GCL SI Manual

1 Numbers

`SIGNUM` (*number*) [Function]
Package:LISP

If NUMBER is zero, returns NUMBER; else returns (/ NUMBER (ABS NUMBER)).

`LOGNOT` (*integer*) [Function]
Package:LISP

Returns the bit-wise logical NOT of INTEGER.

`MOST-POSITIVE-SHORT-FLOAT` [Constant]
Package:LISP The short-float closest in value to positive infinity.

`INTEGER-DECODE-FLOAT` (*float*) [Function]
Package:LISP

Returns, as three values, the integer interpretation of significand F, the exponent E, and the sign S of the given float, so that E FLOAT = S * F * B where B = (FLOAT-RADIX FLOAT)

F is a non-negative integer, E is an integer, and S is either 1 or -1.

`MINUSP` (*number*) [Function]
Package:LISP

Returns T if NUMBER < 0; NIL otherwise.

`LOGORC1` (*integer1 integer2*) [Function]
Package:LISP

Returns the logical OR of (LOGNOT INTEGER1) and INTEGER2.

`MOST-NEGATIVE-SINGLE-FLOAT` [Constant]
Package:LISP Same as MOST-NEGATIVE-LONG-FLOAT.

`BOOLE-C1` [Constant]
Package:LISP Makes BOOLE return the complement of INTEGER1.

`LEAST-POSITIVE-SHORT-FLOAT` [Constant]
Package:LISP The positive short-float closest in value to zero.

`BIT-NAND` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical NAND on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`INT-CHAR` (*integer*) [Function]
Package:LISP

Performs the inverse of CHAR-INT. Equivalent to CODE-CHAR in GCL.

`CHAR-INT` (*char*) [Function]
Package:LISP

Returns the font, bits, and code attributes as a single non-negative integer. Equivalent to CHAR-CODE in GCL.

`LEAST-NEGATIVE-SINGLE-FLOAT` [Constant]
Package:LISP Same as LEAST-NEGATIVE-LONG-FLOAT.

`/=` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if no two of its arguments are numerically equal; NIL otherwise.

`LDB-TEST` (*bytespec integer*) [Function]
Package:LISP

Returns T if at least one of the bits in the specified bytes of INTEGER is 1; NIL otherwise.

`CHAR-CODE-LIMIT` [Constant]
Package:LISP The upper exclusive bound on values produced by CHAR-CODE.

`RATIONAL` (*number*) [Function]
Package:LISP

Converts NUMBER into rational accurately and returns it.

`PI` [Constant]
Package:LISP The floating-point number that is appropriately equal to the ratio of the circumference of the circle to the diameter.

`SIN` (*radians*) [Function]
Package:LISP

Returns the sine of RADIANS.

`BOOLE-ORC2` [Constant]
Package:LISP Makes BOOLE return LOGORC2 of INTEGER1 and INTEGER2.

`NUMERATOR` (*rational*) [Function]
Package:LISP

Returns as an integer the numerator of the given rational number.

`MASK-FIELD` (*bytespec integer*) [Function]
Package:LISP

Extracts the specified byte from INTEGER.

`INCF` [Special Form]
Package:LISP

Syntax:

```
(incf place [delta])
```

Adds the number produced by DELTA (which defaults to 1) to the number in PLACE.

`SINH` (*number*) [Function]
Package:LISP

Returns the hyperbolic sine of NUMBER.

`PHASE` (*number*) [Function]
Package:LISP

Returns the angle part of the polar representation of a complex number. For non-complex numbers, this is 0.

`BOOLE` (*op integer1 integer2*) [Function]
Package:LISP

Returns an integer produced by performing the logical operation specified by OP on the two integers. OP must be the value of one of the following constants: BOOLE-CLR BOOLE-C1 BOOLE-XOR BOOLE-ANDC1 BOOLE-SET BOOLE-C2 BOOLE-EQV BOOLE-ANDC2 BOOLE-1 BOOLE-AND BOOLE-NAND BOOLE-ORC1 BOOLE-2 BOOLE-IOR BOOLE-NOR BOOLE-ORC2 See the variable docs of these constants for their operations.

`SHORT-FLOAT-EPSILON` [Constant]
Package:LISP The smallest positive short-float that satisfies (not (= (float 1 e) (+ (float 1 e) e))).

`LOGORC2` (*integer1 integer2*) [Function]
Package:LISP

Returns the logical OR of INTEGER1 and (LOGNOT INTEGER2).

`BOOLE-C2` [Constant]
Package:LISP Makes BOOLE return the complement of INTEGER2.

`REALPART` (*number*) [Function]
Package:LISP

Extracts the real part of NUMBER.

`BOOLE-CLR` [Constant]
Package:LISP Makes BOOLE return 0.

`BOOLE-IOR` [Constant]
Package:LISP Makes BOOLE return LOGIOR of INTEGER1 and INTEGER2.

`FTRUNCATE` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Values: (quotient remainder) Same as TRUNCATE, but returns first value as a float.

`EQL` (*x y*) [Function]
Package:LISP

Returns T if X and Y are EQ, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character. Returns NIL otherwise.

`LOG` (*number* **&optional** *base*) [Function]
Package:LISP

Returns the logarithm of NUMBER in the base BASE. BASE defaults to the base of natural logarithms.

`DOUBLE-FLOAT-NEGATIVE-EPSILON` [Constant]
Package:LISP Same as LONG-FLOAT-NEGATIVE-EPSILON.

`LOGIOR` (**&rest** *integers*) [Function]
Package:LISP

Returns the bit-wise INCLUSIVE OR of its arguments.

`MOST-NEGATIVE-DOUBLE-FLOAT` [Constant]
Package:LISP Same as MOST-NEGATIVE-LONG-FLOAT.

`/` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Divides the first NUMBER by each of the subsequent NUMBERS. With one arg, returns the reciprocal of the number.

`*RANDOM-STATE*` [Variable]
Package:LISP The default random-state object used by RAMDOM.

`1+` (*number*) [Function]
Package:LISP

Returns NUMBER + 1.

`LEAST-NEGATIVE-DOUBLE-FLOAT` [Constant]
Package:LISP Same as LEAST-NEGATIVE-LONG-FLOAT.

`FCEILING` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Same as CEILING, but returns a float as the first value.

`MOST-POSITIVE-FIXNUM` [Constant]
Package:LISP The fixnum closest in value to positive infinity.

`BIT-ANDC1` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical ANDC1 on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`TAN` (*radians*) [Function]
Package:LISP

Returns the tangent of RADIANS.

`BOOLE-NAND` [Constant]
Package:LISP Makes BOOLE return LOGNAND of INTEGER1 and INTEGER2.

`TANH` (*number*) [Function]
Package:LISP

Returns the hyperbolic tangent of NUMBER.

`ASIN` (*number*) [Function]
Package:LISP

Returns the arc sine of NUMBER.

`BYTE` (*size position*) [Function]
Package:LISP

Returns a byte specifier. In GCL, a byte specifier is represented by a dotted pair (<size> . <position>).

`ASINH` (*number*) [Function]
Package:LISP

Returns the hyperbolic arc sine of NUMBER.

`MOST-POSITIVE-LONG-FLOAT` [Constant]
Package:LISP The long-float closest in value to positive infinity.

`SHIFTF` [Macro]
Package:LISP

Syntax:

```
(shiftf {place}+ newvalue)
```

Evaluates all PLACEs and NEWVALUE in turn, then assigns the value of each form to the PLACE on its left. Returns the original value of the leftmost form.

`LEAST-POSITIVE-LONG-FLOAT` [Constant]
Package:LISP The positive long-float closest in value to zero.

`DEPOSIT-FIELD` (*newbyte bytespec integer*) [Function]
Package:LISP

Returns an integer computed by replacing the specified byte of INTEGER with the specified byte of NEWBYTE.

`BIT-AND` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical AND on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`LOGNAND` (*integer1 integer2*) [Function]
Package:LISP

Returns the complement of the logical AND of INTEGER1 and INTEGER2.

`BYTE-POSITION` (*bytespec*) [Function]
Package:LISP

Returns the position part (in GCL, the cdr part) of the byte specifier.

`ROTATEF` [Macro]

Package:LISP

Syntax:

```
(rotatef {place}*)
```

Evaluates PLACEs in turn, then assigns to each PLACE the value of the form to its right. The rightmost PLACE gets the value of the leftmost PLACE. Returns NIL always.

`BIT-ANDC2` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]

Package:LISP

Performs a bit-wise logical ANDC2 on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`TRUNCATE` (*number* **&optional** *(divisor 1)*) [Function]

Package:LISP

Values: (quotient remainder) Returns NUMBER/DIVISOR as an integer, rounded toward 0. The second returned value is the remainder.

`BOOLE-EQV` [Constant]

Package:LISP Makes BOOLE return LOGEQV of INTEGER1 and INTEGER2.

`BOOLE-SET` [Constant]

Package:LISP Makes BOOLE return -1.

`LDB` (*bytespec integer*) [Function]

Package:LISP

Extracts and right-justifies the specified byte of INTEGER, and returns the result.

`BYTE-SIZE` (*bytespec*) [Function]

Package:LISP

Returns the size part (in GCL, the car part) of the byte specifier.

`SHORT-FLOAT-NEGATIVE-EPSILON` [Constant]

Package:LISP The smallest positive short-float that satisfies (not (= (float 1 e) (- (float 1 e) e))).

`REM` (*number divisor*) [Function]

Package:LISP

Returns the second value of (TRUNCATE NUMBER DIVISOR).

`MIN` (*number* **&rest** *more-numbers*) [Function]

Package:LISP

Returns the least of its arguments.

`EXP` (*number*) [Function]

Package:LISP

Calculates e raised to the power NUMBER, where e is the base of natural logarithms.

`DECODE-FLOAT` (*float*) [Function]
Package:LISP

Returns, as three values, the significand F, the exponent E, and the sign S of the given float, so that E FLOAT = S * F * B where B = (FLOAT-RADIX FLOAT)

S and F are floating-point numbers of the same float format as FLOAT, and E is an integer.

`LONG-FLOAT-EPSILON` [Constant]
Package:LISP The smallest positive long-float that satisfies (not (= (float 1 e) (+ (float 1 e) e))).

`FROUND` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Same as ROUND, but returns first value as a float.

`LOGEQV` (**&rest** *integers*) [Function]
Package:LISP

Returns the bit-wise EQUIVALENCE of its arguments.

`MOST-NEGATIVE-SHORT-FLOAT` [Constant]
Package:LISP The short-float closest in value to negative infinity.

`BIT-NOR` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical NOR on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`CEILING` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Returns the smallest integer not less than or NUMBER/DIVISOR. Returns the remainder as the second value.

`LEAST-NEGATIVE-SHORT-FLOAT` [Constant]
Package:LISP The negative short-float closest in value to zero.

`1-` (*number*) [Function]
Package:LISP

Returns NUMBER - 1.

`<=` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if arguments are in strictly non-decreasing order; NIL otherwise.

`IMAGPART` (*number*) [Function]
Package:LISP

Extracts the imaginary part of NUMBER.

`INTEGERP` (*x*) [Function]
Package:LISP

Returns T if X is an integer (fixnum or bignum); NIL otherwise.

`ASH` (*integer count*) [Function]
Package:LISP

Shifts INTEGER left by COUNT places. Shifts right if COUNT is negative.

`LCM` (*integer* **&rest** *more-integers*) [Function]
Package:LISP

Returns the least common multiple of the arguments.

`COS` (*radians*) [Function]
Package:LISP

Returns the cosine of RADIANS.

`DECF` [Special Form]
Package:LISP

Syntax:

```
(decf place [delta])
```

Subtracts the number produced by DELTA (which defaults to 1) from the number in PLACE.

`ATAN` (*x* **&optional** *(y 1)*) [Function]
Package:LISP Returns the arc tangent of X/Y.

`BOOLE-ANDC1` [Constant]
Package:LISP Makes BOOLE return LOGANDC1 of INTEGER1 and INTEGER2.

`COSH` (*number*) [Function]
Package:LISP Returns the hyperbolic cosine of NUMBER.

`FLOAT-RADIX` (*float*) [Function]
Package:LISP

Returns the representation radix (or base) of the floating-point number.

`ATANH` (*number*) [Function]
Package:LISP

Returns the hyperbolic arc tangent of NUMBER.

`EVENP` (*integer*) [Function]
Package:LISP Returns T if INTEGER is even. Returns NIL if INTEGER is odd.

`ZEROP` (*number*) [Function]
Package:LISP Returns T if NUMBER = 0; NIL otherwise.

`FLOATP` (*x*) [Function]
Package:LISP

Returns T if X is a floating-point number; NIL otherwise.

`SXHASH` (*object*) [Function]
Package:LISP

Computes a hash code for OBJECT and returns it as an integer.

`BOOLE-1` [Constant]
Package:LISP Makes BOOLE return INTEGER1.

`MOST-POSITIVE-SINGLE-FLOAT` [Constant]
Package:LISP Same as MOST-POSITIVE-LONG-FLOAT.

`LOGANDC1` (*integer1 integer2*) [Function]
Package:LISP

Returns the logical AND of (LOGNOT INTEGER1) and INTEGER2.

`LEAST-POSITIVE-SINGLE-FLOAT` [Constant]
Package:LISP Same as LEAST-POSITIVE-LONG-FLOAT.

`COMPLEXP` (*x*) [Function]
Package:LISP

Returns T if X is a complex number; NIL otherwise.

`BOOLE-AND` [Constant]
Package:LISP Makes BOOLE return LOGAND of INTEGER1 and INTEGER2.

`MAX` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns the greatest of its arguments.

`FLOAT-SIGN` (*float1* **&optional** *(float2 (float 1 float1))*) [Function]
Package:LISP

Returns a floating-point number with the same sign as FLOAT1 and with the same absolute value as FLOAT2.

`BOOLE-ANDC2` [Constant]
Package:LISP Makes BOOLE return LOGANDC2 of INTEGER1 and INTEGER2.

`DENOMINATOR` (*rational*) [Function]
Package:LISP

Returns the denominator of RATIONAL as an integer.

`FLOAT` (*number* **&optional** *other*) [Function]
Package:LISP

Converts a non-complex number to a floating-point number. If NUMBER is already a float, FLOAT simply returns NUMBER. Otherwise, the format of the returned float depends on OTHER; If OTHER is not provided, FLOAT returns a SINGLE-FLOAT. If OTHER is provided, the result is in the same float format as OTHER's.

`ROUND` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Rounds NUMBER/DIVISOR to nearest integer. The second returned value is the remainder.

`LOGAND` (**&rest** *integers*) [Function]
Package:LISP

Returns the bit-wise AND of its arguments.

`BOOLE-2` [Constant]
Package:LISP Makes BOOLE return INTEGER2.

`*` (**&rest** *numbers*) [Function]
Package:LISP

Returns the product of its arguments. With no args, returns 1.

`<` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if its arguments are in strictly increasing order; NIL otherwise.

`COMPLEX` (*realpart* **&optional** *(imagpart 0)*) [Function]
Package:LISP

Returns a complex number with the given real and imaginary parts.

`SINGLE-FLOAT-EPSILON` [Constant]
Package:LISP Same as LONG-FLOAT-EPSILON.

`LOGANDC2` (*integer1 integer2*) [Function]
Package:LISP

Returns the logical AND of INTEGER1 and (LOGNOT INTEGER2).

`INTEGER-LENGTH` (*integer*) [Function]
Package:LISP

Returns the number of significant bits in the absolute value of INTEGER.

`MOST-NEGATIVE-FIXNUM` [Constant]
Package:LISP The fixnum closest in value to negative infinity.

`LONG-FLOAT-NEGATIVE-EPSILON` [Constant]
Package:LISP The smallest positive long-float that satisfies (not (= (float 1 e) (- (float 1 e) e))).

`>=` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if arguments are in strictly non-increasing order; NIL otherwise.

`BOOLE-NOR` [Constant]
Package:LISP Makes BOOLE return LOGNOR of INTEGER1 and INTEGER2.

`ACOS` (*number*) [Function]
Package:LISP

Returns the arc cosine of NUMBER.

`MAKE-RANDOM-STATE` (**&optional** *(state *random-state*)*) [Function]
Package:LISP

Creates and returns a copy of the specified random state. If STATE is NIL, then the value of *RANDOM-STATE* is used. If STATE is T, then returns a random state object generated from the universal time.

`EXPT` (*base-number power-number*) [Function]
Package:LISP

Returns BASE-NUMBER raised to the power POWER-NUMBER.

`SQRT` (*number*) [Function]
Package:LISP

Returns the principal square root of NUMBER.

`SCALE-FLOAT` (*float integer*) [Function]
Package:LISP

Returns (* FLOAT (expt (float-radix FLOAT) INTEGER)).

`ACOSH` (*number*) [Function]
Package:LISP

Returns the hyperbolic arc cosine of NUMBER.

`MOST-NEGATIVE-LONG-FLOAT` [Constant]
Package:LISP The long-float closest in value to negative infinity.

`LEAST-NEGATIVE-LONG-FLOAT` [Constant]
Package:LISP The negative long-float closest in value to zero.

`FFLOOR` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Same as FLOOR, but returns a float as the first value.

`LOGNOR` (*integer1 integer2*) [Function]
Package:LISP

Returns the complement of the logical OR of INTEGER1 and INTEGER2.

`PARSE-INTEGER` (*string* **&key** *(start 0) (end (length string)) (radix 10) (junk-allowed nil)*) [Function]
Package:LISP

Parses STRING for an integer and returns it.

`+` (**&rest** *numbers*) [Function]
Package:LISP

Returns the sum of its arguments. With no args, returns 0.

`=` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if all of its arguments are numerically equal; NIL otherwise.

`NUMBERP` (*x*) [Function]
Package:LISP

Returns T if X is any kind of number; NIL otherwise.

`MOST-POSITIVE-DOUBLE-FLOAT` [Constant]
Package:LISP Same as MOST-POSITIVE-LONG-FLOAT.

`LOGTEST` (*integer1 integer2*) [Function]
Package:LISP

Returns T if LOGAND of INTEGER1 and INTEGER2 is not zero; NIL otherwise.

`RANDOM-STATE-P` (*x*) [Function]
Package:LISP

Returns T if X is a random-state object; NIL otherwise.

`LEAST-POSITIVE-DOUBLE-FLOAT` [Constant]
Package:LISP Same as LEAST-POSITIVE-LONG-FLOAT.

`FLOAT-PRECISION` (*float*) [Function]
Package:LISP

Returns the number of significant radix-B digits used to represent the significand F of the floating-point number, where B = (FLOAT-RADIX FLOAT).

`BOOLE-XOR` [Constant]
Package:LISP Makes BOOLE return LOGXOR of INTEGER1 and INTEGER2.

`DPB` (*newbyte bytespec integer*) [Function]
Package:LISP

Returns an integer computed by replacing the specified byte of INTEGER with NEWBYTE.

`ABS` (*number*) [Function]
Package:LISP

Returns the absolute value of NUMBER.

`CONJUGATE` (*number*) [Function]
Package:LISP

Returns the complex conjugate of NUMBER.

`CIS` (*radians*) [Function]
Package:LISP

Returns e raised to i*RADIANS.

`ODDP` (*integer*) [Function]
Package:LISP

Returns T if INTEGER is odd; NIL otherwise.

`RATIONALIZE` (*number*) [Function]
Package:LISP

Converts NUMBER into rational approximately and returns it.

`ISQRT` (*integer*) [Function]
Package:LISP

Returns the greatest integer less than or equal to the square root of the given non-negative integer.

`LOGXOR` (**&rest** *integers*) [Function]
Package:LISP

Returns the bit-wise EXCLUSIVE OR of its arguments.

`>` (*number* **&rest** *more-numbers*) [Function]
Package:LISP

Returns T if its arguments are in strictly decreasing order; NIL otherwise.

`LOGBITP` (*index integer*) [Function]
Package:LISP

Returns T if the INDEX-th bit of INTEGER is 1.

`DOUBLE-FLOAT-EPSILON` [Constant]
Package:LISP Same as LONG-FLOAT-EPSILON.

`LOGCOUNT` (*integer*) [Function]
Package:LISP

If INTEGER is negative, returns the number of 0 bits. Otherwise, returns the number of 1 bits.

`GCD` (**&rest** *integers*) [Function]
Package:LISP

Returns the greatest common divisor of INTEGERs.

`RATIONALP` (*x*) [Function]
Package:LISP

Returns T if X is an integer or a ratio; NIL otherwise.

`MOD` (*number divisor*) [Function]
Package:LISP

Returns the second result of (FLOOR NUMBER DIVISOR).

`MODF` (*number*) [Function]
Package:SYSTEM

Returns the integer and fractional part of a floating point number mod 1.0.

`BOOLE-ORC1` [Constant]
Package:LISP Makes BOOLE return LOGORC1 of INTEGER1 and INTEGER2.

`SINGLE-FLOAT-NEGATIVE-EPSILON` [Constant]
Package:LISP Same as LONG-FLOAT-NEGATIVE-EPSILON.

`FLOOR` (*number* **&optional** *(divisor 1)*) [Function]
Package:LISP

Returns the largest integer not larger than the NUMBER divided by DIVISOR. The second returned value is (- NUMBER (* first-value DIVISOR)).

`PLUSP` (*number*) [Function]
Package:LISP

Returns T if NUMBER > 0; NIL otherwise.

`FLOAT-DIGITS` (*float*) [Function]
Package:LISP

Returns the number of radix-B digits used to represent the significand F of the floating-point number, where B = (FLOAT-RADIX FLOAT).

`RANDOM` (*number* **&optional** *(state *random-state*)*) [Function]
Package:LISP

Generates a uniformly distributed pseudo-random number between zero (inclusive) and NUMBER (exclusive), by using the random state object STATE.

2 Sequences and Arrays and Hash Tables

`VECTOR` (**&rest** *objects*) [Function]
Package:LISP

Constructs a Simple-Vector from the given objects.

`SUBSEQ` (*sequence start* **&optional** *(end (length sequence))*) [Function]
Package:LISP

Returns a copy of a subsequence of SEQUENCE between START (inclusive) and END (exclusive).

`COPY-SEQ` (*sequence*) [Function]
Package:LISP

Returns a copy of SEQUENCE.

`POSITION` (*item sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the index of the first element in SEQUENCE that satisfies TEST with ITEM; NIL if no such element exists.

`ARRAY-RANK` (*array*) [Function]
Package:LISP

Returns the number of dimensions of ARRAY.

`SBIT` (*simple-bit-array* **&rest** *subscripts*) [Function]
Package:LISP

Returns the bit from SIMPLE-BIT-ARRAY at SUBSCRIPTS.

`STRING-CAPITALIZE` (*string* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns a copy of STRING with the first character of each word converted to upper-case, and remaining characters in the word converted to lower case.

`NSUBSTITUTE-IF-NOT` (*new test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements

except that all elements not satisfying TEST are replaced with NEWITEM. SEQUENCE may be destroyed.

`FIND-IF` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the index of the first element in SEQUENCE that satisfies TEST; NIL if no such element exists.

`BIT-EQV` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical EQV on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`STRING<` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

If STRING1 is lexicographically less than STRING2, then returns the longest common prefix of the strings. Otherwise, returns NIL.

`REVERSE` (*sequence*) [Function]
Package:LISP

Returns a new sequence containing the same elements as SEQUENCE but in reverse order.

`NSTRING-UPCASE` (*string* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns STRING with all lower case characters converted to uppercase.

`STRING>=` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

If STRING1 is lexicographically greater than or equal to STRING2, then returns the longest common prefix of the strings. Otherwise, returns NIL.

`ARRAY-ROW-MAJOR-INDEX` (*array* **&rest** *subscripts*) [Function]
Package:LISP

Returns the index into the data vector of ARRAY for the element of ARRAY specified by SUBSCRIPTS.

`ARRAY-DIMENSION` (*array axis-number*) [Function]
Package:LISP

Returns the length of AXIS-NUMBER of ARRAY.

`FIND` (*item sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the first element in SEQUENCE satisfying TEST with ITEM; NIL if no such element exists.

`STRING-NOT-EQUAL` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Similar to STRING=, but ignores cases.

`STRING-RIGHT-TRIM` (*char-bag string*) [Function]
Package:LISP

Returns a copy of STRING with the characters in CHAR-BAG removed from the right end.

`DELETE-IF-NOT` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence formed by destructively removing the elements not satisfying TEST from SEQUENCE.

`REMOVE-IF-NOT` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a copy of SEQUENCE with elements not satisfying TEST removed.

`STRING=` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Returns T if the two strings are character-wise CHAR=; NIL otherwise.

`NSUBSTITUTE-IF` (*new test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements except that all elements satisfying TEST are replaced with NEWITEM. SEQUENCE may be destroyed.

`SOME` (*predicate sequence* **&rest** *more-sequences*) [Function]
Package:LISP

Returns T if at least one of the elements in SEQUENCEs satisfies PREDICATE; NIL otherwise.

`MAKE-STRING` (*size* **&key** *(initial-element #\Space)*) [Function]
Package:LISP

Creates and returns a new string of SIZE length whose elements are all INITIAL-ELEMENT.

`NSUBSTITUTE` (*newitem olditem sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements except that OLDITEMs are replaced with NEWITEM. SEQUENCE may be destroyed.

