CAR x)

(car x)

(CDR x)

(cdr x)

(CAAR x)
.....
(CDDDR x)

(caar x)
.....
(cdddr x)

(RPLACD x y)
(FRPLACD x y)

(rplacd x y)
"

| | |
|---|---|
| (RPLACA x y)<br>(FRPLACA x y) | (rplaca x y)<br>" |
| (RPLNODE x a d)<br>(FRPLNODE x a d) | (rplacd (rplaca x a) d)<br>" |
| (RPLNODE2 x y)<br>(FRPLNODE2 x y) | (rplacd (rplaca x (car y)) (cdr y))<br>" |

### 2.5.1 Creating Lists

| | |
|---|---|
| (MKLIST x) | (if (listp x) x (list x)) |
| (LIST x1 x2 .. xn) | (list x1 x2 .. xn) |
| (APPEND x1 x2 .. xn) | (append x1 x2 .. xn) |
| (APPEND x) | (copy-list x) |
| (NCONC x1 x2 .. xn) | (nconc x1 x2 .. xn) |
| (NCONC1 lst x) | (nconc lst (list x)) |
| (ATTACH x l) | ?? -- probably obsolete<br>(defun attach (x l)<br>  (if (null l)<br>  (cons x l)<br>  (progn (setf (cdr l)<br>  (cons (car l) (cdr l)))<br>  (rplaca l x)))<br>) |

### 2.5.2 Building Lists from Left to Right

| | |
|---|---|
| (TCONC ptr x) | ??<br>(defun tconc (ptr x)<br>  (let ((head (car ptr))<br>  (tail (cdr ptr)))<br>  (if (null head)<br>  (let ((result (list x)))<br>  (cons result result))<br>  (progn (setf (cdr ptr)<br>  (cdr (rplacd tail (list x))))<br>  ptr)))<br>) |
| (LCONC ptr x) | ??<br>(defun lconc (ptr x)<br>  (let ((head (car ptr))<br>  (tail (cdr ptr)))<br>  (if (null head)<br>  (cons x (last x))<br>  (progn (setf (cdr ptr)<br>  (last (rplacd tail x)))<br>  ptr)))<br>) |
| (DOCOLLECT item lst) | ?? |
| (ENDCOLLECT item tail) | ?? |

### 2.5.3 Copying Lists

| | |
|---|---|
| (COPY x) | (copy-tree x ) |
| (COPYALL x) | ?? |
| (HCOPYALL x) | ?? |

Note from LMM:

"I've no trouble with your LIST translations. Are you sure CL has RPLACD? I thought you have to do (progn (SETF (CDR x) y) x).

I think the Interlisp character functions point up a kind of design choice that will come up again and again, in situations where the fundamental mechanism for getting something done in CL and IL differ.

I think a the translator might offer three choices:

(a) leave the functions alone (e.g., translate to IL:DCHCON and IL:MKATOM which are defined in a "compatibility" package). This gives code that works.

(b) produce "interim" translations, which have the same effect, e.g., as you've identified in your last message.

(c) attempt to produce "natural" Common Lisp style (examples follow.)

In the case of an Interlisp program that does PACKC, CHCON, DCHCON, in some cases the "native" CL program would use strings, and others, it would use symbols. (Interlisp programmers use symbols where CL programmers would use strings.)

Usually, the "native" trnaslation of CL functions that deal in character codes is to translate them to deal in character objects. Sometimes, where an IL programmer deals with a list of characters or character codes, the CL programmer would leave it as a string; the problem was that IL didn't have the breadth of sequence functions and so IL programmers would frequently hack lists.

If IL:character/code/atom/list == CL: character/character/string/list

then

(PACKC x) => (coerce 'string x)
(CHCON x) => (coerce 'list x)
(CHCON x flg rdtbl) => (coerce 'list (write-to-string x))

Ignore & flag RDTBL argument

(DCHCON ...) => ignore & use CHCON

(NTHCHARCODE ...) => SCHAR

CHCON1 => SCHAR ... 1

CHARACTER => no-op
FCHARACTER

CHARCODE => use #\.