`STRING-EQUAL` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Given two strings (string1 and string2), and optional integers start1, start2, end1 and end2, compares characters in string1 to characters in string2 (using char-equal).

`STRING-NOT-GREATERP` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Similar to STRING<=, but ignores cases.

`STRING>` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

If STRING1 is lexicographically greater than STRING2, then returns the longest common prefix of the strings. Otherwise, returns NIL.

`STRINGP` (*x*) [Function]
Package:LISP

Returns T if X is a string; NIL otherwise.

`DELETE-IF` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence formed by removing the elements satisfying TEST destructively from SEQUENCE.

`SIMPLE-STRING-P` (*x*) [Function]
Package:LISP

Returns T if X is a simple string; NIL otherwise.

`REMOVE-IF` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a copy of SEQUENCE with elements satisfying TEST removed.

`HASH-TABLE-COUNT` (*hash-table*) [Function]
Package:LISP

Returns the number of entries in the given Hash-Table.

`ARRAY-DIMENSIONS` (*array*) [Function]
Package:LISP

Returns a list whose elements are the dimensions of ARRAY

`SUBSTITUTE-IF-NOT` (*new test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements except that all elements not satisfying TEST are replaced with NEWITEM.

`ADJUSTABLE-ARRAY-P` (*array*) [Function]
Package:LISP

Returns T if ARRAY is adjustable; NIL otherwise.

`SVREF` (*simple-vector index*) [Function]
Package:LISP

Returns the INDEX-th element of SIMPLE-VECTOR.

`VECTOR-PUSH-EXTEND` (*new-element vector* **&optional** *(extension (length vector))*) [Function]
Package:LISP

Similar to VECTOR-PUSH except that, if the fill pointer gets too large, extends VECTOR rather then simply returns NIL.

`DELETE` (*item sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence formed by removing the specified ITEM destructively from SEQUENCE.

`REMOVE` (*item sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a copy of SEQUENCE with ITEM removed.

`STRING` (*x*) [Function]
Package:LISP

Coerces X into a string. If X is a string, then returns X itself. If X is a symbol, then returns X's print name. If X is a character, then returns a one element string containing that character. Signals an error if X cannot be coerced into a string.

`STRING-UPCASE` (*string* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns a copy of STRING with all lower case characters converted to uppercase.

`GETHASH` (*key hash-table* **&optional** *(default nil)*) [Function]
Package:LISP

Finds the entry in HASH-TABLE whose key is KEY and returns the associated value and T, as multiple values. Returns DEFAULT and NIL if there is no such entry.

`MAKE-HASH-TABLE` (**&key** *(test 'eql) (size 1024) (rehash-size 1.5) (rehash-threshold 0.7)*) [Function]
Package:LISP

Creates and returns a hash table.

`STRING/=` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Returns NIL if STRING1 and STRING2 are character-wise CHAR=. Otherwise, returns the index to the longest common prefix of the strings.

`STRING-GREATERP` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]

Package:LISP

Similar to STRING>, but ignores cases.

`ELT` (*sequence index*) [Function]

Package:LISP

Returns the INDEX-th element of SEQUENCE.

`MAKE-ARRAY` (*dimensions* **&key** *(element-type t) initial-element (initial-contents nil) (adjustable nil) (fill-pointer nil) (displaced-to nil) (displaced-index-offset 0) static*) [Function]

Package:LISP

Creates an array of the specified DIMENSIONS. The default for INITIAL- ELEMENT depends on ELEMENT-TYPE. MAKE-ARRAY will always try to find the 'best' array to accommodate the element-type specified. For example on a SUN element-type (mod 1) –> bit (integer 0 10) –> unsigned-char (integer -3 10) –> signed-char si::best-array-element-type is the function doing this. It is also used by the compiler, for coercing array element types. If you are going to declare an array you should use the same element type as was used in making it. eg (setq my-array (make-array 4 :element-type '(integer 0 10))) (the (array (integer 0 10)) my-array) When wanting to optimize references to an array you need to declare the array eg: (the (array (integer -3 10)) my-array) if ar were constructed using the (integer -3 10) element-type. You could of course have used signed-char, but since the ranges may be implementation dependent it is better to use -3 10 range. MAKE-ARRAY needs to do some calculation with the element-type if you don't provide a primitive data-type. One way of doing this in a machine independent fashion:

(defvar *my-elt-type* #. (array-element-type (make-array 1 :element-type '(integer -3 10))))

Then calls to (make-array n :element-type *my-elt-type*) will not have to go through a type inclusion computation. The keyword STATIC (GCL specific) if non nil, will cause the array body to be non relocatable.

`NSTRING-DOWNCASE` (*string* **&key** *(start 0) (end (length string))*) [Function]

Package:LISP Returns STRING with all upper case characters converted to lowercase.

`ARRAY-IN-BOUNDS-P` (*array* **&rest** *subscripts*) [Function]

Package:LISP Returns T if SUBSCRIPTS are valid subscripts for ARRAY; NIL otherwise.

`SORT` (*sequence predicate* **&key** *(key #'identity)*) [Function]

Package:LISP Destructively sorts SEQUENCE. PREDICATE should return non-NIL if its first argument is to precede its second argument.

`HASH-TABLE-P` (*x*) [Function]

Package:LISP

Returns T if X is a hash table object; NIL otherwise.

`COUNT-IF-NOT` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the number of elements in SEQUENCE not satisfying TEST.

`FILL-POINTER` (*vector*) [Function]
Package:LISP

Returns the fill pointer of VECTOR.

`ARRAYP` (*x*) [Function]
Package:LISP

Returns T if X is an array; NIL otherwise.

`REPLACE` (*sequence1 sequence2* **&key** *(start1 0) (end1 (length sequence1)) (start2 0) (end2 (length sequence2))*) [Function]
Package:LISP

Destructively modifies SEQUENCE1 by copying successive elements into it from SEQUENCE2.

`BIT-XOR` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical XOR on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`CLRHASH` (*hash-table*) [Function]
Package:LISP

Removes all entries of HASH-TABLE and returns the hash table itself.

`SUBSTITUTE-IF` (*newitem test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements except that all elements satisfying TEST are replaced with NEWITEM.

`MISMATCH` (*sequence1 sequence2* **&key** *(from-end nil) (test #'eql) test-not (start1 0) (start2 0) (end1 (length sequence1)) (end2 (length sequence2)) (key #'identity)*) [Function]
Package:LISP

The specified subsequences of SEQUENCE1 and SEQUENCE2 are compared element-wise. If they are of equal length and match in every element, the result is NIL. Otherwise, the result is a non-negative integer, the index within SEQUENCE1 of the leftmost position at which they fail to match; or, if one is shorter than and a matching prefix of the other, the index within SEQUENCE1 beyond the last position tested is returned.

`ARRAY-TOTAL-SIZE-LIMIT` [Constant]
Package:LISP The exclusive upper bound on the total number of elements of an array.

`VECTOR-POP` (*vector*) [Function]
Package:LISP

Attempts to decrease the fill-pointer of VECTOR by 1 and returns the element pointed to by the new fill pointer. Signals an error if the old value of the fill pointer is 0.

`SUBSTITUTE` (*newitem olditem sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (count most-positive-fixnum) (key #'identity)*) [Function]
Package:LISP

Returns a sequence of the same kind as SEQUENCE with the same elements except that OLDITEMs are replaced with NEWITEM.

`ARRAY-HAS-FILL-POINTER-P` (*array*) [Function]
Package:LISP

Returns T if ARRAY has a fill pointer; NIL otherwise.

`CONCATENATE` (*result-type* **&rest** *sequences*) [Function]
Package:LISP

Returns a new sequence of the specified RESULT-TYPE, consisting of all elements in SEQUENCEs.

`VECTOR-PUSH` (*new-element vector*) [Function]
Package:LISP

Attempts to set the element of ARRAY designated by its fill pointer to NEW-ELEMENT and increments the fill pointer by one. Returns NIL if the fill pointer is too large. Otherwise, returns the new fill pointer value.

`STRING-TRIM` (*char-bag string*) [Function]
Package:LISP

Returns a copy of STRING with the characters in CHAR-BAG removed from both ends.

`ARRAY-ELEMENT-TYPE` (*array*) [Function]
Package:LISP

Returns the type of the elements of ARRAY

`NOTANY` (*predicate sequence* **&rest** *more-sequences*) [Function]
Package:LISP

Returns T if none of the elements in SEQUENCEs satisfies PREDICATE; NIL otherwise.

`BIT-NOT` (*bit-array* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical NOT in the elements of BIT-ARRAY. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`BIT-ORC1` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical ORC1 on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`COUNT-IF` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the number of elements in SEQUENCE satisfying TEST.

`MAP` (*result-type function sequence* **&rest** *more-sequences*) [Function]
Package:LISP

FUNCTION must take as many arguments as there are sequences provided. The result is a sequence such that the i-th element is the result of applying FUNCTION to the i-th elements of the SEQUENCEs.

`ARRAY-RANK-LIMIT` [Constant]
Package:LISP The exclusive upper bound on the rank of an array.

`COUNT` (*item sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the number of elements in SEQUENCE satisfying TEST with ITEM.

`BIT-VECTOR-P` (*x*) [Function]
Package:LISP

Returns T if X is a bit vector; NIL otherwise.

`NSTRING-CAPITALIZE` (*string* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns STRING with the first character of each word converted to upper-case, and remaining characters in the word converted to lower case.

`ADJUST-ARRAY` (*array dimensions* **&key** *(element-type (array-element-type array)) initial-element (initial-contents nil) (fill-pointer nil) (displaced-to nil) (displaced-index-offset 0)*) [Function]
Package:LISP

Adjusts the dimensions of ARRAY to the given DIMENSIONS. The default value of INITIAL-ELEMENT depends on ELEMENT-TYPE.

`SEARCH` (*sequence1 sequence2* **&key** *(from-end nil) (test #'eql) test-not (start1 0) (start2 0) (end1 (length sequence1)) (end2 (length sequence2)) (key #'identity)*) [Function]
Package:LISP

A search is conducted for the first subsequence of SEQUENCE2 which element-wise matches SEQUENCE1. If there is such a subsequence in SEQUENCE2, the index of the its leftmost element is returned; otherwise, NIL is returned.

`SIMPLE-BIT-VECTOR-P` (*x*) [Function]
Package:LISP

Returns T if X is a simple bit-vector; NIL otherwise.

`MAKE-SEQUENCE` (*type length* **&key** *initial-element*) [Function]
Package:LISP

Returns a sequence of the given TYPE and LENGTH, with elements initialized to INITIAL-ELEMENT. The default value of INITIAL-ELEMENT depends on TYPE.

`BIT-ORC2` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical ORC2 on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`NREVERSE` (*sequence*) [Function]
Package:LISP

Returns a sequence of the same elements as SEQUENCE but in reverse order. SEQUENCE may be destroyed.

`ARRAY-DIMENSION-LIMIT` [Constant]
Package:LISP The exclusive upper bound of the array dimension.

`NOTEVERY` (*predicate sequence* **&rest** *more-sequences*) [Function]
Package:LISP

Returns T if at least one of the elements in SEQUENCEs does not satisfy PREDICATE; NIL otherwise.

`POSITION-IF-NOT` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]
Package:LISP

Returns the index of the first element in SEQUENCE that does not satisfy TEST; NIL if no such element exists.

`STRING-DOWNCASE` (*string* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns a copy of STRING with all upper case characters converted to lowercase.

`BIT` (*bit-array* **&rest** *subscripts*) [Function]
Package:LISP

Returns the bit from BIT-ARRAY at SUBSCRIPTS.

`STRING-NOT-LESSP` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]
Package:LISP

Similar to STRING>=, but ignores cases.

`CHAR` (*string index*) [Function]
Package:LISP

Returns the INDEX-th character in STRING.

`AREF` (*array* **&rest** *subscripts*) [Function]
Package:LISP

Returns the element of ARRAY specified by SUBSCRIPTS.

`FILL` (*sequence item* **&key** *(start 0) (end (length sequence))*) [Function]
Package:LISP

Replaces the specified elements of SEQUENCE all with ITEM.

`STABLE-SORT` (*sequence predicate* **&key** *(key #'identity)*) [Function]
Package:LISP

Destructively sorts SEQUENCE. PREDICATE should return non-NIL if its first argument is to precede its second argument.

`BIT-IOR` (*bit-array1 bit-array2* **&optional** *(result-bit-array nil)*) [Function]
Package:LISP

Performs a bit-wise logical IOR on the elements of BIT-ARRAY1 and BIT-ARRAY2. Puts the results into a new bit array if RESULT-BIT-ARRAY is NIL, into BIT-ARRAY1 if RESULT-BIT-ARRAY is T, or into RESULT-BIT-ARRAY otherwise.

`REMHASH` (*key hash-table*) [Function]
Package:LISP

Removes any entry for KEY in HASH-TABLE. Returns T if such an entry existed; NIL otherwise.

`VECTORP` (*x*) [Function]
Package:LISP

Returns T if X is a vector; NIL otherwise.

`STRING<=` (*string1 string2* **&key** *(start1 0) (end1 (length string1))* [Function]
(start2 0) (end2 (length string2)))
Package:LISP

If STRING1 is lexicographically less than or equal to STRING2, then returns the longest common prefix of the two strings. Otherwise, returns NIL.

`SIMPLE-VECTOR-P` (*x*) [Function]
Package:LISP

Returns T if X is a simple vector; NIL otherwise.

`STRING-LEFT-TRIM` (*char-bag string*) [Function]
Package:LISP

Returns a copy of STRING with the characters in CHAR-BAG removed from the left end.

`ARRAY-TOTAL-SIZE` (*array*) [Function]

Package:LISP

Returns the total number of elements of ARRAY.

`FIND-IF-NOT` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]

Package:LISP

Returns the index of the first element in SEQUENCE that does not satisfy TEST; NIL if no such element exists.

`DELETE-DUPLICATES` (*sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (key #'identity)*) [Function]

Package:LISP

Returns a sequence formed by removing duplicated elements destructively from SEQUENCE.

`REMOVE-DUPLICATES` (*sequence* **&key** *(from-end nil) (test #'eql) test-not (start 0) (end (length sequence)) (key #'identity)*) [Function]

Package:LISP

The elements of SEQUENCE are examined, and if any two match, one is discarded. Returns the resulting sequence.

`POSITION-IF` (*test sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) (key #'identity)*) [Function]

Package:LISP

Returns the index of the first element in SEQUENCE that satisfies TEST; NIL if no such element exists.

`MERGE` (*result-type sequence1 sequence2 predicate* **&key** *(key #'identity)*) [Function]

Package:LISP

SEQUENCE1 and SEQUENCE2 are destructively merged into a sequence of type RESULT-TYPE using PREDICATE to order the elements.

`EVERY` (*predicate sequence* **&rest** *more-sequences*) [Function]

Package:LISP

Returns T if every elements of SEQUENCEs satisfy PREDICATE; NIL otherwise.

`REDUCE` (*function sequence* **&key** *(from-end nil) (start 0) (end (length sequence)) initial-value*) [Function]

Package:LISP

Combines all the elements of SEQUENCE using a binary operation FUNCTION. If INITIAL-VALUE is supplied, it is logically placed before the SEQUENCE.

`STRING-LESSP` (*string1 string2* **&key** *(start1 0) (end1 (length string1)) (start2 0) (end2 (length string2))*) [Function]

Package:LISP

Similar to STRING<, but ignores cases.

3 Characters

`NAME-CHAR` (*name*) [Function]
Package:LISP

Given an argument acceptable to string, Returns a character object whose name is NAME if one exists. Returns NIL otherwise. NAME must be an object that can be coerced to a string.

`CHAR-NAME` (*char*) [Function]
Package:LISP

Returns the name for CHAR as a string; NIL if CHAR has no name. Only #\Backspace, #\Tab, #\Newline (or #\Linefeed), #\Page, #\Return, and #\Rubout have names.

`BOTH-CASE-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is an alphabetic character; NIL otherwise. Equivalent to ALPHA-CHAR-P.

`SCHAR` (*simple-string index*) [Function]
Package:LISP

Returns the character object representing the INDEX-th character in STRING. This is faster than CHAR.

`CHAR-SUPER-BIT` [Constant]
Package:LISP The bit that indicates a super character.

`CHAR-FONT-LIMIT` [Constant]
Package:LISP The upper exclusive bound on values produced by CHAR-FONT.

`CHAR-DOWNCASE` (*char*) [Function]
Package:LISP

Returns the lower-case equivalent of CHAR, if any. If not, simply returns CHAR.

`STRING-CHAR-P` (*char*) [Function]
Package:LISP

Returns T if CHAR can be stored in a string. In GCL, this function always returns T since any character in GCL can be stored in a string.

`CHAR-NOT-LESSP` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly non-increasing order; NIL otherwise. For a lower-case character, the code of its upper-case equivalent is used.

`DISASSEMBLE` (*thing*) [Function]
Package:LISP

Compiles the form specified by THING and prints the intermediate C language code for that form. But does NOT install the result of compilation. If THING is a symbol

that names a not-yet-compiled function, the function definition is disassembled. If THING is a lambda expression, it is disassembled as a function definition. Otherwise, THING itself is disassembled as a top-level form.

`LOWER-CASE-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is a lower-case character; NIL otherwise.

`CHAR<=` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly non-decreasing order; NIL otherwise.

`CHAR-HYPER-BIT` [Constant]
Package:LISP The bit that indicates a hyper character.

`CODE-CHAR` (*code* **&optional** *(bits 0) (font 0)*) [Function]
Package:LISP

Returns a character object with the specified code, if any. If not, returns NIL.

`CHAR-CODE` (*char*) [Function]
Package:LISP

Returns the code attribute of CHAR.

`CHAR-CONTROL-BIT` [Constant]
Package:LISP The bit that indicates a control character.

`CHAR-LESSP` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly increasing order; NIL otherwise. For a lower-case character, the code of its upper-case equivalent is used.

`CHAR-FONT` (*char*) [Function]
Package:LISP

Returns the font attribute of CHAR.

`CHAR<` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly increasing order; NIL otherwise.

`CHAR>=` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly non-increasing order; NIL otherwise.

`CHAR-META-BIT` [Constant]
Package:LISP The bit that indicates a meta character.

`GRAPHIC-CHAR-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is a printing character, i.e., #\Space through #\~; NIL otherwise.

`CHAR-NOT-EQUAL` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if no two of CHARs are the same character; NIL otherwise. Upper case character and its lower case equivalent are regarded the same.

`CHAR-BITS-LIMIT` [Constant]
Package:LISP The upper exclusive bound on values produced by CHAR-BITS.

`CHARACTERP` (*x*) [Function]
Package:LISP

Returns T if X is a character; NIL otherwise.

`CHAR=` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if all CHARs are the same character; NIL otherwise.

`ALPHA-CHAR-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is an alphabetic character, A-Z or a-z; NIL otherwise.

`UPPER-CASE-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is an upper-case character; NIL otherwise.

`CHAR-BIT` (*char name*) [Function]
Package:LISP

Returns T if the named bit is on in the character CHAR; NIL otherwise. In GCL, this function always returns NIL.

`MAKE-CHAR` (*char* **&optional** *(bits 0) (font 0)*) [Function]
Package:LISP

Returns a character object with the same code attribute as CHAR and with the specified BITS and FONT attributes.

`CHARACTER` (*x*) [Function]
Package:LISP

Coerces X into a character object if possible.

`CHAR-EQUAL` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if all of its arguments are the same character; NIL otherwise. Upper case character and its lower case equivalent are regarded the same.

`CHAR-NOT-GREATERP` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly non-decreasing order; NIL otherwise. For a lower-case character, the code of its upper-case equivalent is used.

`CHAR>` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly decreasing order; NIL otherwise.

`STANDARD-CHAR-P` (*char*) [Function]
Package:LISP

Returns T if CHAR is a standard character, i.e., one of the 95 ASCII printing characters #\Space to #\~ and #Newline; NIL otherwise.

`CHAR-UPCASE` (*char*) [Function]
Package:LISP

Returns the upper-case equivalent of CHAR, if any. If not, simply returns CHAR.

`DIGIT-CHAR-P` (*char* **&optional** *(radix 10)*) [Function]
Package:LISP

If CHAR represents a digit in RADIX, then returns the weight as an integer. Otherwise, returns nil.

`CHAR/=` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if no two of CHARs are the same character; NIL otherwise.

`CHAR-GREATERP` (*char* **&rest** *more-chars*) [Function]
Package:LISP

Returns T if the codes of CHARs are in strictly decreasing order; NIL otherwise. For a lower-case character, the code of its upper-case equivalent is used.

`ALPHANUMERICP` (*char*) [Function]
Package:LISP

Returns T if CHAR is either numeric or alphabetic; NIL otherwise.

`CHAR-BITS` (*char*) [Function]
Package:LISP

Returns the bits attribute (which is always 0 in GCL) of CHAR.

`DIGIT-CHAR` (*digit* **&optional** *(radix 10) (font 0)*) [Function]
Package:LISP

Returns a character object that represents the DIGIT in the specified RADIX. Returns NIL if no such character exists.

`SET-CHAR-BIT` (*char name newvalue*) [Function]
Package:LISP

Returns a character just like CHAR except that the named bit is set or cleared, according to whether NEWVALUE is non-NIL or NIL. This function is useless in GCL.

4 Lists

`NINTERSECTION` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns the intersection of LIST1 and LIST2. LIST1 may be destroyed.

`RASSOC-IF` (*predicate alist*) [Function]
Package:LISP

Returns the first cons in ALIST whose cdr satisfies PREDICATE.

`MAKE-LIST` (*size* **&key** *(initial-element nil)*) [Function]
Package:LISP

Creates and returns a list containing SIZE elements, each of which is initialized to INITIAL-ELEMENT.

`NTH` (*n list*) [Function]
Package:LISP

Returns the N-th element of LIST, where the car of LIST is the zeroth element.

`CAAR` (*x*) [Function]
Package:LISP

Equivalent to (CAR (CAR X)).

`NULL` (*x*) [Function]
Package:LISP

Returns T if X is NIL; NIL otherwise.

`FIFTH` (*x*) [Function]
Package:LISP

Equivalent to (CAR (CDDDDR X)).

`NCONC` (**&rest** *lists*) [Function]
Package:LISP

Concatenates LISTs by destructively modifying them.

`TAILP` (*sublist list*) [Function]
Package:LISP

Returns T if SUBLIST is one of the conses in LIST; NIL otherwise.

`CONSP` (*x*) [Function]
Package:LISP

Returns T if X is a cons; NIL otherwise.

`TENTH` (*x*) [Function]
Package:LISP

Equivalent to (CADR (CDDDDR (CDDDDR X))).

`LISTP` (*x*) [Function]
Package:LISP

Returns T if X is either a cons or NIL; NIL otherwise.

`MAPCAN` (*fun list* **&rest** *more-lists*) [Function]
Package:LISP

Applies FUN to successive cars of LISTs, NCONCs the results, and returns it.

`EIGHTH` (*x*) [Function]
Package:LISP

Equivalent to (CADDDR (CDDDDR X)).

`LENGTH` (*sequence*) [Function]
Package:LISP

Returns the length of SEQUENCE.

`RASSOC` (*item alist* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns the first cons in ALIST whose cdr is equal to ITEM.

`NSUBST-IF-NOT` (*new test tree* **&key** *(key #'identity)*) [Function]
Package:LISP

Substitutes NEW for subtrees of TREE that do not satisfy TEST.

`NBUTLAST` (*list* **&optional** *(n 1)*) [Function]
Package:LISP

Changes the cdr of the N+1 th cons from the end of the list LIST to NIL. Returns the whole list.

`CDR` (*list*) [Function]
Package:LISP

Returns the cdr of LIST. Returns NIL if LIST is NIL.

`MAPC` (*fun list* **&rest** *more-lists*) [Function]
Package:LISP

Applies FUN to successive cars of LISTs. Returns the first LIST.

`MAPL` (*fun list* **&rest** *more-lists*) [Function]
Package:LISP

Applies FUN to successive cdrs of LISTs. Returns the first LIST.

`CONS` (*x y*) [Function]
Package:LISP

Returns a new cons whose car and cdr are X and Y, respectively.

`LIST` (**&rest** *args*) [Function]
Package:LISP

Returns a list of its arguments

`THIRD` (*x*) [Function]
Package:LISP
Equivalent to (CADDR X).

`CDDAAR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CDR (CAR (CAR X)))).

`CDADAR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR (CDR (CAR X)))).

`CDAADR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR (CAR (CDR X)))).

`CADDAR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR (CDR (CAR X)))).

`CADADR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR (CAR (CDR X)))).

`CAADDR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CAR (CDR (CDR X)))).

`NTHCDR` (*n list*) [Function]
Package:LISP
Returns the result of performing the CDR operation N times on LIST.

`PAIRLIS` (*keys data* **&optional** *(alist nil)*) [Function]
Package:LISP
Constructs an association list from KEYS and DATA adding to ALIST.

`SEVENTH` (*x*) [Function]
Package:LISP
Equivalent to (CADDR (CDDDDR X)).

`SUBSETP` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Returns T if every element of LIST1 appears in LIST2; NIL otherwise.

`NSUBST-IF` (*new test tree* **&key** *(key #'identity)*) [Function]
Package:LISP
Substitutes NEW for subtrees of TREE that satisfy TEST.

`COPY-LIST` (*list*) [Function]
Package:LISP
Returns a new copy of LIST.

`LAST` (*list*) [Function]
Package:LISP
Returns the last cons in LIST

`CAAAR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CAR (CAR X))).

`LIST-LENGTH` (*list*) [Function]
Package:LISP
Returns the length of LIST, or NIL if LIST is circular.

`CDDDR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CDR (CDR X))).

`INTERSECTION` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Returns the intersection of List1 and List2.

`NSUBST` (*new old tree* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Substitutes NEW for subtrees in TREE that match OLD.

`REVAPPEND` (*x y*) [Function]
Package:LISP
Equivalent to (APPEND (REVERSE X) Y)

`CDAR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR X)).

`CADR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR X)).

`REST` (*x*) [Function]
Package:LISP
Equivalent to (CDR X).

`NSET-EXCLUSIVE-OR` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Returns a list with elements which appear but once in LIST1 and LIST2.

`ACONS` (*key datum alist*) [Function]
Package:LISP
Constructs a new alist by adding the pair (KEY . DATUM) to ALIST.

`SUBST-IF-NOT` (*new test tree* **&key** *(key #'identity)*) [Function]
Package:LISP
Substitutes NEW for subtrees of TREE that do not satisfy TEST.

`RPLACA` (*x y*) [Function]
Package:LISP
Replaces the car of X with Y, and returns the modified X.

`SECOND` (*x*) [Function]
Package:LISP
Equivalent to (CADR X).

`NUNION` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Returns the union of LIST1 and LIST2. LIST1 and/or LIST2 may be destroyed.

`BUTLAST` (*list* **&optional** *(n 1)*) [Function]
Package:LISP
Creates and returns a list with the same elements as LIST but without the last N elements.

`COPY-ALIST` (*alist*) [Function]
Package:LISP Returns a new copy of ALIST.

`SIXTH` (*x*) [Function]
Package:LISP Equivalent to (CADR (CDDDDR X)).

`CAAAAR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CAR (CAR (CAR X)))).

`CDDDAR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CDR (CDR (CAR X)))).

`CDDADR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CDR (CAR (CDR X)))).

`CDADDR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR (CDR (CDR X)))).

`CADDDR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR (CDR (CDR X)))).

`FOURTH` (*x*) [Function]
Package:LISP

Equivalent to (CADDDR X).

`NSUBLIS` (*alist tree* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Substitutes from ALIST for subtrees of TREE.

`SUBST-IF` (*new test tree* **&key** *(key #'identity)*) [Function]
Package:LISP

Substitutes NEW for subtrees of TREE that satisfy TEST.

`NSET-DIFFERENCE` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns a list of elements of LIST1 that do not appear in LIST2. LIST1 may be destroyed.

`POP` [Special Form]
Package:LISP

Syntax:

```
(pop place)
```

Pops one item off the front of the list in PLACE and returns it.

`PUSH` [Special Form]
Package:LISP

Syntax:

```
(push item place)
```

Conses ITEM onto the list in PLACE, and returns the new list.

`CDAAR` (*x*) [Function]
Package:LISP

Equivalent to (CDR (CAR (CAR X))).

`CADAR` (*x*) [Function]
Package:LISP

Equivalent to (CAR (CDR (CAR X))).

`CAADR` (*x*) [Function]
Package:LISP

Equivalent to (CAR (CAR (CDR X))).

`FIRST` (*x*) [Function]
Package:LISP

Equivalent to (CAR X).

`SUBST` (*new old tree* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Substitutes NEW for subtrees of TREE that match OLD.

`ADJOIN` (*item list* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Adds ITEM to LIST unless ITEM is already a member of LIST.

`MAPCON` (*fun list* **&rest** *more-lists*) [Function]
Package:LISP

Applies FUN to successive cdrs of LISTs, NCONCs the results, and returns it.

`PUSHNEW` [Macro]
Package:LISP

Syntax:

```
(pushnew item place {keyword value}*)
```

If ITEM is already in the list stored in PLACE, does nothing. Else, conses ITEM onto the list. Returns NIL. If no KEYWORDs are supplied, each element in the list is compared with ITEM by EQL, but the comparison can be controlled by supplying keywords :TEST, :TEST-NOT, and/or :KEY.

`SET-EXCLUSIVE-OR` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns a list of elements appearing exactly once in LIST1 and LIST2.

`TREE-EQUAL` (*x y* **&key** *(test #'eql) test-not*) [Function]
Package:LISP

Returns T if X and Y are isomorphic trees with identical leaves.

`CDDR` (*x*) [Function]
Package:LISP

Equivalent to (CDR (CDR X)).

`GETF` (*place indicator* **&optional** *(default nil)*) [Function]
Package:LISP

Searches the property list stored in Place for an indicator EQ to Indicator. If one is found, the corresponding value is returned, else the Default is returned.

`LDIFF` (*list sublist*) [Function]
Package:LISP

Returns a new list, whose elements are those of LIST that appear before SUBLIST. If SUBLIST is not a tail of LIST, a copy of LIST is returned.

`UNION` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns the union of LIST1 and LIST2.

`ASSOC-IF-NOT` (*test alist*) [Function]
Package:LISP
Returns the first pair in ALIST whose car does not satisfy TEST.

`RPLACD` (*x y*) [Function]
Package:LISP
Replaces the cdr of X with Y, and returns the modified X.

`MEMBER-IF-NOT` (*test list* **&key** *(key #'identity)*) [Function]
Package:LISP
Returns the tail of LIST beginning with the first element not satisfying TEST.

`CAR` (*list*) [Function]
Package:LISP
Returns the car of LIST. Returns NIL if LIST is NIL.

`ENDP` (*x*) [Function]
Package:LISP
Returns T if X is NIL. Returns NIL if X is a cons. Otherwise, signals an error.

`LIST*` (*arg* **&rest** *others*) [Function]
Package:LISP
Returns a list of its arguments with the last cons being a dotted pair of the next to the last argument and the last argument.

`NINTH` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDDDDR (CDDDDR X))).

`CDAAAR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR (CAR (CAR X)))).

`CADAAR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR (CAR (CAR X)))).

`CAADAR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CAR (CDR (CAR X)))).

`CAAADR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CAR (CAR (CDR X)))).

`CDDDDR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CDR (CDR (CDR X)))).

`SUBLIS` (*alist tree* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Substitutes from ALIST for subtrees of TREE nondestructively.

`RASSOC-IF-NOT` (*predicate alist*) [Function]
Package:LISP

Returns the first cons in ALIST whose cdr does not satisfy PREDICATE.

`NRECONC` (*x y*) [Function]
Package:LISP

Equivalent to (NCONC (NREVERSE X) Y).

`MAPLIST` (*fun list* **&rest** *more-lists*) [Function]
Package:LISP

Applies FUN to successive cdrs of LISTs and returns the results as a list.

`SET-DIFFERENCE` (*list1 list2* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP

Returns a list of elements of LIST1 that do not appear in LIST2.

`ASSOC-IF` (*test alist*) [Function]
Package:LISP

Returns the first pair in ALIST whose car satisfies TEST.

`GET-PROPERTIES` (*place indicator-list*) [Function]
Package:LISP

Looks for the elements of INDICATOR-LIST in the property list stored in PLACE. If found, returns the indicator, the value, and T as multiple-values. If not, returns NILs as its three values.

`MEMBER-IF` (*test list* **&key** *(key #'identity)*) [Function]
Package:LISP

Returns the tail of LIST beginning with the first element satisfying TEST.

`COPY-TREE` (*object*) [Function]
Package:LISP

Recursively copies conses in OBJECT and returns the result.

`ATOM` (*x*) [Function]
Package:LISP

Returns T if X is not a cons; NIL otherwise.

`CDDAR` (*x*) [Function]
Package:LISP

Equivalent to (CDR (CDR (CAR X))).

`CDADR` (*x*) [Function]
Package:LISP
Equivalent to (CDR (CAR (CDR X))).

`CADDR` (*x*) [Function]
Package:LISP
Equivalent to (CAR (CDR (CDR X))).

`ASSOC` (*item alist* **&key** *(test #'eql) test-not*) [Function]
Package:LISP
Returns the first pair in ALIST whose car is equal (in the sense of TEST) to ITEM.

`APPEND` (**&rest** *lists*) [Function]
Package:LISP
Constructs a new list by concatenating its arguments.

`MEMBER` (*item list* **&key** *(test #'eql) test-not (key #'identity)*) [Function]
Package:LISP
Returns the tail of LIST beginning with the first ITEM.

5 Streams and Reading

`MAKE-ECHO-STREAM` (*input-stream output-stream*) [Function]
Package:LISP

Returns a bidirectional stream which gets its input from INPUT-STREAM and sends its output to OUTPUT-STREAM. In addition, all input is echoed to OUTPUT-STREAM.

`*READTABLE*` [Variable]
Package:LISP The current readtable.

`LOAD` (*filename* **&key** *(verbose *load-verbose*) (print nil) (if-does-not-exist :error)*) [Function]
Package:LISP

Loads the file named by FILENAME into GCL.

`OPEN` (*filename* **&key** *(direction :input) (element-type 'string-char) (if-exists :error) (if-does-not-exist :error)*) [Function]
Package:LISP

Opens the file specified by FILENAME, which may be a string, a pathname, or a stream. Returns a stream for the open file. DIRECTION is :INPUT, :OUTPUT, :IO or :PROBE. ELEMENT-TYPE is STRING-CHAR, (UNSIGNED-BYTE n), UNSIGNED-BYTE, (SIGNED-BYTE n), SIGNED-BYTE, CHARACTER, BIT, (MOD n), or :DEFAULT. IF-EXISTS is :ERROR, :NEW-VERSION, :RENAME, :RENAME-AND-DELETE, :OVERWRITE, :APPEND, :SUPERSEDE, or NIL. IF-DOES-NOT-EXIST is :ERROR, :CREATE, or NIL.

If FILENAME begins with a vertical pipe sign: '|' then the resulting stream is actually a one way pipe. It will be open for reading or writing depending on the direction given. The rest of FILENAME in this case is passed to the /bin/sh command. See the posix description of popen for more details.

```
(setq pipe (open "| wc < /tmp/jim"))
(format t "File has ~%d lines" (read pipe))
(close pipe)
```

`*PRINT-BASE*` [Variable]
Package:LISP The radix in which the GCL printer prints integers and rationals. The value must be an integer from 2 to 36, inclusive.

`MAKE-STRING-INPUT-STREAM` (*string* **&optional** *(start 0) (end (length string))*) [Function]
Package:LISP

Returns an input stream which will supply the characters of String between Start and End in order.

`PPRINT` (*object* **&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Pretty-prints OBJECT. Returns OBJECT. Equivalent to (WRITE :STREAM STREAM :PRETTY T) The SI:PRETTY-PRINT-FORMAT property N (which must be a non-negative integer) of a symbol SYMBOL controls the pretty-printing of form (SYMBOL f1 ... fN fN+1 ... fM) in such a way that the subforms fN+1, ..., fM are regarded as the 'body' of the entire form. For instance, the property value of 2 is initially given to the symbol DO.

`*READ-DEFAULT-FLOAT-FORMAT*` [Variable]

Package:LISP The floating-point format the GCL reader uses when reading floating-point numbers that have no exponent marker or have e or E for an exponent marker. Must be one of SHORT-FLOAT, SINGLE-FLOAT, DOUBLE-FLOAT, and LONG-FLOAT.

`READ-PRESERVING-WHITESPACE` (**&optional** *(stream *standard-input*) (eof-error-p t) (eof-value nil) (recursive-p nil)*) [Function]

Package:LISP

Reads an object from STREAM, preserving the whitespace that followed the object.

`STREAMP` (*x*) [Function]

Package:LISP

Returns T if X is a stream object; NIL otherwise.

`SET-DISPATCH-MACRO-CHARACTER` (*disp-char sub-char function* **&optional** *(readtable *readtable*)*) [Function]

Package:LISP

Causes FUNCTION to be called when the DISP-CHAR followed by SUB-CHAR is read.

`WITH-OUTPUT-TO-STRING` [Macro]

Package:LISP

Syntax:

```
(with-output-to-string (var [string]) {decl}* {form}*)
```

Binds VAR to a string output stream that puts characters into STRING, which defaults to a new string. The stream is automatically closed on exit and the string is returned.

`FILE-LENGTH` (*file-stream*) [Function]

Package:LISP

Returns the length of the specified file stream.

`*PRINT-CASE*` [Variable]

Package:LISP The case in which the GCL printer should print ordinary symbols. The value must be one of the keywords :UPCASE, :DOWNCASE, and :CAPITALIZE.

`PRINT` (*object* **&optional** *(stream *standard-output*)*) [Function]

Package:LISP

Outputs a newline character, and then prints OBJECT in the mostly readable representation. Returns OBJECT. Equivalent to (PROGN (TERPRI STREAM) (WRITE OBJECT :STREAM STREAM :ESCAPE T)).

`SET-MACRO-CHARACTER` (*char function* **&optional** *(non-terminating-p nil) (readtable *readtable*)*) [Function]
Package:LISP

Causes CHAR to be a macro character that, when seen by READ, causes FUNCTION to be called.

`FORCE-OUTPUT` (**&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Attempts to force any buffered output to be sent.

`*PRINT-ARRAY*` [Variable]
Package:LISP Whether the GCL printer should print array elements.

`STREAM-ELEMENT-TYPE` (*stream*) [Function]
Package:LISP

Returns a type specifier for the kind of object returned by STREAM.

`WRITE-BYTE` (*integer stream*) [Function]
Package:LISP

Outputs INTEGER to the binary stream STREAM. Returns INTEGER.

`MAKE-CONCATENATED-STREAM` (**&rest** *streams*) [Function]
Package:LISP

Returns a stream which takes its input from each of the STREAMs in turn, going on to the next at end of stream.

`PRIN1` (*object* **&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Prints OBJECT in the mostly readable representation. Returns OBJECT. Equivalent to (WRITE OBJECT :STREAM STREAM :ESCAPE T).

`PRINC` (*object* **&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Prints OBJECT without escape characters. Returns OBJECT. Equivalent to (WRITE OBJECT :STREAM STREAM :ESCAPE NIL).

`CLEAR-OUTPUT` (**&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Clears the output stream STREAM.

`TERPRI` (**&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Outputs a newline character.

`FINISH-OUTPUT` (**&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Attempts to ensure that all output sent to STREAM has reached its destination, and only then returns.

`WITH-OPEN-FILE` [Macro]

Package:LISP

Syntax:

```
(with-open-file (stream filename {options}*) {decl}* {form}*)
```

Opens the file whose name is FILENAME, using OPTIONs, and binds the variable STREAM to a stream to/from the file. Then evaluates FORMs as a PROGN. The file is automatically closed on exit.

`DO` [Special Form]

Package:LISP

Syntax:

```
(do ({(var [init [step]])}*) (endtest {result}*)
          {decl}* {tag | statement}*)
```

Creates a NIL block, binds each VAR to the value of the corresponding INIT, and then executes STATEMENTs repeatedly until ENDTEST is satisfied. After each iteration, assigns to each VAR the value of the corresponding STEP. When ENDTEST is satisfied, evaluates RESULTs as a PROGN and returns the value(s) of the last RESULT (or NIL if no RESULTs are supplied). Performs variable bindings and assignments all at once, just like LET and PSETQ do.

`READ-FROM-STRING` (*string* **&optional** *(eof-error-p t) (eof-value nil)* **&key** *(start 0) (end (length string)) (preserve-whitespace nil)*) [Function]

Package:LISP

Reads an object from STRING.

`WRITE-STRING` (*string* **&optional** *(stream *standard-output*)* **&key** *(start 0) (end (length string))*) [Function]

Package:LISP

Outputs STRING and returns it.

`*PRINT-LEVEL*` [Variable]

Package:LISP How many levels deep the GCL printer should print. Unlimited if NIL.

`*PRINT-RADIX*` [Variable]

Package:LISP Whether the GCL printer should print the radix indicator when printing integers and rationals.

`Y-OR-N-P` (**&optional** *(format-string nil)* **&rest** *args*) [Function]

Package:LISP

Asks the user a question whose answer is either 'Y' or 'N'. If FORMAT-STRING is non-NIL, then FRESH-LINE operation is performed, a message is printed as if FORMAT-STRING and ARGs were given to FORMAT, and then a prompt "(Y or N)" is printed. Otherwise, no prompt will appear.

`MAKE-BROADCAST-STREAM` (**&rest** *streams*) [Function]

Package:LISP

Returns an output stream which sends its output to all of the given streams.

`READ-CHAR` (**&optional** *(stream *standard-input*) (eof-error-p t) (eof-value nil) (recursive-p nil)*) [Function]
Package:LISP

Reads a character from STREAM.

`PEEK-CHAR` (**&optional** *(peek-type nil) (stream *standard-input*) (eof-error-p t) (eof-value nil) (recursive-p nil)*) [Function]
Package:LISP

Peeks at the next character in the input stream STREAM.

`OUTPUT-STREAM-P` (*stream*) [Function]
Package:LISP

Returns non-nil if STREAM can handle output operations; NIL otherwise.

`*QUERY-IO*` [Variable]
Package:LISP The query I/O stream.

`*READ-BASE*` [Variable]
Package:LISP The radix that the GCL reader reads numbers in.

`WITH-OPEN-STREAM` [Macro]
Package:LISP

Syntax:

```
(with-open-stream (var stream) {decl}* {form}*)
```

Evaluates FORMs as a PROGN with VAR bound to the value of STREAM. The stream is automatically closed on exit.

`WITH-INPUT-FROM-STRING` [Macro]
Package:LISP

Syntax:

```
(with-input-from-string (var string {keyword value}*) {decl}*
{form}*)
```

Binds VAR to an input stream that returns characters from STRING and evaluates the FORMs. The stream is automatically closed on exit. Allowed keywords are :INDEX, :START, and :END.

`CLEAR-INPUT` (**&optional** *(stream *standard-input*)*) [Function]
Package:LISP Clears the input stream STREAM.

`*TERMINAL-IO*` [Variable]
Package:LISP The terminal I/O stream.

`LISTEN` (**&optional** *(stream *standard-input*)*) [Function]
Package:LISP

Returns T if a character is available on STREAM; NIL otherwise. This function does not correctly work in some versions of GCL because of the lack of such mechanism in the underlying operating system.

`MAKE-PATHNAME` (**&key** *(defaults (parse-namestring "" (pathname-host *default-pathname-defaults*))) (host (pathname-host defaults)) (device (pathname-device defaults)) (directory (pathname-directory defaults)) (name (pathname-name defaults)) (type (pathname-type defaults)) (version (pathname-version defaults))*) [Function]

Package:LISP

Create a pathname from HOST, DEVICE, DIRECTORY, NAME, TYPE and VERSION.

`PATHNAME-TYPE` (*pathname*) [Function]

Package:LISP

Returns the type slot of PATHNAME.

`*PRINT-GENSYM*` [Variable]

Package:LISP Whether the GCL printer should prefix symbols with no home package with "#:".

`READ-LINE` (**&optional** *(stream *standard-input*) (eof-error-p t) (eof-value nil) (recursive-p nil)*) [Function]

Package:LISP

Returns a line of text read from STREAM as a string, discarding the newline character.

Note that when using line at a time input under unix, input forms will always be followed by a #\newline. Thus if you do

>(read-line) "" nil

the empty string will be returned. After lisp reads the (read-line) it then invokes (read-line). This happens before it does anything else and so happens before the newline character immediately following (read-line) has been read. Thus read-line immediately encounters a #\newline and so returns the empty string. If there had been other characters before the #\newline it would have been different:

>(read-line) how are you " how are you" nil

If you want to throw away "" input, you can do that with the following:

(sloop::sloop while (equal (setq input (read-line)) ""))

You may also want to use character at a time input, but that makes input editing harder. nicolas% stty cbreak nicolas% gcl GCL (GNU Common Lisp) Version(1.1.2) Mon Jan 9 12:58:22 MET 1995 Licensed under GNU Public Library License Contains Enhancements by W. Schelter

>(let ((ifilename nil)) (format t "~%Input file name: ") (setq ifilename (read-line))) Input file name: /tmp/myfile "/tmp/myfile"

>(bye)Bye.

`WRITE-TO-STRING` (*object* **&key** *(escape *print-escape*) (radix *print-radix*) (base *print-base*) (circle *print-circle*) (pretty *print-pretty*) (level *print-level*) (length *print-length*) (case *print-case*) (array *print-array*) (gensym *print-gensym*)*) [Function]

Package:LISP

Returns as a string the printed representation of OBJECT in the specified mode. See the variable docs of *PRINT-...* for the mode.

`PATHNAMEP` (*x*) [Function]
Package:LISP

Returns T if X is a pathname object; NIL otherwise.

`READTABLEP` (*x*) [Function]
Package:LISP

Returns T if X is a readtable object; NIL otherwise.

`READ` (**&optional** *(stream *standard-input*) (eof-error-p t) (eof-value nil) (recursivep nil)*) [Function]
Package:LISP

Reads in the next object from STREAM.

`NAMESTRING` (*pathname*) [Function]
Package:LISP

Returns the full form of PATHNAME as a string.

`UNREAD-CHAR` (*character* **&optional** *(stream *standard-input*)*) [Function]
Package:LISP

Puts CHARACTER back on the front of the input stream STREAM.

`CLOSE` (*stream* **&key** *(abort nil)*) [Function]
Package:LISP

Closes STREAM. A non-NIL value of :ABORT indicates an abnormal termination.

`*PRINT-LENGTH*` [Variable]
Package:LISP How many elements the GCL printer should print at each level of nested data object. Unlimited if NIL.

`SET-SYNTAX-FROM-CHAR` (*to-char from-char* **&optional** *(to-readtable *readtable*) (from-readtable nil)*) [Function]
Package:LISP

Makes the syntax of TO-CHAR in TO-READTABLE be the same as the syntax of FROM-CHAR in FROM-READTABLE.

`INPUT-STREAM-P` (*stream*) [Function]
Package:LISP

Returns non-NIL if STREAM can handle input operations; NIL otherwise.

`PATHNAME` (*x*) [Function]
Package:LISP

Turns X into a pathname. X may be a string, symbol, stream, or pathname.

`FILE-NAMESTRING` (*pathname*) [Function]
Package:LISP

Returns the written representation of PATHNAME as a string.

`MAKE-DISPATCH-MACRO-CHARACTER` (*char* **&optional** *(non-terminating-p nil) (readtable *readtable*)*) [Function]
Package:LISP

Causes the character CHAR to be a dispatching macro character in READTABLE.

`*STANDARD-OUTPUT*` [Variable]
Package:LISP The default output stream used by the GCL printer.

`MAKE-TWO-WAY-STREAM` (*input-stream output-stream*) [Function]
Package:LISP

Returns a bidirectional stream which gets its input from INPUT-STREAM and sends its output to OUTPUT-STREAM.

`*PRINT-ESCAPE*` [Variable]
Package:LISP Whether the GCL printer should put escape characters whenever appropriate.

`COPY-READTABLE` (**&optional** *(from-readtable *readtable*) (to-readtable nil)*) [Function]
Package:LISP

Returns a copy of the readtable FROM-READTABLE. If TO-READTABLE is non-NIL, then copies into TO-READTABLE. Otherwise, creates a new readtable.

`DIRECTORY-NAMESTRING` (*pathname*) [Function]
Package:LISP

Returns the directory part of PATHNAME as a string.

`TRUENAME` (*pathname*) [Function]
Package:LISP

Returns the pathname for the actual file described by PATHNAME.

`*READ-SUPPRESS*` [Variable]
Package:LISP When the value of this variable is NIL, the GCL reader operates normally. When it is non-NIL, then the reader parses input characters but much of what is read is not interpreted.

`GET-DISPATCH-MACRO-CHARACTER` (*disp-char sub-char* **&optional** *(readtable *readtable*)*) [Function]
Package:LISP

Returns the macro-character function for SUB-CHAR under DISP-CHAR.

`PATHNAME-DEVICE` (*pathname*) [Function]
Package:LISP

Returns the device slot of PATHNAME.

`READ-CHAR-NO-HANG` (**&optional** *(stream *standard-input*) (eof-error-p t) (eof-value nil) (recursive-p nil)*) [Function]
Package:LISP

Returns the next character from STREAM if one is available; NIL otherwise.

`FRESH-LINE` (**&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Outputs a newline if it is not positioned at the beginning of a line. Returns T if it output a newline; NIL otherwise.

`WRITE-CHAR` (*char* **&optional** *(stream *standard-output*)*) [Function]
Package:LISP

Outputs CHAR and returns it.

`PARSE-NAMESTRING` (*thing* **&optional** *host (defaults *default-pathname-defaults*)* **&key** *(start 0) (end (length thing)) (junk-allowed nil)*) [Function]
Package:LISP

Parses a string representation of a pathname into a pathname. HOST is ignored.

`PATHNAME-DIRECTORY` (*pathname*) [Function]
Package:LISP

Returns the directory slot of PATHNAME.

`GET-MACRO-CHARACTER` (*char* **&optional** *(readtable *readtable*)*) [Function]
Package:LISP

Returns the function associated with CHAR and, as a second value, returns the non-terminating-p flag.