SELCHARQ => CASE with #\ as case elements"

### 2.5.4 Extracting Tails of Lists

(TAILP x y)

(tailp x y)

(NTH x n)

((lambda (list index) (nthcdr (1- index) list)) x n)
BVM - "NTH returns tails, is one-based and has stupid behavior for n < 1"

(FNTH x n)

"

(LAST x)

(last x)
Although the behavior of last on non-list is not defined

(FLAST x)

"

(NLEFT l n tail)

??

```
(defun nleft (l n tail)
  (if (and tail (tailp tail l))
          (let* ((length (length l))
                 (sub-length (length tail))
                 (diff (- length sub-length n)))
            (if (>= diff 0)
                  (dotimes (i diff l)
                          (setq l (cdr l))))))
)
```

BVM - "The CL translation of the Interlisp definition of NLEFT would be substantially better than the one you give."

(LASTN l n)

?? is LASTN destructive?

### 2.5.5 Counting List Cells

**EQLENGTH, COUNTDOWN, and EQUALN are applicable to circular lists.**
**BVM - "I think worrying about il:equal is a waste of energy. The subtle difference between il:equal and cl:equal should be globally noted as a potential, albeit unlikely, source of incompatibility."**

(LENGTH x)

(length x)
Although length is only defined for true lists

(FLENGTH x)

"

(EQLENGTH x n)

(eql (length x) n)
Although would fail to return if x were circular
BVM - "For its non-circularity consideration, a more faithful translation might be ((lambda (tail) (and (consp tail) (atom (cdr tail)))) (nthcdr (1- n) x)), but it is less obvious what is going on."

(COUNT x)

??

```
(defun count (x)
  (+ (length x)
     (let ((sum 0))
          (dolist (a x)
             (if (consp a)
                   (incf sum (count a))))))
)
```

(COUNTDOWN x n)

??

(EQUALN x y depth)

??

```
(defun equaln (x y depth)
```

```
(cond ((eq depth 0) t)
      ((consp x)
            (and (consp y)
                 (equaln (car x) (car y) (1- depth))))
      (t
            (and (not (consp y))
                 (equal x y))))
)
```

NB equal not equivalent to IL:EQUAL

**2.5.6 Logical Operators**

(LDIFF x y)

(ldiff x y)
Except if y is not a tail of x. (LDIFF would signal an error in this case while ldiff would return (copy-list x))
NB -- if y is nil (LDIFF x y) -> x
BVM - "You might want to recognize the idiom (LDIFF lst (NLEFT lst n)) as (butlast lst n)"

(LDIFFERENCE x y)

(set-difference x y :test #'equal)
NB. equal is not equivalent to IL:EQUAL.

(INTERSECTION x y)

(intersection x y :test #'equal)
NB. equal is not equivalent to IL:EQUAL.
Elimination of duplicate entries is not guaranteed by CLtL.
BVM - "The fact that INTERSECTION advertises duplicate removal suggests that the conservative translation should be (remove-duplicates (intersection x y :test #'equal) :test #'equal)"
BVM - "Recognize the common idiom (INTERSECTION x x) as (remove-duplicates x :test #'equal)"

(UNION x y)

(union x y :test #'equal)
NB. equal is not equivalent to IL:EQUAL.
Again -- treatment of duplicate entries may not be identical.

**2.5.7 Searching Lists**

(MEMB x y)

(member x y :test #'eq)
Not defined if y is not a true list

(FMEMB x y)

"

(MEMBER x y)

(member x y :test #'equal)
NB. equal is not equivalent to IL:EQUAL

(EQMEMB x y)

```
(or (eq x y) (and (consp y)
                  (member x y :test #'eq)))
```

**2.5.8 Substitution Functions**

(SUBST new old expr)

(subst new old expr :test #'equal)
NB. equal is not equivalent to IL:EQUAL.
With this translation, if new is a consp, then new will NOT be copied on each substitution.

(DSUBST new old expr)

(nsubst new old expr :test #'equal)
Same caveat as for SUBST

(LSUBST new old expr)

??
(sort of an nconc subst)

(SUBLIS alst expr flg)

```
(if (null flg)
    (sublis alst expr :test #'equal)
  ??)
```

NB. The usual equal caveat holds. If (eq flg t) then SUBLIS is required to cons an entirely new tree

(DSUBLIS alst expr flg)

```
(if (null flg)
   (nsublis alst expr :test #'equal)
 ??)
```

Same caveat as SUBLIS

(SUBPAIR old new expr flg)

??
ignoring flg and the strange behavior with respect to non-nil final tails of old, roughly equivalent to:

```
(sublis (mapcar #'cons old new) expr :test #'equal)
```

**2.5.9 Association Lists and Property Lists**

(ASSOC key alst)

```
(assoc key alst :test #'eq)
```

But not equivalent if alst is not a true list

BVM - "For ASSOC, DREMOVE, etc, I think you should use test eql instead of eq (in fact, isn't that the default in cl?). This is actually a more widespread and difficult problem with translating IL code--the hidden assumption that a substantial class of integers are immediate and hence testable by eq."

JOP - "I'm not sure I agree with the rational for using eql rather than eq in ASSOC (and friends), for the following reasons: (a) the keys for ASSOC (etc.) are usually symbols, and (b) Although not explicitly stated in CLtL -- it's probably fairly safe to assume that eq comparisons are valid for fixnums."

BVM - "I thought CLtL did explicitly state (p. 193) that it is NOT safe to assume that eq comparisons are valid for fixnums. This is not to say that I am aware of any implementations in which eql fixnums are not eq. However, there are certainly implementations in which the fixnum range is considerably smaller than ours, another subtle obstacle in porting. As for your point (a), it is my impression that people are fairly sloppy about whether assoc keys are symbols or not. Aside from all that, there's a reason that CLtL's default for assoc, etc, is eql. I think that translating il:assoc directly as cl:assoc is appropriate style; at worst, it performs slightly less efficiently than with an eq test, but you know it won't be wrong."

LMM - "The decision of EQ vs EQL in ASSOC is probably one of those decisions to made interactively at translation time..."

(FASSOC key alst)

"

(SASSOC key alst)

```
(assoc key alst :test #'equal)
```

Usual caveat about equal -- non NIL tails of alst

(PUTASSOC key val alst)

??

```
(defun putassoc (key val alst)
  (let ((entry (assoc key alst :test #'eq)))
    (if entry
        (setf (cdr entry) val)
    (progn (nconc alst (cons key val))
                  val)))
)
```

LMM - "I've found on more than one occasion that to do a "natural" translation, I wound up changing an ALIST into a property list, e.g., so I could use (SETF (GETF x y) z) instead of (PUTASSOC x y z)."

(LISTGET lst prop) — (getf lst prop)

(LISTPUT lst prop val) — (setf (getf lst prop) val)

(LISTGET1 lst prop) — (cdr (member prop lst :test #'eq))
NB. Order of evaluation not preserved

(LISTPUT1 lst prop val) — (setf (cdr (member prop lst :test #'eq)) val)
NB. Order of evaluation not preserved

**2.5.10 Other List Functions**

(REMOVE x l) — (remove x l :test #'equal)
NB. equal not equivalent to IL:EQUAL

(DREMOVE x l) — (delete x l :test #'eq)
Not guaranteed to return an eq list if the result is non-nil

(REVERSE l) — (reverse l)
Not equivalent if l is not a list

(DREVERSE l) — (nreverse l)
Same caveat as REVERSE

**2.6 Strings**

**Some thorny issues arise here. Among them: (a.) Some Interlisp string functions will clearly not be applicable to all types of strings (eg GNC GLC), (b.) Some agreement must be attained between the allowable set of character objects and string-chars -- this may confine us to the 96 standard characters, excluding control characters, NS characters, etc. (c.) Reusing string headers is a fairly inoperative idea -- although doable if the reusable string is adjustable (d.) It may be nice to have some general technology for mapping from a index-origin-one indexing scheme to a index-origin-zero indexing scheme. This may include fairly global source modifications**