`FORMAT` (*destination control-string* **&rest** *arguments*) [Function]
Package:LISP

Provides various facilities for formatting output. DESTINATION controls where the result will go. If DESTINATION is T, then the output is sent to the standard output stream. If it is NIL, then the output is returned in a string as the value of the call. Otherwise, DESTINATION must be a stream to which the output will be sent.

CONTROL-STRING is a string to be output, possibly with embedded formatting directives, which are flagged with the escape character `"~"`. Directives generally expand into additional text to be output, usually consuming one or more of ARGUMENTs in the process.

A few useful directives are:

```
~A, ~nA, ~n@A Prints one argument as if by PRINC
~S, ~nS, ~n@S Prints one argument as if by PRIN1
~D, ~B, ~O, ~X Prints one integer in decimal, binary, octal, and hexa
~% Does TERPRI
~& Does FRESH-LINE
```

where n is the minimal width of the field in which the object is printed. ~nA and ~nS put padding spaces on the right; ~n@A and ~n@S put on the left.

```
~R  is for printing numbers in various formats.

  ~nR   prints arg in radix n.
```

```
  ~R    prints arg as a cardinal english number: two
  ~:R   prints arg as an ordinal english number: third
  ~@R   prints arg as an a Roman Numeral: VII
  ~:@R   prints arg as an old Roman Numeral: IIII

~C prints a character.
  ~:C represents non printing characters by their pretty names,eg Space
  ~@C uses the #\ syntax to allow the reader to read it.

~F prints a floating point number arg.
  The full form is ~w,d,k,overflowchar,padcharF
  w represents the total width of the printed representation (vari-
able if
    not present)
  d the number of fractional digits to display
    (format nil "~,2f" 10010.0314) --> "10010.03"
  k arg is multiplied by 10^k before printing it as a decimal number.
  overflowchar width w characters copies of the overflow character will
    be printed.   eg(format t "X>~5,2,,'?F<X" 100.034) --> X>?????<X
  padchar is the character to pad with
    (format t "X>~10,2,1,'?,'bF<X" 100.03417) -->X>bbb1000.34<X
  @ makes + sign print if the arg is positive

~@[print-if-true~]
```

if arg is not nil, then it is retained as an arg for further printing, otherwise it is used up

```
(format nil "~@[x = ~d~]~a" nil 'bil) --> "BIL"
(format nil "~@[x = ~d ~]~a" 8) --> "x = 8 BIL"
```

`PATHNAME-NAME` (*pathname*) [Function]

Package:LISP

Returns the name slot of PATHNAME.

`MAKE-STRING-OUTPUT-STREAM` () [Function]

Package:LISP

Returns an output stream which will accumulate all output given it for the benefit of the function GET-OUTPUT-STREAM-STRING.

`MAKE-SYNONYM-STREAM` (*symbol*) [Function]

Package:LISP

Returns a stream which performs its operations on the stream which is the value of the dynamic variable named by SYMBOL.

`*LOAD-VERBOSE*` [Variable]

Package:LISP The default for the VERBOSE argument to LOAD.

`*PRINT-CIRCLE*` [Variable]

Package:LISP Whether the GCL printer should take care of circular lists.

`*PRINT-PRETTY*` [Variable]
Package:LISP Whether the GCL printer should pretty-print. See the function doc of PPRINT for more information about pretty-printing.

`FILE-WRITE-DATE` (*file*) [Function]
Package:LISP

Returns the time at which the specified file is written, as an integer in universal time format. FILE may be a string or a stream.

`PRIN1-TO-STRING` (*object*) [Function]
Package:LISP

Returns as a string the printed representation of OBJECT in the mostly readable representation. Equivalent to (WRITE-TO-STRING OBJECT :ESCAPE T).

`MERGE-PATHNAMES` (*pathname* **&optional** *(defaults *default-pathname-defaults*) default-version*) [Function]
Package:LISP

Fills in unspecified slots of PATHNAME from DEFAULTS. DEFAULT-VERSION is ignored in GCL.

`READ-BYTE` (*stream* **&optional** *(eof-error-p t) (eof-value nil)*) [Function]
Package:LISP

Reads the next byte from STREAM.

`PRINC-TO-STRING` (*object*) [Function]
Package:LISP

Returns as a string the printed representation of OBJECT without escape characters. Equivalent to (WRITE-TO-STRING OBJECT :ESCAPE NIL).

`*STANDARD-INPUT*` [Variable]
Package:LISP The default input stream used by the GCL reader.

`PROBE-FILE` (*file*) [Function]
Package:LISP

Returns the truename of file if the file exists. Returns NIL otherwise.

`PATHNAME-VERSION` (*pathname*) [Function]
Package:LISP

Returns the version slot of PATHNAME.

`WRITE-LINE` (*string* **&optional** *(stream *standard-output*)* **&key** *(start 0) (end (length string))*) [Function]
Package:LISP

Outputs STRING and then outputs a newline character. Returns STRING.

`WRITE` (*object* **&key** *(stream *standard-output*) (escape *print-escape*) (radix *print-radix*) (base *print-base*) (circle *print-circle*) (pretty *print-pretty*) (level *print-level*) (length *print-length*) (case *print-case*) (array *print-array*) (gensym *print-gensym*)*) [Function]

Package:LISP

Prints OBJECT in the specified mode. See the variable docs of *PRINT-...* for the mode.

`GET-OUTPUT-STREAM-STRING` (*stream*) [Function]

Package:LISP

Returns a string of all the characters sent to STREAM made by MAKE-STRING-OUTPUT-STREAM since the last call to this function.

`READ-DELIMITED-LIST` (*char* **&optional** *(stream *standard-input*) (recursive-p nil)*) [Function]

Package:LISP

Reads objects from STREAM until the next character after an object's representation is CHAR. Returns a list of the objects read.

`READLINE-ON` () [Function]

Package:SI

Begins readline command editing mode when possible. In addition to the basic readline editing features, command word completion is implemented according to the following scheme:

[[pkg]:[:]]txt

pkg – an optional package specifier. Defaults to the current package. The symbols in this package and those in the packages in this package's use list will be searched.

:[:] – an optional internal/external specifier. Defaults to external. The keyword package is denoted by a single colon at the beginning of the token. Only symbols of this type will be searched for completion.

txt – a string. Symbol names beginning with this string are completed. The comparison is case insensitive.

`READLINE-OFF` () [Function]

Package:SI

Disables readline command editing mode.

`*READLINE-PREFIX*` [Variable]

Package:SI

A string implicitly prepended to input text for use in readline command completion. If this string contains one or more colons, it is used to specify the default package and internal/external setting for searched symbols in the case that the supplied text itself contains no explicit package specification. If this string contains characters after the colon(s), or contains no colons at all, it is treated as a symbol name prefix. In this case, the prefix is matched first, then the supplied text, and the completion returned

is relative to the supplied text itself, i.e. contains no prefix. For example, the setting "maxima::$" will complete input text "int" according to the internal symbols in the maxima package of the form "maxima::$int...", and return suggestions to the user of the form "int...".

6 Special Forms and Functions

`LAMBDA-LIST-KEYWORDS` [Constant]

Package:LISP List of all the lambda-list keywords used in GCL.

`THE` [Special Form]

Package:LISP

Syntax:

```
(the value-type form)
```

Declares that the value of FORM must be of VALUE-TYPE. Signals an error if this is not the case.

`SETF` [Special Form]

Package:LISP

Syntax:

```
(setf {place newvalue}*)
```

Replaces the value in PLACE with the value of NEWVALUE, from left to right. Returns the value of the last NEWVALUE. Each PLACE may be any one of the following:

A symbol that names a variable.

A function call form whose first element is the name of the following functions:

```
nth elt subseq rest first ... tenth
c?r c??r c???r c????r
aref svref char schar bit sbit fill-poiter
get getf documentation symbol-value symbol-function
symbol-plist macro-function gethash
char-bit ldb mask-field
apply
```

where '?' stands for either 'a' or 'd'.

the form (THE type place) with PLACE being a place recognized by SETF.

a macro call which expands to a place recognized by SETF.

any form for which a DEFSETF or DEFINE-SETF-METHOD declaration has been made.

`WHEN` [Special Form]

Package:LISP

Syntax:

```
(when test {form}*)
```

If TEST evaluates to non-NIL, then evaluates FORMs as a PROGN. If not, simply returns NIL.

`CCASE` [Macro]

Package:LISP

Syntax:

```
(ccase keyplace {({key | ({key}*)} {form}*)}*)
```

Evaluates KEYPLACE and tries to find the KEY that is EQL to the value of KEYPLACE. If one is found, then evaluates FORMs that follow the KEY and returns the value(s) of the last FORM. If not, signals a correctable error.

`MACROEXPAND` (*form* **&optional** *(env nil)*) [Function]

Package:LISP

If FORM is a macro form, then expands it repeatedly until it is not a macro any more. Returns two values: the expanded form and a T-or-NIL flag indicating whether the original form was a macro.

`MULTIPLE-VALUE-CALL` [Special Form]

Package:LISP

Syntax:

```
(multiple-value-call function {form}*)
```

Calls FUNCTION with all the values of FORMs as arguments.

`DEFSETF` [Macro]

Package:LISP

Syntax:

```
(defsetf access-fun {update-fun [doc] |
                                lambda-list (store-var) {decl | doc}*
{form}*)
```

Defines how to SETF a generalized-variable reference of the form (ACCESS-FUN ...). The doc-string DOC, if supplied, is saved as a SETF doc and can be retrieved by (documentation 'NAME 'setf).

```
(defsetf access-fun update-fun) defines an expansion from
(setf (ACCESS-FUN arg1 ... argn) value) to (UPDATE-FUN arg1 ... argn value).

(defsetf access-fun lambda-list (store-var) . body) defines a macro which
```

expands

```
(setf (ACCESS-FUN arg1 ... argn) value) into the form
(let* ((temp1 ARG1) ... (tempn ARGn) (temp0 value)) rest)
```

where REST is the value of BODY with parameters in LAMBDA-LIST bound to the symbols TEMP1 ... TEMPn and with STORE-VAR bound to the symbol TEMP0.

`TAGBODY` [Special Form]

Package:LISP

Syntax:

```
(tagbody {tag | statement}*)
```

Executes STATEMENTs and returns NIL if it falls off the end.

`ETYPECASE` [Macro]

Package:LISP

Syntax:

```
(etypecase keyform {(type {form}*)}*)
```

Evaluates KEYFORM and tries to find the TYPE in which the value of KEYFORM belongs. If one is found, then evaluates FORMs that follow the KEY and returns the value(s) of the last FORM. If not, signals an error.

`LET*` [Special Form]

Package:LISP

Syntax:

```
(let* ({var | (var [value])}*) {decl}* {form}*)
```

Initializes VARs, binding them to the values of VALUEs (which defaults to NIL) from left to right, then evaluates FORMs as a PROGN.

`PROG1` [Special Form]

Package:LISP

Syntax:

```
(prog1 first {form}*)
```

Evaluates FIRST and FORMs in order, and returns the (single) value of FIRST.

`DEFUN` [Special Form]

Package:LISP

Syntax:

```
(defun name lambda-list {decl | doc}* {form}*)
```

Defines a function as the global function definition of the symbol NAME. The complete syntax of a lambda-list is: ({var}* [&optional {var | (var [initform [svar]])}*] [&rest var] [&key {var | ({var | (keyword var)} [initform [svar]])}* [&allow-other-keys]] [&aux {var | (var [initform])}*]) The doc-string DOC, if supplied, is saved as a FUNCTION doc and can be retrieved by (documentation 'NAME 'function).

`MULTIPLE-VALUE-BIND` [Special Form]

Package:LISP

Syntax:

```
(multiple-value-bind ({var}*) values-form {decl}* {form}*)
```

Binds the VARiables to the results of VALUES-FORM, in order (defaulting to NIL) and evaluates FORMs in order.

`DECLARE` [Special Form]

Package:LISP

Syntax:

```
(declare {decl-spec}*)
```

Gives a declaration. Possible DECL-SPECs are: (SPECIAL {var}*) (TYPE type {var}*) where 'TYPE' is one of the following symbols

```
array fixnum package simple-bit-vector
```

```
atom float pathname simple-string
bignum function random-state simple-vector
bit hash-table ratio single-float
bit-vector integer rational standard-char
character keyword readtable stream
common list sequence string
compiled-function  long-float short-float string-char
complex nil signed-byte symbol
cons null unsigned-byte t
double-float number simple-array vector
```

'TYPE' may also be a list containing one of the above symbols as its first element and more specific information later in the list. For example

```
(vector long-float 80) ; vector of 80 long-floats.
(array long-float *)   ; array of long-floats
(array fixnum)         ; array of fixnums
(array * 30)           ; an array of length 30 but unspecified type
```

A list of 1 element may be replaced by the symbol alone, and a list ending in '*' may drop the the final '*'.

```
(OBJECT {var}*)
(FTYPE type {function-name}*)
    eg: ;; function of two required args and optional args and one value:
     (ftype (function (t t *) t) sort reduce)
        ;; function with 1 arg of general type returning 1 fixnum as value.
     (ftype (function (t) fixnum) length)
(FUNCTION function-name ({arg-type}*) {return-type}*)
(INLINE {function-name}*)
(NOTINLINE {function-name}*)
(IGNORE {var}*)
(OPTIMIZE {({SPEED | SPACE | SAFETY | COMPILATION-SPEED} {0 | 1 | 2 | 3})}*)
(DECLARATION {non-standard-decl-name}*)
(:DYNAMIC-EXTENT {var}*) ;GCL-specific.
```

DEFMACRO [Special Form]

Package:LISP

Syntax:

```
(defmacro name defmacro-lambda-list {decl | doc}* {form}*)
```

Defines a macro as the global macro definition of the symbol NAME. The complete syntax of a defmacro-lambda-list is:

([&whole var] [&environment var] {pseudo-var}* [&optional {var | (pseudo-var [initform [pseudo-var]])}*] {[{&rest | &body} pseudo-var] [&key {var | ({var | (keyword pseudo-var)} [initform [pseudo-var]])}* [&allow-other-keys]] [&aux {var | (pseudo-var [initform])}*] | . var})

where pseudo-var is either a symbol or a list of the following form:

({pseudo-var}* [&optional {var | (pseudo-var [initform [pseudo-var]])}*] {[{&rest | &body} pseudo-var] [&key {var | ({var | (keyword pseudo-var)} [initform [pseudo-var]])}* [&allow-other-keys]] [&aux {var | (pseudo-var [initform])}*] | . var})

As a special case, a non-NIL symbol is accepcted as a defmacro-lambda-list: (DEFMACRO <name> <symbol> ...) is equivalent to (DEFMACRO <name> (&REST <symbol>) ...). The doc-string DOC, if supplied, is saved as a FUNCTION doc and can be retrieved by (documentation 'NAME 'function). See the type doc of LIST for the backquote macro useful for defining macros. Also, see the function doc of PPRINT for the output-formatting.

`*EVALHOOK*` [Variable]

Package:LISP If *EVALHOOK* is not NIL, its value must be a function that can receive two arguments: a form to evaluate and an environment. This function does the evaluation instead of EVAL.

`FUNCTIONP` (*x*) [Function]

Package:LISP

Returns T if X is a function, suitable for use by FUNCALL or APPLY. Returns NIL otherwise.

`LAMBDA-PARAMETERS-LIMIT` [Constant]

Package:LISP The exclusive upper bound on the number of distinct parameter names that may appear in a single lambda-list. Actually, however, there is no such upper bound in GCL.

`FLET` [Special Form]

Package:LISP

Syntax:

```
(flet ({(name lambda-list {decl | doc}* {form}*)}*) . body)
```

Evaluates BODY as a PROGN, with local function definitions in effect. BODY is the scope of each local function definition. Since the scope does not include the function definitions themselves, the local function can reference externally defined functions of the same name. See the doc of DEFUN for the complete syntax of a lambda-list. Doc-strings for local functions are simply ignored.

`ECASE` [Macro]

Package:LISP

Syntax:

```
(ecase keyform {({key | ({key}*)} {form}*)}*)
```

Evaluates KEYFORM and tries to find the KEY that is EQL to the value of KEYFORM. If one is found, then evaluates FORMs that follow the KEY and returns the value(s) of the last FORM. If not, signals an error.

`PROG2` [Special Form]

Package:LISP

Syntax:

```
(prog2 first second {forms}*)
```

Evaluates FIRST, SECOND, and FORMs in order, and returns the (single) value of SECOND.

`PROGV` [Special Form]

Package:LISP

Syntax:

```
(progv symbols values {form}*)
```

SYMBOLS must evaluate to a list of variables. VALUES must evaluate to a list of initial values. Evaluates FORMs as a PROGN, with each variable bound (as special) to the corresponding value.

`QUOTE` [Special Form]

Package:LISP

Syntax:

```
(quote x)
```

or 'x Simply returns X without evaluating it.

`DOTIMES` [Special Form]

Package:LISP

Syntax:

```
(dotimes (var countform [result]) {decl}* {tag | statement}*)
```

Executes STATEMENTs, with VAR bound to each number between 0 (inclusive) and the value of COUNTFORM (exclusive). Then returns the value(s) of RESULT (which defaults to NIL).

`SPECIAL-FORM-P` (*symbol*) [Function]

Package:LISP

Returns T if SYMBOL globally names a special form; NIL otherwise. The special forms defined in Steele's manual are:

```
block if progv
catch labels quote
compiler-let let return-from
declare let* setq
eval-when macrolet tagbody
flet multiple-value-call the
function multiple-value-prog1 throw
go progn unwind-protect
```

In addition, GCL implements the following macros as special forms, though of course macro-expanding functions such as MACROEXPAND work correctly for these macros.

```
and incf prog1
case locally prog2
cond loop psetq
decf multiple-value-bind push
defmacro multiple-value-list return
defun multiple-value-set setf
do or unless
do* pop when
```

```
dolist prog
dotimes prog*
```

FUNCTION [Special Form]

Package:LISP

Syntax:

```
(function x)
```

or #'x If X is a lambda expression, creates and returns a lexical closure of X in the current lexical environment. If X is a symbol that names a function, returns that function.

MULTIPLE-VALUES-LIMIT [Constant]

Package:LISP The exclusive upper bound on the number of values that may be returned from a function. Actually, however, there is no such upper bound in GCL.

APPLYHOOK (*function args evalhookfn applyhookfn* **&optional** *(env nil)*) [Function]

Package:LISP

Applies FUNCTION to ARGS, with *EVALHOOK* bound to EVALHOOKFN and with *APPLYHOOK* bound to APPLYHOOKFN. Ignores the hook function once, for the top-level application of FUNCTION to ARGS.

MACROEXPAND-HOOK [Variable]

Package:LISP Holds a function that can take two arguments (a macro expansion function and the macro form to be expanded) and returns the expanded form. This function is whenever a macro-expansion takes place. Initially this is set to #'FUNCALL.

PROG* [Special Form]

Package:LISP

Syntax:

```
(prog* ({var | (var [init])}*) {decl}* {tag | statement}*)
```

Creates a NIL block, binds VARs sequentially, and then executes STATEMENTs.

BLOCK [Special Form]

Package:LISP

Syntax:

```
(block name {form}*)
```

The FORMs are evaluated in order, but it is possible to exit the block using (RETURN-FROM name value). The RETURN-FROM must be lexically contained within the block.

PROGN [Special Form]

Package:LISP

Syntax:

```
(progn {form}*)
```

Evaluates FORMs in order, and returns whatever the last FORM returns.

`APPLY` (*function arg* **&rest** *more-args*) [Function]

Package:LISP

Applies FUNCTION. The arguments to the function consist of all ARGs except for the last, and all elements of the last ARG.

`LABELS` [Special Form]

Package:LISP

Syntax:

```
(labels ({(name lambda-list {decl | doc}* {form}*)}*) . body)
```

Evaluates BODY as a PROGN, with the local function definitions in effect. The scope of the locally defined functions include the function definitions themselves, so their definitions may include recursive references. See the doc of DEFUN for the complete syntax of a lambda-list. Doc-strings for local functions are simply ignored.

`RETURN` [Special Form]

Package:LISP

Syntax:

```
(return [result])
```

Returns from the lexically surrounding NIL block. The value of RESULT, which defaults to NIL, is returned as the value of the block.

`TYPECASE` [Macro]

Package:LISP

Syntax:

```
(typecase keyform {(type {form}*)}*)
```

Evaluates KEYFORM and tries to find the TYPE in which the value of KEYFORM belongs. If one is found, then evaluates FORMs that follow the KEY and returns the value of the last FORM. If not, simply returns NIL.

`AND` [Special Form]

Package:LISP

Syntax:

```
(and {form}*)
```

Evaluates FORMs in order from left to right. If any FORM evaluates to NIL, returns immediately with the value NIL. Else, returns the value(s) of the last FORM.

`LET` [Special Form]

Package:LISP

Syntax:

```
(let ({var | (var [value])}*) {decl}* {form}*)
```

Initializes VARs, binding them to the values of VALUEs (which defaults to NIL) all at once, then evaluates FORMs as a PROGN.

`COND` [Special Form]

Package:LISP

Syntax:

```
(cond {(test {form}*)}*)
```

Evaluates each TEST in order until one evaluates to a non-NIL value. Then evaluates the associated FORMs in order and returns the value(s) of the last FORM. If no forms follow the TEST, then returns the value of the TEST. Returns NIL, if all TESTs evaluate to NIL.

`GET-SETF-METHOD-MULTIPLE-VALUE` (*form*) [Function]

Package:LISP Returns the five values (or five 'gangs') constituting the SETF method for FORM. See the doc of DEFINE-SETF-METHOD for the meanings of the gangs. The third value (i.e., the list of store variables) may consist of any number of elements. See the doc of GET-SETF-METHOD for comparison.

`CATCH` [Special Form]

Package:LISP

Syntax:

```
(catch tag {form}*)
```

Sets up a catcher with that value TAG. Then evaluates FORMs as a PROGN, but may possibly abort the evaluation by a THROW form that specifies the value EQ to the catcher tag.

`DEFINE-MODIFY-MACRO` [Macro]

Package:LISP

Syntax:

```
(define-modify-macro name lambda-list fun [doc])
```

Defines a read-modify-write macro, like PUSH and INCF. The defined macro will expand a form (NAME place val1 ... valn) into a form that in effect SETFs the value of the call (FUN PLACE arg1 ... argm) into PLACE, where arg1 ... argm are parameters in LAMBDA-LIST which are bound to the forms VAL1 ... VALn. The doc-string DOC, if supplied, is saved as a FUNCTION doc and can be retrieved by (documentation 'NAME 'function).

`MACROEXPAND-1` (*form* **&optional** *(env nil)*) [Function]

Package:LISP

If FORM is a macro form, then expands it once. Returns two values: the expanded form and a T-or-NIL flag indicating whether the original form was a macro.

`FUNCALL` (*function* **&rest** *arguments*) [Function]

Package:LISP

Applies FUNCTION to the ARGUMENTs

`CALL-ARGUMENTS-LIMIT` [Constant]

Package:LISP The upper exclusive bound on the number of arguments that may be passed to a function. Actually, however, there is no such upper bound in GCL.

`CASE` [Special Form]

Package:LISP

Syntax:

```
(case keyform {({key | ({key}*)} {form}*)}*)
```

Evaluates KEYFORM and tries to find the KEY that is EQL to the value of KEYFORM. If one is found, then evaluates FORMs that follow the KEY and returns the value(s) of the last FORM. If not, simply returns NIL.

`DEFINE-SETF-METHOD` [Macro]

Package:LISP

Syntax:

```
(define-setf-method access-fun defmacro-lambda-list {decl | doc}*
          {form}*)
```

Defines how to SETF a generalized-variable reference of the form (ACCESS-FUN ...). When a form (setf (ACCESS-FUN arg1 ... argn) value) is being evaluated, the FORMs are first evaluated as a PROGN with the parameters in DEFMACRO-LAMBDA-LIST bound to ARG1 ... ARGn. Assuming that the last FORM returns five values (temp-var-1 ... temp-var-k) (value-from-1 ... value-form-k) (store-var) storing-form access-form in order, the whole SETF is then expanded into (let* ((temp-var-1 value-from-1) ... (temp-k value-form-k) (store-var VALUE)) storing-from) Incidentally, the five values are called the five gangs of a SETF method. The doc-string DOC, if supplied, is saved as a SETF doc and can be retrieved by (documentation 'NAME 'setf).

`COMPILER-LET` [Special Form]

Package:LISP

Syntax:

```
(compiler-let ({var | (var [value])}*) {form}*)
```

When interpreted, this form works just like a LET form with all VARs declared special. When compiled, FORMs are processed with the VARs bound at compile time, but no bindings occur when the compiled code is executed.

`VALUES` (**&rest** *args*) [Function]

Package:LISP

Returns ARGs in order, as values.

`MULTIPLE-VALUE-LIST` [Special Form]

Package:LISP

Syntax:

```
(multiple-value-list form)
```

Evaluates FORM, and returns a list of multiple values it returned.

`MULTIPLE-VALUE-PROG1` [Special Form]

Package:LISP

Syntax:

```
(multiple-value-prog1 form {form}*)
```

Evaluates the first FORM, saves all the values produced, then evaluates the other FORMs. Returns the saved values.