(STREQUAL x y) — (string= x y)

(ALLOCSTRING n initchar) — (make-string n :initial-element (char-code initchar))

(ALLOCSTRING n initchar old) — ??
(adjust-array old n :initial-element (char-code initchar))

(MKSTRING x flg rdtbl) — ??

```
(defun mkstring
     (x &optional flg (rdtbl *readtable*))
  (let ((*readtable* rdtbl))
    (if (null flg)
          (typecase x
            (string x)
            (symbol (symbol-name x))
            (otherwise
               (princ-to-string x)))
    (prin1-to-string x))
))
```

(SUBSTRING x n m) — ??

```
(defun substring (x n &optional (m -1))
  (let* ((length (length x))
```

```
                (start (if (< n 0)
                           (+ length n)
                          (1- n)))
                (end (if (< m 0)
                           (+1 length m)
                          m)))
   (make-array (- end start)
          :element-type 'string-char
          :displaced-to x
          :displaced-index-offset start)
))
```

(SUBSTRING x n m oldptr)

??
Might be able to do something if oldptr were an adjustable string

(GNC x)

??
Requires x to be adjustable

```
(defun gnc (x)
  (let ((holder (make-array (length x)
                            :element-type 'string-char
                            :displaced-to x)))
    (prog1 (char x 0)
          (adjust-array x (1- (length x))
                  :displaced-to holder
                  :displaced-index-offset 1)))
)
```

I'm not sure what would happen if the translation were simply

```
(prog1 (char x 0)
          (adjust-array x (1- (length x))
                  :displaced-to x
                  :displaced-index-offset 1))
```

Note that a character object is returned rather than a symbol

(GLC x)

??
x required to have a fill-pointer

```
(vector-pop x)
```

Note that a character object is returned rather than a symbol

(CONCAT x1 x2 .. xn)

??

```
(concatenate 'string (MKSTRING x1)
                     (MKSTRING x2)
                     ..
                     (MKSTRING xn))
```

(CONCAT)

```
(make-string 0)
```

(CONCATLIST x)

```
(apply #'CONCAT x)
```

(RPLSTRING x n y)

??

```
(defun rplstring (x n y)
  (let ((start (if (< n 0)
                   (+ (length x) n)
                  (1- n))))
    (do ((i 0 (1+ i))
         (limit (length y))
         (j start (1+ j)))
        ((eql j limit) x)
      (setf (char x j) (char y i))))
)
```

(RPLCHARCODE x n charcode)

??

```
(defun rplcharcode (x n charcode)
  (let ((index (if (< n 0)
```

```
                                (+ (length x) n)
                               (1- n))))
    (Setf (char x index) (char-code charcode))
    x)
)
```

(STRPOS pat string start)

??
roughly
(1+ (search pat string :start1 (1- start))

(STRPOS pat string start skip anchor tail)

??

(STRPOSL a str strat)

??
roughly

```
(1+ (search (mapcar #'code-char a)
                    str :start1 (1- start)))
```

(MAKEBITTABLE l neg a)

??

**2.7 Arrays**

**Suppose Interlisp arrays are represented by Common Lisp vectors, then two strategies present themselves for translation of the array facilities.**
**a.) Perform everywhere suitable subtractions -- but attempt global code simplification**
**b.) Use an addition vector cell and preserve origin-1 indexing**
**I will attempt to list translations appropriate for both strategies**

**NB. The index origin of a translated Interlisp vector will not be knowable at run-time.**

**BVM - "Since you can't tell by looking at a call to ELT or SETA whether the array is 0- or 1-origin, you can only use method "a" (subtract 1 from all indices) if the user is willing to globally declare "I never use zero-origin arrays". Note that when using method "b", you have to inflate the size of the vector by 1 even on calls to ARRAY with origin constant zero, unless you never care about ARRAYSIZE translating correctly."**