`MACROLET` [Special Form]

Package:LISP

Syntax:

```
(macrolet ({(name defmacro-lambda-list {decl | doc}* . body)}*)
          {form}*)
```

Evaluates FORMs as a PROGN, with the local macro definitions in effect. See the doc of DEFMACRO for the complete syntax of a defmacro-lambda-list. Doc-strings for local macros are simply ignored.

`GO` [Special Form]

Package:LISP

Syntax:

```
(go tag)
```

Jumps to the specified TAG established by a lexically surrounding TAGBODY.

`PROG` [Special Form]

Package:LISP

Syntax:

```
(prog ({var | (var [init])}*) {decl}* {tag | statement}*)
```

Creates a NIL block, binds VARs in parallel, and then executes STATEMENTs.

`*APPLYHOOK*` [Variable]

Package:LISP Used to substitute another function for the implicit APPLY normally done within EVAL. If *APPLYHOOK* is not NIL, its value must be a function which takes three arguments: a function to be applied, a list of arguments, and an environment. This function does the application instead of APPLY.

`RETURN-FROM` [Special Form]

Package:LISP

Syntax:

```
(return-from name [result])
```

Returns from the lexically surrounding block whose name is NAME. The value of RESULT, which defaults to NIL, is returned as the value of the block.

`UNLESS` [Special Form]

Package:LISP

Syntax:

```
(unless test {form}*)
```

If TEST evaluates to NIL, then evaluates FORMs as a PROGN. If not, simply returns NIL.

`MULTIPLE-VALUE-SETQ` [Special Form]

Package:LISP

Syntax:

```
(multiple-value-setq variables form)
```

Sets each variable in the list VARIABLES to the corresponding value of FORM. Returns the value assigned to the first variable.

`LOCALLY` [Special Form]

Package:LISP

Syntax:

```
(locally {decl}* {form}*)
```

Gives local pervasive declarations.

`IDENTITY` (*x*) [Function]

Package:LISP

Simply returns X.

`NOT` (*x*) [Function]

Package:LISP

Returns T if X is NIL; NIL otherwise.

`DEFCONSTANT` [Macro]

Package:LISP

Syntax:

```
(defconstant name initial-value [doc])
```

Declares that the variable NAME is a constant whose value is the value of INITIAL-VALUE. The doc-string DOC, if supplied, is saved as a VARIABLE doc and can be retrieved by (documentation 'NAME 'variable).

`VALUES-LIST` (*list*) [Function]

Package:LISP

Returns all of the elements of LIST in order, as values.

`ERROR` (*control-string* **&rest** *args*) [Function]

Package:LISP

Signals a fatal error.

`IF` [Special Form]

Package:LISP

Syntax:

```
(if test then [else])
```

If TEST evaluates to non-NIL, then evaluates THEN and returns the result. If not, evaluates ELSE (which defaults to NIL) and returns the result.

`UNWIND-PROTECT` [Special Form]

Package:LISP

Syntax:

```
(unwind-protect protected-form {cleanup-form}*)
```

Evaluates PROTECTED-FORM and returns whatever it returned. Guarantees that CLEANUP-FORMs be always evaluated before exiting from the UNWIND-PROTECT form.

`EVALHOOK` (*form evalhookfn applyhookfn* **&optional** *(env nil)*) [Function]

Package:LISP

Evaluates FORM with *EVALHOOK* bound to EVALHOOKFN and *APPLYHOOK* bound to APPLYHOOKFN. Ignores these hooks once, for the top-level evaluation of FORM.

`OR` [Special Form]

Package:LISP

Syntax:

```
(or {form}*)
```

Evaluates FORMs in order from left to right. If any FORM evaluates to non-NIL, quits and returns that (single) value. If the last FORM is reached, returns whatever values it returns.

`CTYPECASE` [Macro]

Package:LISP

Syntax:

```
(ctypecase keyplace {(type {form}*)}*)
```

Evaluates KEYPLACE and tries to find the TYPE in which the value of KEYPLACE belongs. If one is found, then evaluates FORMs that follow the KEY and returns the value(s) of the last FORM. If not, signals a correctable error.

`EVAL` (*exp*) [Function]

Package:LISP

Evaluates EXP and returns the result(s).

`PSETF` [Macro]

Package:LISP

Syntax:

```
(psetf {place newvalue}*)
```

Similar to SETF, but evaluates all NEWVALUEs first, and then replaces the value in each PLACE with the value of the corresponding NEWVALUE. Returns NIL always.

`THROW` [Special Form]

Package:LISP

Syntax:

```
(throw tag result)
```

Evaluates TAG and aborts the execution of the most recent CATCH form that sets up a catcher with the same tag value. The CATCH form returns whatever RESULT returned.

`DEFPARAMETER` [Macro]

Package:LISP

Syntax:

```
(defparameter name initial-value [doc])
```

Declares the variable NAME as a special variable and initializes the value. The doc-string DOC, if supplied, is saved as a VARIABLE doc and can be retrieved by (documentation 'NAME 'variable).

`DEFVAR` [Macro]

Package:LISP

Syntax:

```
(defvar name [initial-value [doc]])
```

Declares the variable NAME as a special variable and, optionally, initializes it. The doc-string DOC, if supplied, is saved as a VARIABLE doc and can be retrieved by (documentation 'NAME 'variable).

7 Compilation

`COMPILE` (*name* **&optional** *(definition nil)*) [Function]

Package:LISP

If DEFINITION is NIL, NAME must be the name of a not-yet-compiled function. In this case, COMPILE compiles the function, installs the compiled function as the global function definition of NAME, and returns NAME. If DEFINITION is non-NIL, it must be a lambda expression and NAME must be a symbol. COMPILE compiles the lambda expression, installs the compiled function as the function definition of NAME, and returns NAME. There is only one exception for this: If NAME is NIL, then the compiled function is not installed but is simply returned as the value of COMPILE. In any case, COMPILE creates temporary files whose filenames are "gazonk***". By default, i.e. if :LEAVE-GAZONK is not supplied or is NIL, these files are automatically deleted after compilation.

`LINK` (*files image* **&optional** *post extra-libs (run-user-init t)* **&aux** *raw init*) [Function]

Package:LISP

On systems where dlopen is used for relocations, one cannot make custom images containing loaded binary object files simply by loading the files and executing save-system. This function is provided for such cases.

After compiling source files into objects, LINK can be called with a list of binary and source FILES which would otherwise normally be loaded in sequence before saving the image to IMAGE. LINK will use the system C linker to link the binary files thus supplied with GCL's objects, using EXTRA-LIBS as well if provided, and producing a raw_IMAGE executable. This executable is then run to initialize first GCL's objects, followed by the supplied files, in order, if RUN-USER-INIT is set. In such a case, source files are loaded at their position in the sequence. Any optional code which should be run after file initialization can be supplied in the POST variable. The image is then saved using save-system to IMAGE.

This method of creating lisp images may also have the advantage that all new object files are kept out of the lisp core and placed instead in the final image's .text section. This should in principle reduce the core size, speed up garbage collection, and forego any performance penalty induced by data cache flushing on some machines.

In both the RAW and SAVED image, any calls to LOAD binary object files which have been specified in this list will bypass the normal load procedure, and simply initialize the already linked in module. One can rely on this feature by disabling RUN-USER-INIT, and instead passing the normal build commands in POST. In the course of executing this code, binary modules previously linked into the .text section of the executable will be initialized at the same point at which they would have normally been loaded into the lisp core, in the executable's .data section. In this way, the user can choose to take advantage of the aforementioned possible benefits of this linking method in a relatively transparent way.

All binary objects specified in FILES must have been compiled with :SYSTEM-P set to T.

`EVAL-WHEN` [Special Form]

Package:LISP

Syntax:

```
(eval-when ({situation}*) {form}*)
```

A situation must be either COMPILE, LOAD, or EVAL. The interpreter evaluates only when EVAL is specified. If COMPILE is specified, FORMs are evaluated at compile time. If LOAD is specified, the compiler arranges so that FORMs be evaluated when the compiled code is loaded.

`COMPILE-FILE` (*input-pathname* **&key** *output-file (load nil) (message-file nil) ;GCL specific keywords: system-p c-debug c-file h-file data-file*) [Function]

Package:LISP

Compiles the file specified by INPUT-PATHNAME and generates a fasl file specified by OUTPUT-FILE. If the filetype is not specified in INPUT-PATHNAME, then ".lsp" is used as the default file type for the source file. :LOAD specifies whether to load the generated fasl file after compilation. :MESSAGE-FILE specifies the log file for the compiler messages. It defaults to the value of the variable COMPILER:*DEFAULT-MESSAGE-FILE*. A non-NIL value of COMPILER::*COMPILE-PRINT* forces the compiler to indicate the form currently being compiled. More keyword parameters are accepted, depending on the version. Most versions of GCL can receive :O-FILE, :C-FILE, :H-FILE, and :DATA-FILE keyword parameters, with which you can control the intermediate files generated by the GCL compiler. Also :C-DEBUG will pass the -g flag to the C compiler.

By top level forms in a file, we mean the value of *top-level-forms* after doing (TF form) for each form read from a file. We define TF as follows:

(defun TF (x) (when (consp x) (setq x (macroexpand x)) (when (consp x) (cond ((member (car x) '(progn eval-when)) (mapcar 'tf (cdr x))) (t (push x *top-level-forms*))))))

Among the common lisp special forms only DEFUN and DEFMACRO will cause actual native machine code to be generated. The rest will be specially treated in an init section of the .data file. This is done so that things like putprop,setq, and many other forms would use up space which could not be usefully freed, if we were to compile to native machine code. If you have other 'ordinary' top level forms which you need to have compiled fully to machine code you may either set compiler::*COMPILE-ORDINARIES* to t, or put them inside a

(PROGN 'COMPILE ...forms-which-need-to-be-compiled)

The compiler will take each of them and make a temporary function which will be compiled and invoked once. It is permissible to wrap a (PROGN 'COMPILE ..) around the whole file. Currently this construction binds the compiler::*COMPILE-ORDINARIES* flag to t. Setting this flag globally to a non nil value to cause all top level forms to generate machine code. This might be useful in a system such as PCL, where a number of top level lambda expressions are given. Note that most common lisps will simply ignore the top level atom 'compile, since it has no side effects.

Defentry, clines, and defcfun also result in machine code being generated.

subsection Evaluation at Compile time

In GCL the eval-when behaviour was changed in order to allow more efficient init code, and also to bring it into line with the resolution passed by the X3j13 committee. Evaluation at compile time is controlled by placing eval-when special forms in the code, or by the value of the variable compiler::*eval-when-defaults* [default value :defaults]. If that variable has value :defaults, then the following hold:

Eval at Compile Type of Top Level Form

Partial:	defstructs, defvar, defparameter
Full:	defmacro, defconstant, defsetf, define-setf-method, deftype, package ops, proclaim
None:	defun, others

By 'partial' we mean (see the X3J13 Common Lisp document (doc/compile-file-handling-of-top-level-forms) for more detail), that functions will not be defined, values will not be set, but other miscellaneous compiler properties will be set: eg properties to inline expand defstruct accessors and testers, defstruct properties allowing subsequent defstructs to include this one, any type hierarch information, special variable information will be set up.

Example:

```
(defun foo () 3)
(defstruct jo a b)
```

As a side effect of compiling these two forms, foo would not have its function cell changed. Neither would jo-a, although it would gain a property which allows it to expand inline to a structure access. Thus if it had a previous definition (as commonly happens from previously loading the file), this previous definition would not be touched, and could well be inconsistent with the compiler properties. Unfortunately this is what the CL standard says to do, and I am just trying to follow it.

If you prefer a more intuitive scheme, of evaling all forms in the file, so that there are no inconsistencies, (previous behaviour of AKCL) you may set compiler::*eval-when-defaults* to '(compile eval load).

The variable compiler::*FASD-DATA* [default t] controls whether an ascii output is used for the data section of the object file. The data section will be in ascii if *fasd-data* is nil or if the system-p keyword is supplied to compile-file and *fasd-data* is not eq to :system-p.

The old GCL variable *compile-time-too* has disappeared.

See OPTIMIZE on how to enable warnings of slow constructs.

`PROCLAIM` (*decl-spec*) [Function]

Package:LISP

Puts the declaration given by DECL-SPEC into effect globally. See the doc of DECLARE for possible DECL-SPECs.

`PROVIDE` (*module-name*) [Function]

Package:LISP

Adds the specified module to the list of modules maintained in *MODULES*.

`COMPILED-FUNCTION-P` (*x*) [Function]
Package:LISP

Returns T if X is a compiled function; NIL otherwise.

`GPROF-START` () [Function]
Package:SYSTEM

GCL now has preliminary support for profiling with gprof, an externally supplied profiling tool at the C level which typically accompanies gcc. Support must be enabled at compile time with –enable-gprof. This function starts the profiling timers and counters.

`GPROF-QUIT` () [Function]
Package:SYSTEM

GCL now has preliminary support for profiling with gprof, an externally supplied profiling tool at the C level which typically accompanies gcc. Support must be enabled at compile time with –enable-gprof. This function reports the profiling results in the form of a call graph to standard output, and clears the profiling arrays. Please note that lisp functions are not (yet) displayed with their lisp names. Please see also the PROFILE function.

`GPROF-SET` (*begin end*) [Function]
Package:SYSTEM

GCL now has preliminary support for profiling with gprof, an externally supplied profiling tool at the C level which typically accompanies gcc. Support must be enabled at compile time with –enable-gprof. This function sets the address range used by GPROF-START in specifying the section of the running program which is to be profiled. All subsequent calls to GPROF-START will use this new address range. By default, the range is set to begin at the starting address of the .text section, and to end at the current end of the running core. These default values can be restored by calling GPROF-SET with both argments set to 0.

`*DEFAULT-SYSTEM-P*` [Variable]
Pakcage:COMPILER Specifies the default setting of :SYSTEM-P used by COMPILE. Defaults to NIL.

`*DEFAULT-C-FILE*` [Variable]
Pakcage:COMPILER Specifies the default setting of :C-FILE used by COMPILE. Defaults to NIL.

`*DEFAULT-H-FILE*` [Variable]
Pakcage:COMPILER Specifies the default setting of :H-FILE used by COMPILE. Defaults to NIL.

`*DEFAULT-DATA-FILE*` [Variable]
Pakcage:COMPILER Specifies the default setting of :DATA-FILE used by COMPILE. Defaults to NIL.

`*FEATURES*` [Variable]

Package:LISP List of symbols that name features of the current version of GCL. These features are used to decide the read-time conditionalization facility provided by '#+' and '#-' read macros. When the GCL reader encounters

```
#+ feature-description form
```

it reads FORM in the usual manner if FEATURE-DESCRIPTION is true. Otherwise, the reader just skips FORM.

```
#- feature-description form
```

is equivalent to

```
#- (not feature-description) form
```

A feature-description may be a symbol, which is true only when it is an element of *FEATURES*. Or else, it must be one of the following:

```
(and feature-desciption-1 ... feature-desciption-n)
(or  feature-desciption-1 ... feature-desciption-n)
(not feature-desciption)
```

The AND description is true only when all of its sub-descriptions are true. The OR description is true only when at least one of its sub-descriptions is true. The NOT description is true only when its sub-description is false.

8 Symbols

`GENSYM` (**&optional** *(x nil)*) [Function]

Package:LISP

Creates and returns a new uninterned symbol whose name is a prefix string (defaults to "G"), followed by a decimal number. The number is incremented by each call to GENSYM. X, if an integer, resets the counter. If X is a string, it becomes the new prefix.

`KEYWORDP` (*x*) [Function]

Package:LISP

Returns T if X is a symbol and it belongs to the KEYWORD package; NIL otherwise.

`REMPROP` (*symbol indicator*) [Function]

Package:LISP

Look on property list of SYMBOL for property with specified INDICATOR. If found, splice this indicator and its value out of the plist, and return T. If not found, returns NIL with no side effects.

`SYMBOL-PACKAGE` (*symbol*) [Function]

Package:LISP

Returns the contents of the package cell of the symbol SYMBOL.

`*PACKAGE*` [Variable]

Package:LISP The current package.

`SHADOWING-IMPORT` (*symbols* **&optional** *(package *package*)*) [Function]

Package:LISP

Imports SYMBOLS into PACKAGE, disregarding any name conflict. If a symbol of the same name is already present, then it is uninterned. SYMBOLS must be a list of symbols or a symbol.

`REMF` [Macro]

Package:LISP

Syntax:

```
(remf place indicator)
```

PLACE may be any place expression acceptable to SETF, and is expected to hold a property list or NIL. This list is destructively altered to remove the property specified by INDICATOR. Returns T if such a property was present; NIL otherwise.

`MAKUNBOUND` (*symbol*) [Function]

Package:LISP

Makes empty the value slot of SYMBOL. Returns SYMBOL.

`USE-PACKAGE` (*packages-to-use* **&optional** *(package *package*)*) [Function]

Package:LISP

Adds all packages in PACKAGE-TO-USE list to the use list for PACKAGE so that the external symbols of the used packages are available as internal symbols in PACKAGE.

`MAKE-SYMBOL` (*string*) [Function]
Package:LISP

Creates and returns a new uninterned symbol whose print name is STRING.

`PSETQ` [Special Form]
Package:LISP

Syntax:

```
(psetq {var form}*)
```

Similar to SETQ, but evaluates all FORMs first, and then assigns each value to the corresponding VAR. Returns NIL always.

`PACKAGE-USED-BY-LIST` (*package*) [Function]
Package:LISP

Returns the list of packages that use PACKAGE.

`SYMBOLP` (*x*) [Function]
Package:LISP

Returns T if X is a symbol; NIL otherwise.

`NIL` [Constant]
Package:LISP Holds NIL.

`SET` (*symbol value*) [Function]
Package:LISP

Assigns the value of VALUE to the dynamic variable named by SYMBOL, and returns the value assigned.

`SETQ` [Special Form]
Package:LISP

Syntax:

```
(setq {var form}*)
```

VARs are not evaluated and must be symbols. Assigns the value of the first FORM to the first VAR, then assigns the value of the second FORM to the second VAR, and so on. Returns the last value assigned.

`UNUSE-PACKAGE` (*packages-to-unuse* **&optional** *(package *package*)*) [Function]
Package:LISP

Removes PACKAGES-TO-UNUSE from the use list for PACKAGE.

`T` [Constant]
Package:LISP Holds T.

`PACKAGE-USE-LIST` (*package*) [Function]
Package:LISP

Returns the list of packages used by PACKAGE.

`LIST-ALL-PACKAGES` () [Function]
Package:LISP

Returns a list of all existing packages.

`COPY-SYMBOL` (*symbol* **&optional** *(copy-props nil)*) [Function]
Package:LISP

Returns a new uninterned symbol with the same print name as SYMBOL. If COPY-PROPS is NIL, the function, the variable, and the property slots of the new symbol have no value. Otherwise, these slots are given the values of the corresponding slots of SYMBOL.

`SYMBOL-PLIST` (*symbol*) [Function]
Package:LISP

Returns the property list of SYMBOL.

`SYMBOL-NAME` (*symbol*) [Function]
Package:LISP

Returns the print name of the symbol SYMBOL.

`FIND-SYMBOL` (*name* **&optional** *(package *package*)*) [Function]
Package:LISP

Returns the symbol named NAME in PACKAGE. If such a symbol is found, then the second value is :INTERN, :EXTERNAL, or :INHERITED to indicate how the symbol is accessible. If no symbol is found then both values are NIL.

`SHADOW` (*symbols* **&optional** *(package *package*)*) [Function]
Package:LISP

Creates an internal symbol in PACKAGE with the same name as each of the specified SYMBOLS. SYMBOLS must be a list of symbols or a symbol.

`FBOUNDP` (*symbol*) [Function]
Package:LISP

Returns T if SYMBOL has a global function definition or if SYMBOL names a special form or a macro; NIL otherwise.

`MACRO-FUNCTION` (*symbol*) [Function]
Package:LISP

If SYMBOL globally names a macro, then returns the expansion function. Returns NIL otherwise.

`IN-PACKAGE` (*package-name* **&key** *(nicknames nil) (use '(lisp))*) [Function]
Package:LISP

Sets *PACKAGE* to the package with PACKAGE-NAME, creating the package if it does not exist. If the package already exists then it is modified to agree with USE and NICKNAMES arguments. Any new nicknames are added without removing any old ones not specified. If any package in the USE list is not currently used, then it is added to the use list.

`MAKE-PACKAGE` (*package-name* **&key** *(nicknames nil) (use '(lisp))*) [Function]
Package:LISP

Makes a new package having the specified PACKAGE-NAME and NICKNAMES. The package will inherit all external symbols from each package in the USE list.

`PACKAGE-SHADOWING-SYMBOLS` (*package*) [Function]
Package:LISP

Returns the list of symbols that have been declared as shadowing symbols in PACKAGE.

`INTERN` (*name* **&optional** *(package *package*)*) [Function]
Package:LISP

Returns a symbol having the specified name, creating it if necessary. Returns as the second value one of the symbols :INTERNAL, :EXTERNAL, :INHERITED, and NIL.

`EXPORT` (*symbols* **&optional** *(package *package*)*) [Function]
Package:LISP

Makes SYMBOLS external symbols of PACKAGE. SYMBOLS must be a list of symbols or a symbol.

`PACKAGEP` (*x*) [Function]
Package:LISP

Returns T if X is a package; NIL otherwise.

`SYMBOL-FUNCTION` (*symbol*) [Function]
Package:LISP

Returns the current global function definition named by SYMBOL.

`SYMBOL-VALUE` (*symbol*) [Function]
Package:LISP

Returns the current value of the dynamic (special) variable named by SYMBOL.

`BOUNDP` (*symbol*) [Function]
Package:LISP

Returns T if the global variable named by SYMBOL has a value; NIL otherwise.

`DOCUMENTATION` (*symbol doc-type*) [Function]
Package:LISP

Returns the doc-string of DOC-TYPE for SYMBOL; NIL if none exists. Possible doc-types are: FUNCTION (special forms, macros, and functions) VARIABLE (dynamic variables, including constants) TYPE (types defined by DEFTYPE) STRUCTURE (structures defined by DEFSTRUCT) SETF (SETF methods defined by DEFSETF, DEFINE-SETF-METHOD, and DEFINE-MODIFY-MACRO) All built-in special forms, macros, functions, and variables have their doc-strings.

`GENTEMP` (**&optional** *(prefix "t") (package *package*)*) [Function]
Package:LISP

Creates a new symbol interned in the package PACKAGE with the given PREFIX.

`RENAME-PACKAGE` (*package new-name* **&optional** *(new-nicknames nil)*) [Function]
Package:LISP

Replaces the old name and nicknames of PACKAGE with NEW-NAME and NEW-NICKNAMES.

`UNINTERN` (*symbol* **&optional** *(package *package*)*) [Function]
Package:LISP

Makes SYMBOL no longer present in PACKAGE. Returns T if SYMBOL was present; NIL otherwise. If PACKAGE is the home package of SYMBOL, then makes SYMBOL uninterned.

`UNEXPORT` (*symbols* **&optional** *(package *package*)*) [Function]
Package:LISP

Makes SYMBOLS no longer accessible as external symbols in PACKAGE. SYMBOLS must be a list of symbols or a symbol.

`PACKAGE-NICKNAMES` (*package*) [Function]
Package:LISP

Returns as a list the nickname strings for the specified PACKAGE.

`IMPORT` (*symbols* **&optional** *(package *package*)*) [Function]
Package:LISP

Makes SYMBOLS internal symbols of PACKAGE. SYMBOLS must be a list of symbols or a symbol.

`GET` (*symbol indicator* **&optional** *(default nil)*) [Function]
Package:LISP

Looks on the property list of SYMBOL for the specified INDICATOR. If this is found, returns the associated value. Otherwise, returns DEFAULT.

`FIND-ALL-SYMBOLS` (*string-or-symbol*) [Function]
Package:LISP

Returns a list of all symbols that have the specified name.

`FMAKUNBOUND` (*symbol*) [Function]
Package:LISP

Discards the global function definition named by SYMBOL. Returns SYMBOL.

`PACKAGE-NAME` (*package*) [Function]
Package:LISP

Returns the string that names the specified PACKAGE.

`FIND-PACKAGE` (*name*) [Function]
Package:LISP

Returns the specified package if it already exists; NIL otherwise. NAME may be a string that is the name or nickname of the package. NAME may also be a symbol, in which case the symbol's print name is used.

`APROPOS-LIST` (*string* **&optional** *(package nil)*) [Function]

Package:LISP

Returns, as a list, all symbols whose print-names contain STRING as substring. If PACKAGE is non-NIL, then only the specified package is searched.

9 Operating System

9.1 Command Line

The variable si::*command-args* is set to the list of strings passed in when gcl is invoked.

Various flags are understood.

`-eval` Call read and then eval on the command argument following `-eval`

`-load` Load the file whose pathname is specified after `-load`.

`-f` Replace si::*command-args* by the the list starting after `-f`. Open the file following `-f` for input, skip the first line, and then read and eval the rest of the forms in the file. This can be used as with the shells to write small shell programs:

```
#!/usr/local/bin/gcl.exe -f
(format t "hello world ~a~%" (nth 1 si::*command-args*))
```

The value si::*command-args* will have the appropriate value. Thus if the above 2 line file is made executable and called `foo` then

```
tutorial% foo billy
hello world billy
```

NOTE: On many systems (eg SunOs) the first line of an executable script file such as:

```
#!/usr/local/bin/gcl.exe -f
```

only reads the first 32 characters! So if your pathname where the executable together with the '-f' amount to more than 32 characters the file will not be recognized. Also the executable must be the actual large binary file, [or a link to it], and not just a `/bin/sh` script. In latter case the `/bin/sh` interpreter would get invoked on the file.

Alternately one could invoke the file `foo` without making it executable:

```
tutorial% gcl -f foo "from bill"
hello world from bill
```

Finally perhaps the best way (why do we save the best for last.. I guess because we only figure it out after all the others..) The following file `myhello` has 4 lines:

```
#!/bin/sh
#| Lisp will skip the next 2 lines on reading
exec gcl   -f "$0" $ |#
(format t "hello world ~a~%" (nth 1 si::*command-args*))
```

```
marie% chmod a+x myhello
marie% myhello bill
hello world bill
```

The advantage of this method is that `gcl` can itself be a shell script, which sets up environment and so on. Also the normal path will be searched to find `gcl` The disadvantage is that this would cause 2 invocations of `sh` and one invocation of `gcl`. The plan using `gcl.exe` bypasses the `sh` entirely. Inded

invoking `gcl.exe` to print `hello world` is faster on most systems than a similar `csh` or `bash` script, but slightly slower than the old `sh`.

`-batch` Do not enter the command print loop. Useful if the other command line arguments do something. Do not print the License and acknowledgement information. Note if your program does print any License information, it must print the GCL header information also.

`-dir` Directory where the executable binary that is running is located. Needed by save and friends. This gets set as si::*system-directory*

`-libdir`

```
-libdir /d/wfs/gcl-2.0/
```

would mean that the files like gcl-tk/tk.o would be found by concatting the path to the libdir path, ie in

```
/d/wfs/gcl-2.0/gcl-tk/tk.o
```

`-compile` Invoke the compiler on the filename following `-compile`. Other flags affect compilation.

`-o-file` If nil follows `-o-file` then do not produce an `.o` file.

`-c-file` If `-c-file` is specified, leave the intermediate `.c` file there.

`-h-file` If `-h-file` is specified, leave the intermediate `.h` file there.

`-data-file`
If `-data-file` is specified, leave the intermediate `.data` file there.

`-system-p`
If `-system-p` is specified then invoke `compile-file` with the `:system-p t` keyword argument, meaning that the C init function will bear a name based on the name of the file, so that it may be invoked by name by C code.

9.2 Operating System Definitions

`GET-DECODED-TIME` () [Function]
Package:LISP

Returns the current time in decoded time format. Returns nine values: second, minute, hour, date, month, year, day-of-week, daylight-saving-time-p, and time-zone.

`HOST-NAMESTRING` (*pathname*) [Function]
Package:LISP

Returns the host part of PATHNAME as a string.

`RENAME-FILE` (*file new-name*) [Function]
Package:LISP

Renames the file FILE to NEW-NAME. FILE may be a string, a pathname, or a stream.

`FILE-AUTHOR` (*file*) [Function]
Package:LISP

Returns the author name of the specified file, as a string. FILE may be a string or a stream

`PATHNAME-HOST` (*pathname*) [Function]
Package:LISP

Returns the host slot of PATHNAME.

`FILE-POSITION` (*file-stream* **&optional** *position*) [Function]
Package:LISP

Sets the file pointer of the specified file to POSITION, if POSITION is given. Otherwise, returns the current file position of the specified file.

`DECODE-UNIVERSAL-TIME` (*universal-time* **&optional** *(timezone -9)*) [Function]
Package:LISP

Converts UNIVERSAL-TIME into a decoded time at the TIMEZONE. Returns nine values: second, minute, hour, date, month (1 - 12), year, day-of-week (0 - 6), daylight-saving-time-p, and time-zone. TIMEZONE in GCL defaults to 6, the time zone of Austin, Texas.

`USER-HOMEDIR-PATHNAME` (**&optional** *host*) [Function]
Package:LISP

Returns the home directory of the logged in user as a pathname. HOST is ignored.

`*MODULES*` [Variable]
Package:LISP A list of names of the modules that have been loaded into GCL.

`SHORT-SITE-NAME` () [Function]
Package:LISP

Returns a string that identifies the physical location of the current GCL.

`DIRECTORY` (*name*) [Function]
Package:LISP

Returns a list of files that match NAME. NAME may be a string, a pathname, or a file stream.

`SOFTWARE-VERSION` () [Function]
Package:LISP

Returns a string that identifies the software version of the software under which GCL is currently running.

`INTERNAL-TIME-UNITS-PER-SECOND` [Constant]
Package:LISP The number of internal time units that fit into a second.

`ENOUGH-NAMESTRING` (*pathname* **&optional** *(defaults *default-pathname-defaults*)*) [Function]
Package:LISP

Returns a string which uniquely identifies PATHNAME with respect to DEFAULTS.

`REQUIRE` (*module-name* **&optional** *(pathname)*) [Function]
Package:LISP

If the specified module is not present, then loads the appropriate file(s). PATHNAME may be a single pathname or it may be a list of pathnames.

`ENCODE-UNIVERSAL-TIME` (*second minute hour date month year* **&optional** *(timezone)*) [Function]
Package:LISP

Does the inverse operation of DECODE-UNIVERSAL-TIME.

`LISP-IMPLEMENTATION-VERSION` () [Function]
Package:LISP

Returns a string that tells you when the current GCL implementation is brought up.

`MACHINE-INSTANCE` () [Function]
Package:LISP

Returns a string that identifies the machine instance of the machine on which GCL is currently running.

`ROOM` (**&optional** *(x t)*) [Function]
Package:LISP

Displays information about storage allocation in the following format.

for each type class

the number of pages so-far allocated for the type class

the maximum number of pages for the type class

the percentage of used cells to cells so-far allocated

the number of times the garbage collector has been called to collect cells of the type class

the implementation types that belongs to the type class

the number of pages actually allocated for contiguous blocks

the maximum number of pages for contiguous blocks

the number of times the garbage collector has been called to collect contiguous blocks

the number of pages in the hole

the maximum number of pages for relocatable blocks

the number of times the garbage collector has been called to collect relocatable blocks

the total number of pages allocated for cells

the total number of pages allocated

the number of available pages

the number of pages GCL can use.

The number of times the garbage collector has been called is not shown, if the number is zero. The optional X is ignored.

`GET-UNIVERSAL-TIME ()` [Function]
Package:LISP

Returns the current time as a single integer in universal time format.

`GET-INTERNAL-RUN-TIME ()` [Function]
Package:LISP

Returns the run time in the internal time format. This is useful for finding CPU usage. If the operating system allows, a second value containing CPU usage of child processes is returned.

`*DEFAULT-PATHNAME-DEFAULTS*` [Variable]
Package:LISP The default pathname-defaults pathname.

`LONG-SITE-NAME ()` [Function]
Package:LISP

Returns a string that identifies the physical location of the current GCL.

`DELETE-FILE` (*file*) [Function]
Package:LISP Deletes FILE.

`GET-INTERNAL-REAL-TIME ()` [Function]
Package:LISP

Returns the real time in the internal time format. This is useful for finding elapsed time.

`MACHINE-TYPE ()` [Function]
Package:LISP

Returns a string that identifies the machine type of the machine on which GCL is currently running.

`TIME` [Macro]
Package:LISP

Syntax:

```
(time form)
```

Evaluates FORM and outputs timing statistics on *TRACE-OUTPUT*.

`SOFTWARE-TYPE ()` [Function]
Package:LISP

Returns a string that identifies the software type of the software under which GCL is currently running.

`LISP-IMPLEMENTATION-TYPE ()` [Function]
Package:LISP

Returns a string that tells you that you are using a version of GCL.

`SLEEP` (*n*) [Function]
Package:LISP

This function causes execution to be suspended for N seconds. N may be any non-negative, non-complex number.

`BREAK-ON-FLOATING-POINT-EXCEPTIONS` (**&key** *division-by-zero* floating-point-invalid-operation floating-point-overflow floating-point-underflow floating-point-inexact) Package:SI [Function]

Break on the specified IEEE floating point error conditions. With no arguments, report the exceptions currently trapped. Disable the break by setting the key to nil, e.g.

> (break-on-floaing-point-exceptions :division-by-zero t) (DIVISION-BY-ZERO)

> (break-on-floaing-point-exceptions) (DIVISION-BY-ZERO)

> (break-on-floaing-point-exceptions :division-by-zero nil) NIL

On some of the most common platforms, the offending instruction will be disassembled, and the register arguments looked up in the saved context and reported in as operands. Within the error handler, addresses may be disassembled, and other registers inspected, using the functions defined in gcl_fpe.lsp.

10 Structures

`DEFSTRUCT` [Macro]

Package:LISP

Syntax:

```
(defstruct
         {name | (name {:conc-name | (:conc-name prefix-string) |
                        :constructor | (:constructor symbol [lambda-list]) |
                        :copier | (:copier symbol) |
                        :predicate | (:predicate symbol) |
                        (:include symbol) |
                        (:print-function function) |
                        (:type {vector | (vector type) | list}) |
                        :named | (:static { nil | t})
                        (:initial-offset number)}*)}
         [doc]
         {slot-name |
          (slot-name [default-value-form] {:type type | :read-only flag}*) }*
         )
```

Defines a structure. The doc-string DOC, if supplied, is saved as a STRUCTURE doc and can be retrieved by (documentation 'NAME 'structure). STATIC is gcl specific and makes the body non relocatable.

See the files misc/rusage.lsp misc/cstruct.lsp, for examples of making a lisp structure correspond to a C structure.

`HELP` (**&optional** *symbol*) [Function]

Package:LISP

GCL specific: Prints the documentation associated with SYMBOL. With no argument, this function prints the greeting message to GCL beginners.

11 Iteration and Tests

`DO-EXTERNAL-SYMBOLS` [Macro]

Package:LISP

Syntax:

```
(do-external-symbols (var [package [result-form]])
          {decl}* {tag | statement}*)
```

Executes STATEMENTs once for each external symbol in the PACKAGE (which defaults to the current package), with VAR bound to the current symbol. Then evaluates RESULT-FORM (which defaults to NIL) and returns the value(s).

`DO*` [Special Form]

Package:LISP

Syntax:

```
(do* ({(var [init [step]])}*) (endtest {result}*)
          {decl}* {tag | statement}*)
```

Just like DO, but performs variable bindings and assignments in serial, just like LET* and SETQ do.

`DO-ALL-SYMBOLS` [Macro]

Package:LISP

Syntax:

```
(do-all-symbols (var [result-form]) {decl}* {tag | statement}*)
```

Executes STATEMENTs once for each symbol in each package, with VAR bound to the current symbol. Then evaluates RESULT-FORM (which defaults to NIL) and returns the value(s).

`YES-OR-NO-P` (**&optional** *(format-string nil)* **&rest** *args*) [Function]

Package:LISP

Asks the user a question whose answer is either 'YES' or 'NO'. If FORMAT- STRING is non-NIL, then FRESH-LINE operation is performed, a message is printed as if FORMAT-STRING and ARGs were given to FORMAT, and then a prompt "(Yes or No)" is printed. Otherwise, no prompt will appear.

`MAPHASH` *#'hash-table* [Function]

Package:LISP

For each entry in HASH-TABLE, calls FUNCTION on the key and value of the entry; returns NIL.

`MAPCAR` (*fun list* **&rest** *more-lists*) [Function]

Package:LISP

Applies FUN to successive cars of LISTs and returns the results as a list.

`DOLIST` [Special Form]

Package:LISP

Syntax:

```
(dolist (var listform [result]) {decl}* {tag | statement}*)
```

Executes STATEMENTs, with VAR bound to each member of the list value of LISTFORM. Then returns the value(s) of RESULT (which defaults to NIL).

`EQ` (*x y*) [Function]

Package:LISP

Returns T if X and Y are the same identical object; NIL otherwise.

`EQUALP` (*x y*) [Function]

Package:LISP

Returns T if X and Y are EQUAL, if they are characters and satisfy CHAR-EQUAL, if they are numbers and have the same numerical value, or if they have components that are all EQUALP. Returns NIL otherwise.

`EQUAL` (*x y*) [Function]

Package:LISP

Returns T if X and Y are EQL or if they are of the same type and corresponding components are EQUAL. Returns NIL otherwise. Strings and bit-vectors are EQUAL if they are the same length and have identical components. Other arrays must be EQ to be EQUAL.

`DO-SYMBOLS` [Macro]

Package:LISP

Syntax:

```
(do-symbols (var [package [result-form]]) {decl}* {tag |
statement}*)
```

Executes STATEMENTs once for each symbol in the PACKAGE (which defaults to the current package), with VAR bound to the current symbol. Then evaluates RESULT-FORM (which defaults to NIL) and returns the value(s).

`LOOP` [Special Form]

Package:LISP

Syntax:

```
(loop {form}*)
```

Executes FORMs repeatedly until exited by a THROW or RETURN. The FORMs are surrounded by an implicit NIL block.

12 User Interface

`-` [Special Variable]

Package:LISP Holds the top-level form that GCL is currently evaluating.

`-` (*number* **&rest** *more-numbers*) [Function]

Package:LISP

Subtracts the second and all subsequent NUMBERs from the first NUMBER. With one arg, negates it.

`UNTRACE` [Macro]

Package:LISP

Syntax:

```
(untrace {function-name}*)
```

Removes tracing from the specified functions. With no FUNCTION-NAMEs, untraces all functions.

`***` [Variable]

Package:LISP Gets the previous value of ** when GCL evaluates a top-level form.

`MAKE-STRING-INPUT-STREAM` (*string* **&optional** *(start 0) (end (length string)))* [Function]

Package:LISP

Returns an input stream which will supply the characters of String between Start and End in order.

`STEP` [Macro]

Package:LISP

Syntax:

```
(step form)
```

Evaluates FORM in the single-step mode and returns the value.

`*BREAK-ENABLE*` [Variable]

Package:LISP GCL specific: When an error occurrs, control enters to the break loop only if the value of this variable is non-NIL.

`/` [Special Variable]

Package:LISP Holds a list of the values of the last top-level form.

`DESCRIBE` (*x*) [Function]

Package:LISP

Prints a description of the object X.

`ED` (**&optional** *x*) [Function]

Package:LISP

Invokes the editor. The action depends on the version of GCL.

`*DEBUG-IO*` [Variable]
Package:LISP Holds the I/O stream used by the GCL debugger.

`*BREAK-ON-WARNINGS*` [Variable]
Package:LISP When the function WARN is called, control enters to the break loop only if the value of this varialbe is non-NIL.

`CERROR` (*continue-format-string error-format-string* **&rest** *args*) [Function]
Package:LISP

Signals a correctable error.

`**` [Variable]
Package:LISP Gets the previous value of * when GCL evaluates a top-level form.

`+++` [Special Variable]
Package:LISP Gets the previous value of `++` when GCL evaluates a top-level form.

`INSPECT` (*x*) [Function]
Package:LISP

Shows the information about the object X in an interactive manner

`//` [Special Variable]
Package:LISP Gets the previous value of / when GCL evaluates a top-level form.

`*TRACE-OUTPUT*` [Variable]
Package:LISP The trace output stream.

`++` [Special Variable]
Package:LISP Gets the previous value of `+` when GCL evaluates a top-level form.

`*ERROR-OUTPUT*` [Variable]
Package:LISP Holds the output stream for error messages.

`DRIBBLE` (**&optional** *pathname*) [Function]
Package:LISP

If PATHNAME is given, begins to record the interaction to the specified file. If PATHNAME is not given, ends the recording.

`*` [Variable]
Package:LISP Holds the value of the last top-level form.

`///` [Special Variable]
Package:LISP Gets the previous value of // when GCL evaluates a top-level form.

`WARN` (*format-string* **&rest** *args*) [Function]
Package:LISP

Formats FORMAT-STRING and ARGs to *ERROR-OUTPUT* as a warning message.

`BREAK` (**&optional** *(format-string nil)* **&rest** *args*) [Function]

Package:LISP

Enters a break loop. If FORMAT-STRING is non-NIL, formats FORMAT-STRING and ARGS to *ERROR-OUTPUT* before entering a break loop. Typing :HELP at the break loop will list the break-loop commands.

`+` [Special Variable]

Package:LISP Holds the last top-level form.

`TRACE` [Macro]

Package:LISP

Syntax:

```
(trace {function-name}*)
```

Traces the specified functions. With no FUNCTION-NAMEs, returns a list of functions currently being traced.

Additional Keywords are allowed in GCL with the syntax (trace {fn | (fn {:kw form}*)}*)

For each FN naming a function, traces that function. Each :KW should be one of the ones listed below, and FORM should have the corresponding form. No :KW may be given more than once for the same FN. Returns a list of all FNs now traced which weren't already traced.

EXAMPLE (Try this with your favorite factorial function FACT):

```
;; print entry args and exit values

(trace FACT)

;; Break coming out of FACT if the value is bigger than 1000.

(trace (fact :exit
     (progn
       (if (> (car values) 1000)(break "big result"))
       (car values))))

;; Hairy example:

;;make arglist available without the si:: prefix
(import 'si::arglist)

(trace (fact
        :DECLARATIONS
        ((in-string "Here comes input: ")
         (out-string "Here comes output: ")
         all-values
         (silly (+ 3 4)))
        :COND
        (equal (rem (car arglist) 2) 0)
```

```
:ENTRY
(progn
  (cond
   ((equal (car arglist) 8)
    (princ "Entering FACT on input 8!! ")
    (setq out-string "Here comes output from inside (FACT 8): "))
   (t
    (princ in-string)))
  (car arglist))
:EXIT
(progn
  (setq all-values (cons (car values) all-values))
  (princ out-string)
  (when (equal (car arglist) 8)
        ;; reset out-string
        (setq out-string "Here comes output: "))
  (cons 'fact values))
:ENTRYCOND
(not (= (car arglist) 6))
:EXITCOND
(not (= (car values) (* 6 (car arglist))))
:DEPTH
5))
```

Syntax is `:keyword` *form1* `:keyword` *form2* ...

`:declarations`

`DEFAULT: NIL`

FORM is ((var1 form1)(var2 form2)...), where the var_i are symbols distinct from each other and from all symbols which are similarly declared for currently traced functions. Each form is evaluated immediately. Upon any invocation of a traced function when not already inside a traced function call, each var is bound to that value of form .

`:COND`

`DEFAULT: T`

Here, FORM is any Lisp form to be evaluated (by EVAL) upon entering a call of FN, in the environment where si::ARGLIST is bound to the current list of arguments of FN. Note that even if the evaluation of FORM changes the value of SI::ARGLIST (e.g. by evaluation of (SETQ si::ARGLIST ...)), the list of arguments passed to FN is unchanged. Users may alter args passed by destructively modifying the list structure of SI::ARGLIST however. The call is traced (thus invoking the :ENTRYCOND and :EXITCOND forms, at least) if and only if FORM does not evaluate to NIL.

`:ENTRYCOND`

`DEFAULT: T`

This is evaluated (by EVAL) if the :COND form evaluates to non-NIL, both in an environment where SI::ARGLIST is bound to the current list

of arguments of FN. If non-NIL, the :ENTRY form is then evaluated and printed with the trace "prompt".

`:ENTRY`

`DEFAULT: (CONS (QUOTE x) SI::ARGLIST),`

where x is the symbol we call FN If the :COND and :ENTRYCOND forms evaluate to non-NIL, then the trace "prompt" is printed and then this FORM is evaluated (by EVAL) in an environment where SI::ARGLIST is bound to the current list of arguments of FN. The result is then printed.

`:EXITCOND`

`DEFAULT: T`

This is evaluated (by EVAL) in the environment described below for the :EXIT form. The :EXIT form is then evaluated and printed with the "prompt" if and only if the result here is non-NIL.

`:EXIT`

`DEFAULT: (CONS (QUOTE x) VALUES),`

where x is the symbol we call FN Upon exit from tracing a given call, this FORM is evaluated (after the appropriate trace "prompt" is printed), using EVAL in an environment where SI::ARGLIST is bound to the current list of arguments of FN and VALUES is bound to the list of values returned by FN (recalling that Common Lisp functions may return multiple values).

`:DEPTH`

`DEFAULT:  No depth limit`

FORM is simply a positive integer specifying the maximum nesting of traced calls of FN, i.e. of calls of FN in which the :COND form evaluated to non-NIL. For calls of FN in which this limit is exceeded, even the :COND form is not evaluated, and the call is not traced.

13 Doc

`APROPOS` (*string* **&optional** *(package nil)*) [Function]
Package:LISP

Prints those symbols whose print-names contain STRING as substring. If PACKAGE is non-NIL, then only the specified package is searched.

`INFO` (*string* **&optional** *(list-of-info-files *default-info-files*)*) [Function]
PACKAGE:SI

Find all documentation about STRING in LIST-OF-INFO-FILES. The search is done for STRING as a substring of a node name, or for STRING in the indexed entries in the first index for each info file. Typically that should be a variable and function definition index, if the info file is about a programming language. If the windowing system is connected, then a choice box is offered and double clicking on an item brings up its documentation.

Otherwise a list of choices is offered and the user may select some of these choices.

list-of-info-files is of the form

```
("gcl-si.info" "gcl-tk.info" "gcl.info")
```

The above list is the default value of *default-info-files*, a variable in the SI package. To find these files in the file system, the search path *info-paths* is consulted as is the master info directory `dir`.

see *Index *default-info-files*:: and *Index *info-paths*::. For example

```
(info "defun")

 0: DEFUN :(gcl-si.info)Special Forms and Functions.
 1: (gcl.info)defun.
Enter n, all, none, or multiple choices eg 1 3 : 1

Info from file /home/wfs/gcl-doc/gcl.info:
defun                                                                [Macro]
------------------------------------------------------------------------------
'Defun'  function-name lambda-list [[{declaration}* | documentation]]
...
```

would list the node `(gcl.info)defun`. That is the node entitled `defun` from the info file gcl.info. That documentation is based on the ANSI common lisp standard. The choice

```
DEFUN :(gcl-si.info)Special Forms and Functions.
```

refers to the documentation on DEFUN from the info file gcl-si.info in the node *Special Forms And Functions*. This is an index reference and only the part of the node which refers to `defun` will be printed.

```
(info "factor" '("maxima.info"))
```

would search the maxima info files index and nodes for `factor`.

`*info-paths*` [Variable]

Package SI:

A list of strings such as

```
'("" "/usr/info/" "/usr/local/lib/info/" "/usr/local/info/"
  "/usr/local/gnu/info/" )
```

saying where to look for the info files. It is used implicitly by `info`, see *Index info::.

Looking for maxima.info would look for the file maxima.info in all the directories listed in *info-paths*. If nto found then it would look for `dir` in the *info-paths* directories, and if it were found it would look in the `dir` for a menu item such as

```
* maxima: (/home/wfs/maxima-5.0/info/maxima.info).
```

If such an entry exists then the directory there would be used for the purpose of finding `maxima.info`

14 Type

`COERCE` (*x type*) [Function]
Package:LISP

Coerces X to an object of the type TYPE.

`TYPE-OF` (*x*) [Function]
Package:LISP

Returns the type of X.

`CONSTANTP` (*symbol*) [Function]
Package:LISP

Returns T if the variable named by SYMBOL is a constant; NIL otherwise.

`TYPEP` (*x type*) [Function]
Package:LISP

Returns T if X is of the type TYPE; NIL otherwise.

`COMMONP` (*x*) [Function]
Package:LISP

Returns T if X is a Common Lisp object; NIL otherwise.

`SUBTYPEP` (*type1 type2*) [Function]
Package:LISP

Returns T if TYPE1 is a subtype of TYPE2; NIL otherwise. If it could not determine, then returns NIL as the second value. Otherwise, the second value is T.

`CHECK-TYPE` [Macro]
Package:LISP

Syntax:

```
(check-type place typespec [string])
```

Signals an error, if the contents of PLACE are not of the specified type.

`ASSERT` [Macro]
Package:LISP

Syntax:

```
(assert test-form [({place}*) [string {arg}*]])
```

Signals an error if the value of TEST-FORM is NIL. STRING is an format string used as the error message. ARGs are arguments to the format string.

`DEFTYPE` [Macro]
Package:LISP

Syntax:

```
(deftype name lambda-list {decl | doc}* {form}*)
```

Defines a new type-specifier abbreviation in terms of an 'expansion' function (lambda lambda-list1 {decl}* {form}*) where lambda-list1 is identical to LAMBDA-LIST except that all optional parameters with no default value specified in LAMBDA-LIST defaults to the symbol '*', but not to NIL. When the type system of GCL encounters a type specifier (NAME arg1 ... argn), it calls the expansion function with the arguments arg1 ... argn, and uses the returned value instead of the original type specifier. When the symbol NAME is used as a type specifier, the expansion function is called with no argument. The doc-string DOC, if supplied, is saved as the TYPE doc of NAME, and is retrieved by (documentation 'NAME 'type).