**Interlisp array element-types may be translated as follows**

| | |
|---|---|
| BIT | bit |
| BYTE | (unsigned-byte 8) |
| WORD | (unsigned-byte 16) |
| FLOATP | float |
| POINTER | t |
| DOUBLEPOINTER | ?? |
| XPOINTER | ?? |
| FLAG | (member t nil) |
| (BITS n) | (unsigned-byte n) |
| FIXP | (signed-byte 32) or t |
| SIGNEDWORD | (signed-byte 16) |

**One might imagine two functions -- translate-type and inverse-translate-type -- to move from Interlisp types to Common Lisp types and back again**

(ARRAY size type init)

a.)
```
(make-array size :element-type
        (translate-type type) :initial-element
        init)
```
b.)
```
(make-array (1+ size) :element-type
        (translate-type type) :initial-element
        init)
```
Of course, if the array origin is explicitly specified as zero (0), then translation a.) may always be employed

| | |
|---|---|
| (ELT a n) | a.) (aref a (1- n))<br>b.) (aref a n) |
| (SETA a n v) | a.) (setf (aref a (1- n)) v)<br>b.) (setf (aref a n) v) |
| (ARRAYTYP a) | (inverse-translate-type<br>(array-element-type a)) |
| (ARRAYSIZE a) | a.) (array-total-size a)<br>b.) (1- (array-total-size a)) |
| (ARRAYORIG a) | ??<br>always 1?<br>BVM - "I can't imagine any use for ARRAYORIG other than as an ORIGIN argument to ARRAY, where it will be fine to throw it out; any other use is untranslatable. Well, maybe some index checker would use it, in which case zero would be a safe translation." |
| (COPYARRAY a) | (copy-seq a) |
| (ARRAYP a) | (vectorp a) |

**2.7 Arrays Interlisp-10 Arrays**

**Probably, no functions in this section need be supported by the translator. I list those not mentioned elsewhere here for completeness.**

| | |
|---|---|
| (ELTD a n) | ??<br>BVM - "ELTD and SETD can only be used on arrays of type DOUBLEPOINTER. You could tediously translate them as index (+ n (/ (1- (length a)) 2) 1), but it doesn't seem worth it. ARRAYBEG is blatantly untranslatable." |
| (SETD a n v) | ?? |
| (ARRAYBEG a) | ?? |

**2.8 Hash Arrays**

**Interlisp Harryp's will most likely be represented as Common Lisp hash-tables even though Interlisp Harryp's support options more extensive then those of their counterparts.**

**BVM - "All the hash functions need to watch out for harray = NIL for the bogus SYSHASHARRAY feature. Probably a global note in the translator's guide is sufficient; anyone who actually wrote a program depending on the feature deserves to lose."**

| | |
|---|---|
| (HARRAY len) | (make-hash-table :size len :test #'eq)<br>or<br>(make-hash-table :size len)<br>BVM - "In the case of HARRAY, you need to watch out for (list (harray len)) and (cons (harray len) overflow) and turn them into (make-hash-table :size len :rehash-size overflow). HARRAY all by itself is strictly speaking untranslatable, because it implicitly has overflow action ERROR." |
| (HASHARRAY minkeys) | " |
| (HASHARRAY minkeys overflow ) | (make-hash-table :size minkeys :rehash-size overflow)<br>BVM - "I believe the overflow arg to HASHARRAY is a superset of |