`DYNAMIC-EXTENT` [Declaration]

Package:LISP Declaration to allow locals to be cons'd on the C stack. For example (defun foo (&rest l) (declare (:dynamic-extent l)) ...) will cause l to be a list formed on the C stack of the foo function frame. Of course passing L out as a value of foo will cause havoc. (setq x (make-list n)) (setq x (cons a b)) (setq x (list a b c ..)) also are handled on the stack, for dynamic-extent x.

15 GCL Specific

`SYSTEM` (*string*) [Function]

Package:LISP

GCL specific: Executes a Shell command as if STRING is an input to the Shell. Not all versions of GCL support this function. At least on POSIX systems, this call should return two integers represeting the exit status and any possible terminating signal respectively.

`*TMP-DIR*` [Variable]

Package:COMPILER GCL specific: Directory in which temporary "gazonk" files used by the compiler are to be created.

`*IGNORE-MAXIMUM-PAGES*` [Variable]

Package:SI GCL specific: Tells the GCL memory manager whether (non-NIL) or not (NIL) it should expand memory whenever the maximum allocatable pages have been used up.

`*OPTIMIZE-MAXIMUM-PAGES*` [Variable]

Package:SI

GCL specific: Tells the GCL memory manager whether to attempt to adjust the maximum allowable pages for each type to approximately optimize the garbage collection load in the current process. Defaults to T. Set to NIL if you care more about memory usage than runtime.

`MACHINE-VERSION ()` [Function]

Package:LISP

Returns a string that identifies the machine version of the machine on which GCL is currently running.

`BY ()` [Function]

Package:LISP

GCL specific: Exits from GCL.

`DEFCFUN` [Macro]

Package:LISP

Syntax:

```
(defcfun header n {element}*)
```

GCL specific: Defines a C-language function which calls Lisp functions and/or handles Lisp objects. HEADER gives the header of the C function as a string. Non-negative-integer is the number of the main stack entries used by the C function, primarily for protecting Lisp objects from being garbage-collected. Each ELEMENT may give a C code fragment as a string, or it may be a list ((symbol {arg}*) {place}*) which, when executed, calls the Lisp function named by SYMBOL with the specified arguments and saves the value(s) to the specified places. The DEFCFUN form has the above meanings only after compiled; The GCL interpreter simply ignores this form.

An example which defines a C function list2 of two arguments, but which calls the 'lisp' function CONS by name, and refers to the constant 'NIL. Note to be loaded by `load` the function should be static.

(defCfun "static object list2(x,y) object x,y;" 0 "object z;" ('NIL z) ((CONS y z) z) ((CONS x z) z) "return(z);")

In lisp the operations in the body would be (setq z 'nil) (setq z (cons y z)) (setq z (cons x z))

Syntax:

```
(defCfun header non-negative-integer
        { string
          | ( function-symbol { value }* )
          | (( function-symbol  { value }* ) { place }* ) })
```

```
value:
place:
         { C-expr | ( C-type C-expr ) }

C-function-name:
C-expr:
         { string | symbol }

C-type:
         { object | int | char | float | double }
```

`CLINES` [Macro]

Package:LISP

Syntax:

```
(clines {string}*)
```

GCL specific: The GCL compiler embeds STRINGs into the intermediate C language code. The interpreter ignores this form.

`ALLOCATE` (*type number* **&optional** *(really-allocate nil)*) [Function]

Package:LISP

GCL specific: Sets the maximum number of pages for the type class of the GCL implementation type TYPE to NUMBER. If REALLY-ALLOCATE is given a non-NIL value, then the specified number of pages will be allocated immediately.

`GBC` (*x*) [Function]

Package:LISP

GCL specific: Invokes the garbage collector (GC) with the collection level specified by X. NIL as the argument causes GC to collect cells only. T as the argument causes GC to collect everything.

`SAVE` (*pathname*) [Function]

Package:LISP

GCL specific: Saves the current GCL core image into a program file specified by PATHNAME. This function depends on the version of GCL. The function si::save-system is to be preferred in almost all circumstances. Unlike save, it makes the relocatable section permanent, and causes no future gc of currently loaded .o files.

`HELP*` (*string* **&optional** *(package 'lisp)*) [Function]

Package:LISP

GCL specific: Prints the documentation associated with those symbols in the specified package whose print names contain STRING as substring. STRING may be a symbol, in which case the print-name of that symbol is used. If PACKAGE is NIL, then all packages are searched.

`DEFLA` [Macro]

Package:LISP

Syntax:

```
(defla name lambda-list {decl | doc}* {form}*)
```

GCL specific: Used to DEFine Lisp Alternative. For the interpreter, DEFLA is equivalent to DEFUN, but the compiler ignores this form.

`PROCLAMATION` (*decl-spec*) [Function]

Package:LISP

GCL specific: Returns T if the specified declaration is globally in effect; NIL otherwise. See the doc of DECLARE for possible DECL-SPECs.

`DEFENTRY` [Macro]

Package:LISP

Syntax:

```
(defentry name arg-types c-function)
```

GCL specific: The compiler defines a Lisp function whose body consists of a calling sequence to the C language function specified by C-FUNCTION. The interpreter ignores this form. The ARG-TYPES specifies the C types of the arguments which C-FUNCTION requires. The list of allowed types is (object char int float double string). Code will be produced to coerce from a lisp object to the appropriate type before passing the argument to the C-FUNCTION. The c-function should be of the form (c-result-type c-fname) where c-result-type is a member of (void object char int float double string). c-fname may be a symbol (in which case it will be downcased) or a string. If c-function is not a list, then (object c-function) is assumed. In order for C code to be loaded in by `load` you should declare any variables and functions to be static. If you will link them in at build time, of course you are allowed to define new externals.

```
  Sample usage:
--File begin-----
;; JOE takes X a lisp string and Y a fixnum and returns a character.
(clines "#include \"foo.ch\"")
```

```
(defentry joe (string int) (char "our_c_fun"))
---File end------
---File foo.ch---
/* C function for extracting the i'th element of a string */
static char our_c_fun(p,i)
char *p;
int i;
   {
return p[i];
   }
-----File end---
```

One must be careful of storage allocation issues when passing a string. If the C code invokes storage allocation (either by calling `malloc` or `make_cons` etc), then there is a possibility of a garbage collection, so that if the string passed was not constructed with `:static t` when its array was constructed, then it could move. If the C function may allocate storage, then you should pass a copy:

```
(defun safe-c-string (x)
  (let* ((n (length x))
         (a (make-array (+ n 1) :element-type 'string-char
           :static t :fill-pointer n)))
    (si::copy-array-portion x y 0 0 n)
    (setf (aref a n) (code-char 0)))
    a)
```

`COPY-ARRAY-PORTION` (*x,y,i1,i2,n1*) [Function]

Package:SI Copy elements from X to Y starting at X[i1] to Y[i2] and doing N1 elements if N1 is supplied otherwise, doing the length of X - I1 elements. If the types of the arrays are not the same, this has implementation dependent results.

`BYE` (**&optional** *(exit-status 0)*) [Function]

Package:LISP

GCL specific: Exits from GCL with exit-status.

`USE-FAST-LINKS` (*turn-on*) [Function]

Package:LISP

GCL specific: If TURN-ON is not nil, the fast link mechanism is enabled, so that ordinary function calls will not appear in the invocation stack, and calls will be much faster. This is the default. If you anticipate needing to see a stack trace in the debugger, then you should turn this off.

15.1 Bignums

A directory mp was added to hold the new multi precision arithmetic code. The layout and a fair amount of code in the mp directory is an enhanced version of gpari version 34. The gpari c code was rewritten to be more efficient, and gcc assembler macros were added to allow inlining of operations not possible to do in C. On a 68K machine, this allows the C

version to be as efficient as the very carefully written assembler in the gpari distribution. For the main machines, an assembler file (produced by gcc) based on this new method, is included. This is for sites which do not have gcc, or do not wish to compile the whole system with gcc.

Bignum arithmetic is much faster now. Many changes were made to cmpnew also, to add 'integer' as a new type. It differs from variables of other types, in that storage is associated to each such variable, and assignments mean copying the storage. This allows a function which does a good deal of bignum arithmetic, to do very little consing in the heap. An example is the computation of PI-INV in scratchpad, which calculates the inverse of pi to a prescribed number of bits accuracy. That function is now about 20 times faster, and no longer causes garbage collection. In versions of GCL where HAVE_ALLOCA is defined, the temporary storage growth is on the C stack, although this often not so critical (for example it makes virtually no difference in the PI-INV example, since in spite of the many operations, only one storage allocation takes place.

Below is the actual code for PI-INV

On a sun3/280 (cli.com)

Here is the comparison of lucid and gcl before and after on that pi-inv. Times are in seconds with multiples of the gcl/akcl time in parentheses.

On a sun3/280 (cli.com)

```
pi-inv   akcl-566  franz        lucid         old kcl/akcl
----------------------------------------
10000      3.3     9.2(2.8 X)  15.3 (4.6X)    92.7   (29.5 X)
20000      12.7    31.0(2.4 X) 62.2 (4.9X)    580.0  (45.5 X)
```

```
(defun pi-inv (bits &aux (m 0))
  (declare (integer bits m))
  (let* ((n (+ bits (integer-length bits) 11))
         (tt (truncate (ash 1 n) 882))
         (d (* 4 882 882))
         (s 0))
    (declare (integer s d tt n))
    (do ((i 2 (+ i 2))
         (j 1123 (+ j 21460)))
        ((zerop tt) (cons s (- (+ n 2))))
      (declare (integer i j))
        (setq s (+ s (* j tt))
              m (- (* (- i 1) (- (* 2 i) 1) (- (* 2 i) 3)))
              tt (truncate (* m tt) (* d (the integer (expt i 3))))))))
```

16 C Interface

16.1 Available Symbols

When GCL is built, those symbols in the system libraries which are referenced by functions linked in in the list of objects given in `unixport/makefile`, become available for reference by GCL code.

On some systems it is possible with `faslink` to load `.o` files which reference other libraries, but in general this practice is not portable.

17 System Definitions

`ALLOCATE-CONTIGUOUS-PAGES` (*number* **&optional** *(really-allocate nil)*) [Function]

Package:SI

GCL specific: Sets the maximum number of pages for contiguous blocks to NUMBER. If REALLY-ALLOCATE is non-NIL, then the specified number of pages will be allocated immediately.

`FREEZE-DEFSTRUCT` (*name*) [Function]

Package:SI

The inline defstruct type checker will be made more efficient, in that it will only check for types which currently include NAME. After calling this the defstruct should not be altered.

`MAXIMUM-ALLOCATABLE-PAGES` (*type*) [Function]

Package:SI

GCL specific: Returns the current maximum number of pages for the type class of the GCL implementation type TYPE.

`ALLOCATED-RELOCATABLE-PAGES` () [Function]

Package:SI

GCL specific: Returns the number of pages currently allocated for relocatable blocks.

`PUTPROP` (*symbol value indicator*) [Function]

Package:SI

Give SYMBOL the VALUE on INDICATOR property.

`ALLOCATED-PAGES` (*type*) [Function]

Package:SI

GCL specific: Returns the number of pages currently allocated for the type class of the GCL implementation type TYPE.

`ALLOCATE-RELOCATABLE-PAGES` (*number*) [Function]

Package:SI

GCL specific: Sets the maximum number of pages for relocatable blocks to NUMBER.

`ALLOCATED-CONTIGUOUS-PAGES` () [Function]

Package:SI

GCL specific: Returns the number of pages currently allocated for contiguous blocks.

`MAXIMUM-CONTIGUOUS-PAGES` () [Function]

Package:SI

GCL specific: Returns the current maximum number of pages for contiguous blocks.

`GET-HOLE-SIZE` () [Function]

Package:SI

GCL specific: Returns as a fixnum the size of the memory hole (in pages).

`SPECIALP` (*symbol*) [Function]
Package:SI

GCL specific: Returns T if the SYMBOL is a globally special variable; NIL otherwise.

`OUTPUT-STREAM-STRING` (*string-output-stream*) [Function]
Package:SI

GCL specific: Returns the string corresponding to the STRING-OUTPUT-STREAM.

`GET-STRING-INPUT-STREAM-INDEX` (*string-input-stream*) [Function]
Package:SI

GCL specific: Returns the current index of the STRING-INPUT-STREAM.

`STRING-CONCATENATE` (**&rest** *strings*) [Function]
Package:SI

GCL specific: Returns the result of concatenating the given STRINGS.

`BDS-VAR` (*i*) [Function]
Package:SI

GCL specific: Returns the symbol of the i-th entity in the bind stack.

`ERROR-SET` (*form*) [Function]
Package:SI

GCL specific: Evaluates the FORM in the null environment. If the evaluation of the FORM has successfully completed, SI:ERROR-SET returns NIL as the first value and the result of the evaluation as the rest of the values. If, in the course of the evaluation, a non-local jump from the FORM is atempted, SI:ERROR-SET traps the jump and returns the corresponding jump tag as its value.

`COMPILED-FUNCTION-NAME` (*compiled-function-object*) [Function]
Package:SI

GCL specific: Returns the name of the COMPILED-FUNCTION-OBJECT.

`STRUCTUREP` (*object*) [Function]
Package:SI

GCL specific: Returns T if the OBJECT is a structure; NIL otherwise.

`IHS-VS` (*i*) [Function]
Package:SI

GCL specific: Returns the value stack index of the i-th entity in the invocation history stack.

`UNIVERSAL-ERROR-HANDLER` (*error-name correctable function-name continue-format-string error-format-string* **&rest** *args*) [Function]
Package:SI

GCL specific: Starts the error handler of GCL. When an error is detected, GCL calls SI:UNIVERSAL-ERROR-HANDLER with the specified arguments. ERROR-NAME is the name of the error. CORRECTABLE is T for a correctable error and NIL for a fatal error. FUNCTION-NAME is the name of the function that caused

the error. CONTINUE-FORMAT-STRING and ERROR-FORMAT-STRING are the format strings of the error message. ARGS are the arguments to the format strings. To change the error handler of GCL, redefine SI:UNIVERSAL-ERROR- HANDLER.

`*INTERRUPT-ENABLE*` [Variable]

Package:SI GCL specific: If the value of SI:*INTERRUPT-ENABLE* is non-NIL, GCL signals an error on the terminal interrupt (this is the default case). If it is NIL, GCL ignores the interrupt and assigns T to SI:*INTERRUPT-ENABLE*.

`CHDIR` (*pathname*) [Function]

Package:SI

GCL/UNIX specific: Changes the current working directory to the specified pathname.

`COPY-STREAM` (*in-stream out-stream*) [Function]

Package:SI

GCL specific: Copies IN-STREAM to OUT-STREAM until the end-of-file on IN-STREAM.

`INIT-SYSTEM` () [Function]

Package:SI

GCL specific: Initializes the library and the compiler of GCL. Since they have already been initialized in the standard image of GCL, calling SI:INIT- SYSTEM will cause an error.

`*INDENT-FORMATTED-OUTPUT*` [Variable]

Package:SI GCL specific: The FORMAT directive ~% indents the next line if the value of this variable is non-NIL. If NIL, ~% simply does Newline.

`SET-HOLE-SIZE` (*fixnum*) [Function]

Package:SI

GCL specific: Sets the size of the memory hole (in pages).

`FRS-BDS` (*i*) [Function]

Package:SI

GCL specific: Returns the bind stack index of the i-th entity in the frame stack.

`IHS-FUN` (*i*) [Function]

Package:SI

GCL specific: Returns the function value of the i-th entity in the invocation history stack.

`*MAKE-CONSTANT` (*symbol value*) [Function]

Package:SI

GCL specific: Makes the SYMBOL a constant with the specified VALUE.

`FIXNUMP` (*object*) [Function]

Package:SI

GCL specific: Returns T if the OBJECT is a fixnum; NIL otherwise.

`BDS-VAL` (*i*) [Function]
Package:SI

GCL specific: Returns the value of the i-th entity in the bind stack.

`STRING-TO-OBJECT` (*string*) [Function]
Package:SI

GCL specific: (SI:STRING-TO-OBJECT STRING) is equivalent to (READ-FROM-STRING STRING), but much faster.

`*SYSTEM-DIRECTORY*` [Variable]
Package:SI GCL specific: Holds the name of the system directory of GCL.

`FRS-IHS` (*i*) [Function]
Package:SI

GCL specific: Returns the invocation history stack index of the i-th entity in the frame stack.

`RESET-GBC-COUNT` () [Function]
Package:SI

GCL specific: Resets the counter of the garbage collector that records how many times the garbage collector has been called for each implementation type.

`CATCH-BAD-SIGNALS` () [Function]
Package:SI

GCL/BSD specific: Installs a signal catcher for bad signals: SIGILL, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS. The signal catcher, upon catching the signal, signals an error (and enter the break-level). Since the internal memory of GCL may be broken, the user should check the signal and exit from GCL if necessary. When the signal is caught during garbage collection, GCL terminates immediately.

`RESET-STACK-LIMITS` () [Function]
Package:SI

GCL specific: Resets the stack limits to the normal state. When a stack has overflowed, GCL extends the limit for the stack in order to execute the error handler. After processing the error, GCL resets the stack limit by calling SI:RESET-STACK-LIMITS.

`*GBC-MESSAGE*` [Variable]
Package:SI GCL specific: If the value of SI:*GBC-MESSAGE* is non-NIL, the garbage collector prints some information on the terminal. Usually SI:*GBC-MESSAGE* should be set NIL.

`*GBC-NOTIFY*` [Variable]
Package:SI GCL specific: If the value is non-NIL, the garbage collector prints a very brief one line message about the area causing the collection, and the time spent in internal time units.

`*AFTER-GBC-HOOK*` [Variable]

Package:SI Defaults to nil, but may be set to a function of one argument TYPE which is a lisp variable indicating the TYPE which caused the current collection.

`ALLOCATED` (*type*) [Funcition]

Package:SI

Returns 6 values:

nfree	number free
npages	number of pages
maxpage	number of pages to grow to
nppage	number per page
gbccount	number of gc's due to running out of items of this size
nused	number of items used

Note that all items of the same size are stored on similar pages. Thus for example on a 486 under linux the following basic types are all the same size and so will share the same allocated information: CONS BIGNUM RATIO COMPLEX STRUCTURE.

`*MAKE-SPECIAL` (*symbol*) [Function]

Package:SI

GCL specific: Makes the SYMBOL globally special.

`MAKE-STRING-OUTPUT-STREAM-FROM-STRING` (*string*) [Function]

Package:SI

GCL specific: Creates a string-output-stream corresponding to the STRING and returns it. The STRING should have a fill-pointer.

`*IGNORE-EOF-ON-TERMINAL-IO*` [Variable]

Package:SI GCL specific: If the value of SI:*IGNORE-EOF-ON-TERMINAL-IO* is non-NIL, GCL ignores the eof-character (usually ^D) on the terminal and the terminal never becomes end-of-file. The default value of SI:*IGNORE-EOF-ON-TERMINAL-IO* is NIL.

`ADDRESS` (*object*) [Function]

Package:SI

GCL specific: Returns the address of the OBJECT as a fixnum. The address of an object depends on the version of GCL. E.g. (SI:ADDRESS NIL) returns 1879062044 on GCL/AOSVS dated March 14, 1986.

`*LISP-MAXPAGES*` [Variable]

Package:SI GCL specific: Holds the maximum number of pages (1 page = 2048 bytes) for the GCL process. The result of changing the value of SI:*LISP-MAXPAGES* is unpredictable.

`ARGC` () [Function]

Package:SI

GCL specific: Returns the number of arguments on the command line that invoked the GCL process.

`NANI` (*fixnum*) [Function]
Package:SI

GCL specific: Returns the object in the address FIXNUM. This function is the inverse of SI:ADDRESS. Although SI:ADDRESS is a harmless operation, SI:NANI is quite dangerous and should be used with care.

`*NOTIFY-GBC*` [Variable]
Package:SI GCL specific: If the value of this variable is non-NIL, then the garbage collector notifies that it begins to run whenever it is invoked. Otherwise, garbage collection begins silently.

`SAVE-SYSTEM` (*pathname*) [Function]
Package:SI

GCL specific: Saves the current GCL core imange into a program file specified by PATHNAME. This function differs from SAVE in that the contiguous and relocatable areas are made permanent in the saved image. Usually the standard image of GCL interpreter/compiler is saved by SI:SAVE-SYSTEM. This function causes an exit from lisp. Various changes are made to the memory of the running system, such as closing files and resetting io streams. It would not be possible to continue normally.

`UNCATCH-BAD-SIGNALS` () [Function]
Package:SI

GCL/BSD specific: Undoes the effect of SI:CATCH-BAD-SIGNALS.

`VS` (*i*) [Function]
Package:SI

GCL specific: Returns the i-th entity in the value stack.

`DISPLACED-ARRAY-P` (*array*) [Function]
Package:SI

GCL specific: Returns T if the ARRAY is a displaced array; NIL otherwise.

`ARGV` (*fixnum*) [Function]
Package:SI

GCL specific: Returns the FIXNUM-th argument on the command line that invoked the GCL process.

`*DEFAULT-TIME-ZONE*` [Variable]
Package:SI GCL specific: Holds the default time zone. The initial value of SI:*DEFAULT- TIME-ZONE* is 6 (the time zone of Austin, Texas).

`GETENV` (*string*) [Function]
Package:SI

GCL/UNIX specific: Returns the environment with the name STRING as a string; if the environment specified by STRING is not found, returns NIL.

`FASLINK` (*file string*) [Function]

Package:SI

GCL/BSD specific: Loads the FASL file FILE while linking the object files and libraries specified by STRING. For example, (faslink "foo.o" "bar.o boo.o -lpixrect") loads foo.o while linking two object files (bar.o and boo.o) and the library pixrect. Usually, foo.o consists of the C language interface for the functions defined in the object files or the libraries.

A more portable way of making references to C code, is to build it in at the time of the original make. If foo.c references things in -lpixrect, and foo.o is its compilation in the gcl/unixport directory

(cd gcl/unixport ; make "EXTRAS= foo.o -lpixrect ")

should add them. If EXTRAS was already joe.o in the unixport/makefile you should of course add joe.o to the above "EXTRAS= joe.o foo.o.."

Faslink does not work on most UNIX systems which are derived from SYS V or AIX.

`TOP-LEVEL` () [Function]

Package:SI

GCL specific: Starts the standard top-level listner of GCL. When the GCL process is invoked, it calls SI:TOP-LEVEL by (FUNCALL 'SI:TOP-LEVEL). To change the top-level of GCL, redefine SI:TOP-LEVEL and save the core imange in a file. When the saved imange is invoked, it will start the redefined top-level.

`FRS-VS` (*i*) [Function]

Package:SI

GCL specific: Returns the value stack index of the i-th entity in the frame stack.

`WRITE-DEBUG-SYMBOLS` (*start file* **&key** *(main-file "/usr/local/schelter/xgcl/unixport/raw_gcl") (output-file "debug-symbols.o")*) [Function]

Package:SI

Write out a file of debug-symbols using address START as the place where FILE will be loaded into the running executable MAIN-FILE. The last is a keyword argument.

`PROF` (*x y*) [Function]

Package:SI

These functions in the SI package are GCL specific, and allow monitoring the run time of functions loaded into GCL, as well as the basic functions. Sample Usage: (si::set-up-profile 1000000) (si::prof 0 90) run program (si::prof 0 0) ;; turn off profile (si::display-prof) (si::clear-profile) (si::prof 0 90) ;; start profile again run program .. Profile can be stopped with (si::prof 0 0) and restarted with (si::prof 0 90) The START-ADDRESS will correspond to the beginning of the profile array, and the SCALE will mean that 256 bytes of code correspond to SCALE bytes in the profile array.

Thus if the profile array is 1,000,000 bytes long and the code segment is 5 megabytes long you can profile the whole thing using a scale of 50 Note that long runs may result in overflow, and so an understating of the time in a function.

You must run intensively however since, with a scale of 128 it takes 6,000,000 times through a loop to overflow the sampling in one part of the code.

`CATCH-FATAL` (*i*) [Function]

Package:SI

Sets the value of the C variable catch_fatal to I which should be an integer. If catch_fatal is 1, then most unrecoverable fatal errors will be caught. Upon catching such an error catch_fatal becomes -1, to avoid recursive errors. The top level loop automatically sets catch_fatal to 1, if the value is less than zero. Catching can be turned off by making catch_fatal = 0.

`*MULTIPLY-STACKS*` [Variable]

Package:SI

If this variable is set to a positive fixnum, then the next time through the TOP-LEVEL loop, the loop will be exited. The size of the stacks will be multiplied by the value of *multiply-stacks*, and the TOP-LEVEL will be called again. Thus to double the size of the stacks:

>(setq si::*multiply-stacks* 2) [exits top level and reinvokes it, with the new stacks in place] >

We must exit TOP-LEVEL, because it and any other lisp functions maintain many pointers into the stacks, which would be incorrect when the stacks have been moved. Interrupting the process of growing the stacks, can leave you in an inconsistent state.

`GBC-TIME` (**&optional** *x*) [Function]

Package:SI

Sets the internal C variable gc_time to X if X is supplied and then returns gc_time. If gc_time is greater or equal to 0, then gc_time is incremented by the garbage collector, according to the number of internal time units spent there. The initial value of gc_time is -1.

`FWRITE` (*string start count stream*) [Function]

Package:SI

Write from STRING starting at char START (or 0 if it is nil) COUNT characters (or to end if COUNT is nil) to STREAM. STREAM must be a stream such as returned by FP-OUTPUT-STREAM. Returns nil if it fails.