| | |
|---|---|
| | the allowable values to :rehash-size, though the commonly-used numeric values are compatible (hasharray also supports the values ERROR and arbitrary function)." |
| (HASHARRAY minkeys overflow nil nil nil rehash-threshold) | (make-hash-table :size minkeys :rehash-size overflow :rehash-threshold rehash-threshold) |
| (HASHARRAY minkeys overflow hashbitsfns equivfn nil rehash-threshold) | (make-hash-table :size minkeys :test (get-know-test-fn hashbitsfns equivfn) :rehash-size overflow :rehash-threshold rehash-threshold) |
| (HARRAYSIZE harray) | ?? |
| (CLRHASH harray) | (clrhash harray) |
| (PUTHASH key val harray) | ((lambda (x y z)<br>(if y (setf (gethash x z) y)<br>(remhash x z)))<br>key val harray)<br>BVM - "This is another good place for a simplifier, since val=nil is usually a constant. (Unfortunately, you can rarely get rid of the remhash arm--only if the value being stored is a non-nil constant.)" |
| (GETHASH key harray) | (values (gethash key harray))<br>BVM - "I hope the simplifier knows about eliminating (values &) in non-mv context." |
| (REHASH oldharray newharray) | ?? |
| (MAPHASH harray maphfn) | ((lambda (x y)<br>(maphash #'(lambda (key val) (funcall y val key)) x))<br>harray maphfn)<br>BVM - "Yet another place where a simplifier with sufficient smarts about lambdas would make the translation more pleasant in the common case where the maphfn is a lambda expression. Alternatively, arrange for the translator to manually permute the arg list." |
| (DMPHASH harray1 ... harrayn) | (progn (print '(setq harray1 ,harray1))<br>...<br>(print '(setq harrayn ,harrayn))) |
| (HARRAYPROP harray prop) | ??<br>BVM - ".. the only instance of which we can translate is (HARRAYPROP a 'NUMKEYS) => (hash-table-count a)." |
| (HARRAYPROP harray prop nv) | ?? |

**2.9 Numeric and Arithmetic Functions**

**Since Common Lisp arithmetic functions are fully generic -- the type specific Interlisp arithmetic functions pose a problem. They can either be a.) Correctly translated with a substantial cost in performance and complexity or b.) incorrectly translated to their generic counterparts. I will give translation for both possibilities.**

**BVM - "I suspect most people will want the type-specific operations to translate generically (in code I've looked at, I virtually always do), even though this will occasionally cause subtle bugs."**

**There may be redundancy in the following section for completeness.**

**Many of the following predicates could be simplified in a test context.**

| | |
|---|---|
| (SMALLP x) | ((lambda (x) (and (typep x 'fixnum) x)) x) |

| | |
|---|---|
| (FIXP x) | ((lambda (x) (and (integerp x) x)) x) |
| (FLOATP x) | ((lambda (x) (and (floatp x) x)) x) |
| (NUMBERP x) | ((lambda (x) (and (numberp x) x)) x) |
| MIN.SMALLP | most-negative-fixnum |
| MAX.SMALLP | most-positive-fixnum<br>BVM - "MAX.SMALLP is often used as a synonym for 2↑16-1, so this translation should be flagged." |
| MIN.FIXP | ?? |
| MAX.FIXP | ?? |
| MIN.INTEGER | ?? |
| MAX.INTEGER | ??<br>BVM - "MIN.INTEGER & MAX.INTEGER are obviously untranslatable, but I think we've even de-documented them." |
| (OVERFLOW flg) | ?? |
| (IPLUS x1 ... xn) | a.) (+ (truncate x1) ... (truncate xn))<br>b.) (+ x1 .. xn) |
| (PLUS x1 .. xn) | (+ x1 ... xn) |
| (FPLUS x1 ... xn) | a.) (+ (float x1) .... (float xn))<br>b.) (+ x1 .. xn) |
| (IMINUS x) | a.) (- (truncate x))<br>b.) (- x) |
| (MINUS x) | (- x) |
| (FMINUS x) | a.) (- (float x))<br>b.) (- x) |
| (IDIFFERENCE x y) | a.) (- (truncate x) (truncate y))<br>b.) (- x y) |
| (DIFFERENCE x y) | (- x y) |
| (FDIFFERENCE x y) | a.) (- (float x) (float y))<br>b.) (- x y) |
| (ITIMES x1 ... xn) | a.) (* (truncate x1) ... (truncate xn))<br>b.) (* x1 .. xn) |
| (TIMES x1 .. xn) | (* x1 ... xn) |
| (FTIMES x1 ... xn) | a.) (* (float x1) .... (float xn))<br>b.) (* x1 .. xn) |
| (IQUOTIENT x y) | (truncate x y) |
| (QUOTIENT x y) | a.) ??<br>b.) (/ x y) -- although this is likely to be wrong more often than not<br>BVM - "QUOTIENT -- I think it should only be translated as / in the case where you know that one of its args is floatp; usage tends not to be very consistent. So (if (or (floatp x) (floatp y)) (/ x y) (truncate x y)) is better, if ugly." |
| (FQUOTIENT x y) | a.) (/ (float x) (float y))<br>b.) (/ x y) -- fairly safe |

| | |
|---|---|
| (IGREATERP x y) | a.) (> (truncate x) (truncate y))<br>b.) (> x y) |
| (GREATERP x y)<br>(FGREATERP x y) | (> x y)<br>"" |
| (ILESSP x y) | a.) (< (truncate x) (truncate y))<br>b.) (< x y) |
| (LESSP x y)<br>(FLESSP x y) | (< x y)<br>"" |
| (IGEQ x y) | a.) (>= (truncate x) (truncate y))<br>b.) (>= x y) |
| (GEQP x y)<br>(FGEQP x y) | (>= x y)<br>"" |
| (ILEQ x y) | a.) (<= (truncate x) (truncate y))<br>b.) (<= x y) |
| (LEQP x y)<br>(FLEQP x y) | (<= x y)<br>"" |
| (IEQP x y) | a.) (= (truncate x) (truncate y))<br>b.) (= x y) |
| (EQP x y) | (= x y)<br>Strictly incorrect, but probably good enough |
| (FEQP x y) | "" |
| (IREMAINDER x y) | a.) (rem (truncate x) (truncate y))<br>b.) (rem x y) |
| (REMAINDER x y) | (rem x y) |
| (FREMAINDER x y) | a.) (rem (float x) (float y))<br>b.) (rem x y) |
| (IMIN x1 ... xn) | a.) (min (truncate x1) ... (truncate xn))<br>close, but not correct since (IMIN 1.2 1.1) returns 1.2<br>b.) (min x1 ... xn)<br>BVM - "For IMIN, it happens to be a bug that (IMIN 1.2 1.1) returns 1.2, so I wouldn't sweat it. Ditto IMAX and IABS." |
| (MIN x1 ... xn) | (min x1 ... xn) |
| (FMIN x1 ... xn) | a.) (min (float x1) ... (float xn))<br>b.) (min x1 ... xn) |
| (IMAX x1 ... xn) | a.) (max (truncate x1) ... (truncate xn))<br>close, but not correct since (IMAX 1.1 1.2) returns 1.1<br>b.) (max x1 ... xn) |
| (MAX x1 ... xn) | (max x1 ... xn) |
| (FMAX x1 ... xn) | a.) (max (float x1) ... (float xn))<br>b.) (min x1 ... xn) |

| | |
|---|---|
| (IABS x) | a.) (abs (truncate x))<br>Not quite right, since (IABS -0.1) returns -0.1<br>b.) (abs x) |
| (ABS x) | (abs x) |
| (FABS x) | a.) (abs (float x))<br>b.) (abs x) |
| (ADD1 x) | (1+ x) |
| (SUB1 x) | (1- x) |
| (ZEROP x) | (zerop x) |
| (MINUP x) | (minusp x) |
| (FIX x) | (truncate x) |
| (GCD x y) | (gcd x y) |