`FREAD` (*string start count stream*) [Function]

Package:SI

Read characters into STRING starting at char START (or 0 if it is nil) COUNT characters (or from start to length of STRING if COUNT is nil). Characters are read from STREAM. STREAM must be a stream such as returned by FP-INPUT-STREAM. Returns nil if it fails. Return number of characters read if it succeeds.

`SGC-ON` (**&optional** *ON*) [Function]

Package:SI

If ON is not nil then SGC (stratified garbage collection) is turned on. If ON is supplied and is nil, then SGC is turned off. If ON is not supplied, then it returns T if SGC is on, and NIL if SGC is off.

The purpose of SGC is to prevent paging activity during garbage collection. It is efficient if the actual number of pages being written to form a small percentage of the total image size. The image should be built as compactly as possible. This can be accomplished by using a settings such as (si::allocate-growth 'cons 1 10 50 20) to limit the growth in the cons maxpage to 10 pages per time. Then just before calling si::save-system to save your image you can do something like:

(si::set-hole-size 500)(gbc nil) (si::sgc-on t) (si::save-system ..)

This makes the saved image come up with SGC on. We have set a reasonably large hole size. This is so that allocation of pages either because they fill up, or through specific calls to si::allocate, will not need to move all the relocatable data. Moving relocatable data requires turning SGC off, performing a full gc, and then turning it back on. New relocatable data is collected by SGC, but moving the old requires going through all pages of memory to change pointers into it.

Using si::*notify-gbc* gives information about the number of pages used by SGC.

Note that SGC is only available on operating systems which provide the mprotect system call, to write protect pages. Otherwise we cannot tell which pages have been written too.

`ALLOCATE-SGC` (*type min-pages max-pages percent-free*) [Function]
Package:SI

If MIN-PAGES is 0, then this type will not be swept by SGC. Otherwise this is the minimum number of pages to make available to SGC. MAX-PAGES is the upper limit of such pages. Only pages with PERCENT-FREE objects on them, will be assigned to SGC. A list of the previous values for min, max and percent are returned.

`ALLOCATE-GROWTH` (*type min max percent percent-free*) [Function]
Package:SI

The next time after a garbage collection for TYPE, if PERCENT-FREE of the objects of this TYPE are not actually free, and if the maximum number of pages for this type has already been allocated, then the maximum number will be increased by PERCENT of the old maximum, subject to the condition that this increment be at least MIN pages and at most MAX pages. A list of the previous values for min, max, percent, and percent-free for the type TYPE is returned. A value of 0 means use the system default, and if an argument is out of range then the current values are returned with no change made.

Examples: (si::allocate-growth 'cons 1 10 50 10) would insist that after a garbage collection for cons, there be at least 10% cons's free. If not the number of cons pages would be grown by 50% or 10 pages which ever was smaller. This might be reasonable if you were trying to build an image which was 'full', ie had few free objects of this type.

(si::allocate-growth 'fixnum 0 10000 30 40) would grow space till there were normally 40% free fixnums, usually growing by 30% per time.

(si::allocate-growth 'cons 0 0 0 40) would require 40% free conses after garbage collection for conses, and would use system defaults for the the rate to grow towards this goal.

(si::allocate-growth 'cons -1 0 0 0) would return the current values, but not make any changes.

`OPEN-FASD` (*stream direction eof-value table*) [Function]

Package:SI

Given file STREAM open for input or output in DIRECTION, set it up to start writing or reading in fasd format. When reading from this stream the EOF-VALUE will be returned when the end a fasd end of dump marker is encountered. TABLE should be an eq hashtable on output, a vector on input, or nil. In this last case a default one will be constructed.

We shall refer to the result as a 'fasd stream'. It is suitable as the arg to CLOSE-FASD, READ-FASD-TOP, and as the second second arg to WRITE-FASD. As a lisp object it is actually a vector, whose body coincides with:

struct fasd { object stream; /* lisp object of type stream */ object table; /* hash table used in dumping or vector on input*/ object eof; /* lisp object to be returned on coming to eof mark */ object direction; /* holds Cnil or Kinput or Koutput */ object package; /* the package symbols are in by default */ object index; /* integer. The current_dump index on write */ object filepos; /* nil or the position of the start */ object table_length; /* On read it is set to the size dump array needed or 0 */ object macro ; }

We did not use a defstruct for this, because we want the compiler to use this and it makes bootstrapping more difficult. It is in "cmpnew/fasdmacros.lsp"

`WRITE-FASD-TOP` (*X FASD-STREAM*) [Function]

Package:SI

Write X to FASD-STREAM.

`READ-FASD-TOP` (*FASD-STREAM*) [Function]

Package:SI

Read the next object from FASD-STREAM. Return the eof-value of FASD-STREAM if we encounter an eof marker put out by CLOSE-FASD. Encountering end of actual file stream causes an error.

`CLOSE-FASD` (*FASD-STREAM*) [Function]

Package:SI

On output write an eof marker to the associated file stream, and then make FASD-STREAM invalid for further output. It also attempts to write information to the stream on the size of the index table needed to read from the stream from the last open. This is useful in growing the array. It does not alter the file stream, other than for writing this information to it. The file stream may be reopened for further use. It is an error to OPEN-FASD the same file or file stream again with out first calling CLOSE-FASD.

`FIND-SHARING-TOP` (*x table*) [Function]

Package:SI

X is any lisp object and TABLE is an eq hash table. This walks through X making entries to indicate the frequency of symbols,lists, and arrays. Initially items get -1

when they are first met, and this is decremented by 1 each time the object occurs. Call this function on all the objects in a fasd file, which you wish to share structure.

`*LOAD-PATHNAME*` [Variable]
Package:SI Load binds this to the pathname of the file being loaded.

`DEFINE-INLINE-FUNCTION` (*fname vars* **&body** *body*) [Macro]
Package:SI

This is equivalent to defun except that VARS may not contain &optional, &rest, &key or &aux. Also a compiler property is added, which essentially saves the body and turns this into a let of the VARS and then execution of the body. This last is done using si::DEFINE-COMPILER-MACRO Example: (si::define-inline-function myplus (a b c) (+ a b c))

`DEFINE-COMPILER-MACRO` (*fname vars* **&body** *body*) [Macro]
Package:SI

FNAME may be the name of a function, but at compile time the macro expansion given by this is used.

(si::define-compiler-macro mycar (a) ‘(car ,a))

`DBL` () [Function]
Package:SI

Invoke a top level loop, in which debug commands may be entered. These commands may also be entered at breaks, or in the error handler. See SOURCE-LEVEL-DEBUG

`NLOAD` (*file*) [Function]
Package:SI

Load a file with the readtable bound to a special readtable, which permits tracking of source line information as the file is loaded. see SOURCE-LEVEL-DEBUG

`BREAK-FUNCTION` (*function* **&optional** *line absolute*) [Function]
Package:SI

Set a breakpoint for a FUNCTION at LINE if the function has source information loaded. If ABSOLUTE is not nil, then the line is understood to be relative to the beginning of the buffer. See also dbl-break-function, the emacs command.

`XDR-OPEN` (*stream*) [Function]
Package:SI

Returns an object suitable for passing to XDR-READ if the stream is an input stream, and XDR-WRITE if it was an output stream. Note the stream must be a unix stream, on which si::fp-input-stream or si::fp-output-stream would act as the identity.

`FP-INPUT-STREAM` (*stream*) [Function]
Package:SI

Return a unix stream for input associated to STREAM if possible, otherwise return nil.

`FP-OUTPUT-STREAM` (*stream*) [Function]

Package:SI

Return a unix stream for output associated to STREAM if possible, otherwise return nil.

`XDR-READ` (*stream element*) [Function]

Package:SI

Read one item from STREAM of type the type of ELEMENT. The representation of the elements is machine independent. The xdr routines are what is used by the basic unix rpc calls.

`XDR-WRITE` (*stream element*) [Function]

Package:SI

Write to STREAM the given ELEMENT.

`*TOP-LEVEL-HOOK*` [Variable]

Package:SI If this variable is has a function as its value at start up time, then it is run immediately after the init.lsp file is loaded. This is useful for starting up an alternate top level loop.

`RUN-PROCESS` (*string arglist*) [Function]

Package:SI

Execute the command STRING in a subshell passing the strings in the list ARGLIST as arguments to the command. Return a two way stream associated to this. Use si::fp-output-stream to get an associated output stream or si::fp-input-stream.

Bugs: It does not properly deallocate everything, so that it will fail if you call it too many times.

`*CASE-FOLD-SEARCH*` [Variable]

Package: SI Non nil means that a string-match should ignore case

`STRING-MATCH` (*pattern string* **&optional** *start end*) [Function]

Package: SI Match regexp PATTERN in STRING starting in string starting at START and ending at END. Return -1 if match not found, otherwise return the start index of the first matchs. The variable *MATCH-DATA* will be set to a fixnum array of sufficient size to hold the matches, to be obtained with match-beginning and match-end. If it already contains such an array, then the contents of it will be over written.

The form of a regexp pattern is discussed in See ⟨undefined⟩ [Regular Expressions], page ⟨undefined⟩.

`MATCH-BEGINNING` (*index*) [Function]

Returns the beginning of the I'th match from the previous STRING-MATCH, where the 0th is for the whole regexp and the subsequent ones match parenthetical expressions. -1 is returned if there is no match, or if the *match-data* vector is not a fixnum array.

`MATCH-END` (*index*) [Function]

Returns the end of the I'th match from the previous STRING-MATCH

`SOCKET` (*port* **&key** *host server async myaddr myport daemon*) [Function]

Establishes a socket connection to the specified PORT under a variety of circumstances.

If HOST is specified, then it is a string designating the IP address of the server to which we are the client. ASYNC specifies that the connection should be made asynchronously, and the call return immediately. MYADDR and MYPORT can specify the IP address and port respectively of a client connection, for example when the running machine has several network interfaces.

If SERVER is specified, then it is a function which will handle incoming connections to this PORT. DAEMON specifies that the running process should be forked to handle incoming connections in the background. If DAEMON is set to the keyword PERSISTENT, then the backgrounded process will survive when the parent process exits, and the SOCKET call returns NIL. Any other non-NIL setting of DAEMON causes the socket call to return the process id of the backgrounded process. DAEMON currently only works on BSD and Linux based systems.

If DAEMON is not set or nil, or if the socket is not a SERVER socket, then the SOCKET call returns a two way stream. In this case, the running process is responsible for all I/O operations on the stream. Specifically, if a SERVER socket is created as a non-DAEMON, then the running process must LISTEN for connections, ACCEPT them when present, and call the SERVER function on the stream returned by ACCEPT.

`ACCEPT` (*stream*) [Function]

Creates a new two-way stream to handle an individual incoming connection to STREAM, which must have been created with the SOCKET function with the SERVER keyword set. ACCEPT should only be invoked when LISTEN on STREAM returns T. If the STREAM was created with the DAEMON keyword set in the call to SOCKET, ACCEPT is unnecessary and will be called automatically as needed.

17.1 Regular Expressions

The function `string-match` (*Index string-match::) is used to match a regular expression against a string. If the variable `*case-fold-search*` is not nil, case is ignored in the match. To determine the extent of the match use *Index match-beginning:: and *Index match-end::.

Regular expressions are implemented using Henry Spencer's package (thank you Henry!), and much of the description of regular expressions below is copied verbatim from his manual entry. Code for delimited searches, case insensitive searches, and speedups to allow fast searching of long files was contributed by W. Schelter. The speedups use an adaptation by Schelter of the Boyer and Moore string search algorithm to the case of branched regular expressions. These allow such expressions as 'not_there|really_not' to be searched for 30 times faster than in GNU emacs (1995), and 200 times faster than in the original Spencer method. Expressions such as [a-u]bcdex get a speedup of 60 and 194 times respectively. This is based on searching a string of 50000 characters (such as the file tk.lisp).

- A regular expression is a string containing zero or more *branches* which are separated by `|`. A match of the regular expression against a string is simply a match of the string with one of the branches.
- Each branch consists of zero or more *pieces*, concatenated. A matching string must contain an initial substring matching the first piece, immediately followed by a second substring matching the second piece and so on.
- Each piece is an *atom* optionally followed by `+`, `*`, or `?`.
- An atom followed by `+` matches a sequence of 1 or more matches of the atom.
- An atom followed by `*` matches a sequence of 0 or more matches of the atom.
- An atom followed by `?` matches a match of the atom, or the null string.
- An atom is
  - a regular expression in parentheses matching a match for the regular expression
  - a *range* see below
  - a `.` matching any single character
  - a `^` matching the null string at the beginning of the input string
  - a `$` matching the null string at the end of the input string
  - a `\` followed by a single character matching that character
  - a single character with no other significance (matching that character).
- A *range* is a sequence of characters enclosed in `[]`. It normally matches any single character from the sequence.
  - If the sequence begins with `^`, it matches any single character *not* from the rest of the sequence.
  - If two characters in the sequence are separated by `-`, this is shorthand for the full list of ASCII characters between them (e.g. `[0-9]` matches any decimal digit).
  - To include a literal `]` in the sequence, make it the first character (following a possible `^`).
  - To include a literal `-`, make it the first or last character.

Ordering Multiple Matches

In general there may be more than one way to match a regular expression to an input string. For example, consider the command

```
(string-match "(a*)b*"  "aabaaabb")
```

Considering only the rules given so far, the value of (list-matches 0 1) might be `("aabb" "aa")` or `("aaab" "aaa")` or `("ab" "a")` or any of several other combinations. To resolve this potential ambiguity **string-match** chooses among alternatives using the rule *first then longest*. In other words, it considers the possible matches in order working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

[1] If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

[2] If a regular expression contains `|` operators then the leftmost matching sub-expression is chosen.

[3] In *****, `+`, and **?** constructs, longer matches are chosen in preference to shorter ones.

[4] In sequences of expression components the components are considered from left to right.

In the example from above, **(a*)b*** matches **aab**: the **(a*)** portion of the pattern is matched first and it consumes the leading **aa**; then the **b*** portion of the pattern consumes the next **b**. Or, consider the following example:

```
(string-match "(ab|a)(b*)c"  "xabc") ==> 1
(list-matches 0 1 2 3) ==> ("abc" "ab" "" NIL)
(match-beginning 0) ==> 1
(match-end 0) ==> 4
(match-beginning 1) ==> 1
(match-end 1) ==> 3
(match-beginning 2) ==> 3
(match-end 2) ==> 3
(match-beginning 3) ==> -1
(match-end 3) ==> -1
```

In the above example the return value of `1` (which is `> -1`) indicates that a match was found. The entire match runs from 1 to 4. Rule 4 specifies that **(ab|a)** gets first shot at the input string and Rule 2 specifies that the **ab** sub-expression is checked before the **a** sub-expression. Thus the **b** has already been claimed before the **(b*)** component is checked and **(b*)** must match an empty string.

The special characters in the string `"\()[]+.*|^$?"`, must be quoted, if a simple string search is desired. The function re-quote-string is provided for this purpose.

```
(re-quote-string "*standard*") ==> "\\*standard\\*"

(string-match (re-quote-string "*standard*") "X *standard* ")
 ==> 2

(string-match "*standard*" "X *standard* ")
Error: Regexp Error: ?+* follows nothing
```

Note there is actually just one `\` before the `*` but the printer makes two so that the string can be read, since `\` is also the lisp quote character. In the last example an error is signalled since the special character `*` must follow an atom if it is interpreted as a regular expression.

18 Debugging

18.1 Source Level Debugging in Emacs

In emacs load (load "dbl.el") from the gcl/doc directory. [It also requires gcl.el from that directory. Your system administrator should do make in the doc directory, so that these files are copied to the standard location.]

OVERVIEW:

Lisp files loaded with si::nload will have source line information about them recorded. Break points may be set, and functions stepped. Source code will be automatically displayed in the other window, with a little arrow beside the current line. The backtrace (command :bt) will show line information and you will get automatic display of the source as you move up and down the stack.

FUNCTIONS: break points which have been set. si::nload (file) load a lisp file collecting source line information.

si::break-function (function &optional line absolute) set up a breakpoint for FUNCTION at LINE relative to start or ABSOLUTE

EMACS COMMANDS: M-x dbl makes a dbl buffer, suitable for running an inferior gcl. It has special keybindings for stepping and viewing sources. You may start your favorite gcl program in the dbl shell buffer.

Inferior Dbl Mode: Major mode for interacting with an inferior Dbl process. The following commands are available:

C-c l dbl-find-line

ESC d dbl-:down ESC u dbl-:up ESC c dbl-:r ESC n dbl-:next ESC i dbl-:step ESC s dbl-:step

M-x dbl-display-frame displays in the other window the last line referred to in the dbl buffer.

ESC i and ESC n in the dbl window, call dbl to step and next and then update the other window with the current file and position.

If you are in a source file, you may select a point to break at, by doing C-x SPC.

Commands: Many commands are inherited from shell mode. Additionally we have:

M-x dbl-display-frame display frames file in other window ESC i advance one line in program ESC n advance one line in program (skip over calls). M-x send-dbl-command used for special printing of an arg at the current point. C-x SPACE sets break point at current line.

When visiting a lisp buffer (if gcl.el is loaded in your emacs) the command c-m-x evaluates the current defun into the process running in the other window. Line information will be kept. This line information allows you to set break points at a given line (by typing C-x \space on the line in the source file where you want the break to occur. Once stopped within a function you may single step with M-s. This moves one line at a time in the source code, displaying a little arrow beside your current position. M-c is like M-s, except that function invocations are skipped over, rather than entered into. M-c continues execution.

Keywords typed at top level, in the debug loop have a special meaning:

:delete [n1] [n2] .. – delete all break points or just n1,n2

:disable [n1] [n2] .. – disable all break points or just n1,n2

:enable [n1] [n2] .. – enable all break points or just n1,n2

:info [:bkpt] –print information about

:break [fun] [line] – break at the current location, or if fun is supplied in fun. Break at the beginning unless a line offset from the beginning of fun is supplied.

:fr [n] go to frame n When in frame n, if the frame is interpreted, typing the name of locals, will print their values. If it is compiled you must use (si::loc j) to print 'locj'. Autodisplay of the source will take place if it is interpreted and the line can be determined.

:up [n] go up n frames from the current frame.

:down [n] go down n frames

:bt [n] back trace starting at the current frame and going to top level If n is specified show only n frames.

:r If stopped in a function resume. If at top level in the dbl loop, exit and resume an outer loop.

:q quit the computation back to top level dbl loop.

:step step to the next line with line information

:next step to the next line with line information skipping over function invocations.

Files: debug.lsp dbl.el gcl.el

18.2 Low Level Debug Functions

Use the following functions to directly access GCL stacks.

```
(SI:VS i) Returns the i-th entity in VS.
(SI:IHS-VS i) Returns the VS index of the i-th entity in IHS.
(SI:IHS-FUN i) Returns the function of the i-th entity in IHS.
(SI:FRS-VS i) Returns the VS index of the i-th entity in FRS.
(SI:FRS-BDS i) Returns the BDS index of the i-th entity in FRS.
(SI:FRS-IHS i) Returns the IHS index of the i-th entity in FRS.
(SI:BDS-VAR i) Returns the symbol of the i-th entity in BDS.
(SI:BDS-VAL i) Returns the value of the i-th entity in BDS.

(SI:SUPER-GO i tag)
Jumps to the specified tag established by the TAGBODY frame at
FRS[i].  Both arguments are evaluated.  If FRS[i] happens to be
a non-TAGBODY frame, then (THROW (SI:IHS-TAG i) (VALUES)) is
performed.
```

19 Miscellaneous

19.1 Environment

The environment in GCL which is passed to macroexpand and other functions requesting an environment, should be a list of 3 lists. The first list looks like ((v1 val1) (v2 val2) ..) where vi are variables and vali are their values. The second is a list of ((fname1 . fbody1) (fname2 . fbody2) ...) where fbody1 is either (macro lambda-list lambda-body) or (lambda-list lambda-body) depending on whether this is a macro or a function. The third list contains tags and blocks.

19.2 Initialization

If the file init.lsp exists in the current directory, it is loaded at startup. The first argument passed to the executable image should be the system directory. Normally this would be gcl/unixport. This directory is stored in the si::*system-directory* variable. If the file sys-init.lsp exists in the system directory, it is loaded before init.lsp. See also si::*TOP-LEVEL-HOOK*.

19.3 Low Level X Interface

A sample program for drawing things on X windows from lisp is included in the file gcl/lsp/littleXlsp.lsp

That routine invokes the corresponding C routines in XLIB. So in order to use it you must 'faslink' in the X routines. Directions are given at the beginning of the lisp file, for either building them into the image or using faslink.

This program is also a good tutorial on invoking C from lisp.

See also defentry and faslink.

20 Compiler Definitions

`EMIT-FN` (*turn-on*) [Function]

Package:COMPILER

If TURN-ON is t, the subsequent calls to COMPILE-FILE will cause compilation of foo.lisp to emit a foo.fn as well as foo.o. The .fn file contains cross referencing information as well as information useful to the collection utilities in cmpnew/collectfn This latter file must be manually loaded to call emit-fn.

`*CMPINCLUDE-STRING*` [Variable]

Package:COMPILER If it is a string it holds the text of the cmpinclude.h file appropriate for this version. Otherwise the usual #include of *cmpinclude* will be used. To disable this feature set *cmpinclude-string* to NIL in the init-form.

`EMIT-FN` (*turn-on*) [Function]

Package:COMPILER

If TURN-ON is t, then subsequent calls to compile-file on a file foo.lisp cause output of a file foo.fn. This .fn file contains lisp structures describing the functions in foo.lisp. Some tools for analyzing this data base are WHO-CALLS, LIST-UNDEFINED-FUNCTIONS, LIST-UNCALLED-FUNCTIONS, and MAKE-PROCLAIMS.

Usage: (compiler::emit-fn t) (compile-file "foo1.lisp") (compile-file "foo2.lisp")

This would create foo1.fn and foo2.fn. These may be loaded using LOAD. Each time compile-file is called the data base is cleared. Immediately after the compilation, the data base consists of data from the compilation. Thus if you wished to find functions called but not defined in the current file, you could do (list-undefined-functions), immediately following the compilation. If you have a large system, you would load all the .fn files before using the above tools.

`MAKE-ALL-PROCLAIMS` (**&rest** *directories*) [Function]

Package:COMPILER

For each D in DIRECTORIES all files in (directory D) are loaded.

For example (make-all-proclaims "lsp/*.fn" "cmpnew/*.fn") would load any files in lsp/*.fn and cmpnew/*.fn.

[See EMIT-FN for details on creation of .fn files]

Then calculations on the newly loaded .fn files are made, to determine function proclamations. If number of values of a function cannot be determined [for example because of a final funcall, or call of a function totally unknown at this time] then return type * is assigned.

Finally a file sys-proclaim.lisp is written out. This file contains function proclamations.

(load "sys-proclaim.lisp") (compile-file "foo1.lisp") (compile-file "foo2.lisp")

`MAKE-PROCLAIMS` (**&optional** *(stream *standard-output*)*) [Function]

Package:COMPILER

Write to STREAM the function proclaims from the current data base. Usually a number of .fn files are loaded prior to running this. See EMIT-FN for details on how to collect this. Simply use LOAD to load in .fn files.

`LIST-UNDEFINED-FUNCTIONS ()` [Function]

Package:COMPILER

Return a list of all functions called but not defined, in the current data base (see EMIT-FN).

```
Sample:
(compiler::emit-fn t)
(compile-file "foo1.lisp")
(compiler::list-undefined-functions)
or
(mapcar 'load (directory "*.fn")) (compiler::list-undefined-functions)
```

`WHO-CALLS` (*function-name*) [Function]

Package:COMPILER

List all functions in the data base [see emit-fn] which call FUNCTION-NAME.

`LIST-UNCALLED-FUNCTIONS ()` [Function]

Package:COMPILER

Examine the current data base [see emit-fn] for any functions or macros which are called but are not: fboundp, OR defined in the data base, OR having special compiler optimizer properties which would eliminate an actual call.

`*CC*` [Variable]

Package:COMPILER Has value a string which controls which C compiler is used by GCL. Usually this string is obtained from the machine.defs file, but may be reset by the user, to change compilers or add an include path.

`*SPLIT-FILES*` [Variable]

Package:COMPILER This affects the behaviour of compile-file, and is useful for cases where the C compiler cannot handle large C files resulting from lisp compilation. This scheme should allow arbitrarily long lisp files to be compiled.

If the value [default NIL] is a positive integer, then the source file will be compiled into several object files whose names have 0,1,2,.. prepended, and which will be loaded by the main object file. File 0 will contain compilation of top level forms thru position *split-files* in the lisp source file, and file 1 the next forms, etc. Thus a 180k file would probably result in three object files (plus the master object file of the same name) if *split-files* was set to 60000. The package information will be inserted in each file.

`*COMPILE-ORDINARIES*` [Variable]

Package:COMPILER If this has a non nil value [default = nil], then all top level forms will be compiled into machine instructions. Otherwise only defun's, defmacro's, and top level forms beginning with (progn 'compile ...) will do so.

Appendix A Function and Variable Index

(Index is nonexistent)

Short Contents

Table of Contents