**2.9.2 Logical Arithmetic Functions**

| | |
|---|---|
| (LOGAND x1 .. xn) | (logand x1 .. xn) |
| (LOGOR x1 .. xn) | (logior x1 .. xn) |
| (LOGXOR x1 .. xn) | (logxor x1 .. xn) |
| (LSH x n) | (ash x n) |
| (RSH x n) | (ash x (- n)) |
| (LLSH x n) | ??<br>usually (ash x n) will suffice |
| (LRSH x n) | ??<br>usually (ash x (- n)) will suffice |
| (INTEGERLENGTH n) | (if (< n 0)<br>(1+ (integer-length n)<br>(integer-length n)) |
| (POWEROFTWOP n) | ??<br>roughly (zerop (logand n (1- n))) |
| (EVENP x) | (evenp x) |
| (EVENP x y) | (zerop (mod x y)) |
| (ODDP x) | (oddp x) |
| (ODDP x y) | (not (zerop (mod x y))) |
| (LOGNOT n) | (lognot n) |
| (BITTEST n mask) | (logtest n mask) |
| (BITCLEAR n mask) | (logandc2 n mask) |
| (BITSET n mask) | (logior n mask) |

| | |
|---|---|
| (MASK.1'S position size) | ((lambda (x y) (ldb (byte y x) -1)) position size) |
| (MASK.0'S position size) | ((lambda (x y) (dpb 0 (byte y x) -1)) position size) |
| (LOADBYTE n position size) | ((lambda (x y z) (ldb (byte z y) x)) n position size) |
| (DEPOSITBYTE n position size byte) | ((lambda (w x y z) (dpb z (byte y x) w)) n position size byte) |
| (ROT x n fieldsize) | ?? |
| (BYTE size position) | (byte size position) |
| (BYTESIZE bytespec) | (byte-size bytespec) |
| (BYTEPOSITION bytespec) | (byte-position bytespec) |
| (LDB bytespec val) | (ldb bytespec val) |
| (DPB n bytespec val) | (dpb n bytespec val) |

**2.9.3 Floating Point Arithmetic**

| | |
|---|---|
| MIN.FLOAT | most-negative-single-float |
| MAX.FLOAT | most-positive-single-float |
| (FLOAT x) | (float x) |

**2.9.5 Special Functions**

| | |
|---|---|
| (EXPT m n) | (expt m n) |
| (SQRT n) | (sqrt n) |
| (LOG x) | (log x) |
| (ANTILOG x) | (exp x) |
| (SIN x) | (sin (degrees-to-radians x))<br>where<br>(defun degrees-to-radians (degrees)<br>(* (/ pi 180) degrees)) |
| (SIN x t) | (sin x) |
| (COS x) | (cos (degrees-to-radians x)) |
| (COS x t) | (cos x) |
| (TAN x) | (tan (degrees-to-radians x)) |
| (TAN x t) | (tan x) |
| (ARCSIN x) | (radians-to-degrees (asin (degrees-to-radians x)))<br>where<br>(defun radians-to-degrees (radians)<br>(* (/ 180 pi) radians)) |
| (ARCSIN x t) | (asin x) |

| | |
|---|---|
| (ARCCOS x) | (radians-to-degrees (acos (degrees-to-radians x))) |
| (ARCCOS x t) | (acos x) |
| (ARCTAN x) | (radians-to-degrees (atan (degrees-to-radians x)))<br>NB: The IRM claims the range of ARCTAN is [0, pi] -- while in the most current loadup the range is [-pi/2, pi/2]. The later situation agrees with Common Lisp. |
| (ARCTAN x t) | (atan x) |
| (ARCTAN2 x y) | (radians-to-degrees<br>(atan<br>(degrees-to-radians x)<br>(degrees-to-radians y))) |
| (ARCTAN2 x y t) | (atan x y) |
| (RAND lower upper) | ((lambda (x y) (+ x (random (1+ (- y x))))) lower upper)<br>NB. The 1+ to generate an inclusive upper bound is not correct if either x or y is of type float |
| (RAND) | (random (1+ most-positive-fixnum)) |
| (RANDSET X) | (defun randset (x)<br>(case x<br>((t) (setq *random-state*<br>(make-random-state)))<br>((nil) *random-state*)<br>(otherwise (setq *random-state* x)))<br>) |