LISP

|  |  |
| --- | --- |
| **Overview**  **Data**   * Linked lists are the primary data structure. Name came from LISt Processor. * Data and programs can be stored identically as linked lists. * Data types are determined during execution. Declaration of variables for data type or memory size isn't necessary * LISP isn't case sensitive * More recent versions of LISP support hash tables and objects   **Bindings**   * Many bindings are made at execution time.   **Storage Mgt**   * allocated implicitly * allocation of memory is done through use * no explicit free * garbage collection   **Scope**   * Nonlocals use either static or dynamic scope. //you decide   **Parameter Passing**   * **Functions pass parameters by value** * Macros and special functions pass unevaluated arguments //Assignment3   **Operations**   * Syntax is prefix notation using parentheses to surround the arguments causing it to earn many nicknames including "Lost In Stupid Parentheses".   + Every "statement" in LISP is a function call.   + (*function arg1 arg2 …*)   **Sequence Control**   * Recursion is a major looping approach | LISP was the first functional language. Avoiding the use of assignments (SETx functions), code can be free of side effects; therefore, it is easier to prove correctness of code.  LISP is the second oldest language still in use. Only FORTRAN is older. It pioneered:   * dynamic storage * recursion * binary-tree data structures * interpretive language * garbage collection   At UTSA, we use GNU CLISP. (See the CS3723 set up page for more information or [click here](http://www.cs.utsa.edu/~clark/cs3723/LispSetUp.htm).)  We have a GNU CLISP on VDI which will allow you to execute LISP code while in class. Click on the Start button to get the list of all applications. Select GNU CLISP.  You might like using Notepad++ to enter your LISP code. It does a good job of matching parentheses. |
| **Data Syntax Terminology**  **Atom** symbol, numeric constant, string literal, NIL  **List** parentheses around one or more atoms or lists  **quote** quoting a symbol or list makes it unevaluated  VARIABLES NOT CASE SENSITIVE  When evaluating a value, if it is a list, the first item is the name of the function and the subsequent items are arguments. Unless the first item is special or a macro, the arguments are evaluated and passed by value.  The setf function is special. It doesn't evaluate its first argument.  (setf *variable expression*) | Atoms:  x, y, total, "hello world", 123, 5.8  Lists:  (x y z) //function x passing y and z  (w (x y) z)  (print "hello world")  Quote:  (print '(x y z)) //quote= list contains atoms x , y ,z  (print '(w (x y) z))  Output:  (X Y Z)  (W (X Y) Z)  Evaluating:  (setf x 'orange)  (setf y 'apple)  (print (list x y)) // list = function  Output:  (ORANGE APPLE) |
| **List Accessing Functions**  (**car** *list*) - returns the first item in the list  (**cdr** *list*) - returns a list of items after the first item.  (**nth** *n**list*) - returns the nth item in the list relative to 0.  // 0 = index of list  (**cadr** *list*) - returns the (car (cdr *list*)) *//car of the cdr*  (**cddr** *list*) - returns the (cdr (cdr *list*)) // cdr of cdr  Most LISPs supprt at least 3 levels of those CAR/CDR functions (e.g., CADDR, CDADR). Many support four levels.  Historical note:  Based on the IBM 704, **CAR** got its name from the Contents of the Address portion of the Register. **CDR** got is name from the Contents of the Decrement part of the Register. | > (car '(X Y Z))  X  > (car (X Y Z))  Error: X is not a function  > (car '(LISP is fun))  LISP  > (car '( (x) y z))  (X)  > (car '( (w x) y z))  (W X)  > (car '()) // car of empty list// empty list and NIL synonymous.  NIL (in older LISPs this caused an error)  > (car NIL)  NIL  > (cdr '(x y z))  (Y Z)  > (cdr '( (w x) y z)) //list Y Z  (Y Z)  > (cdr '())  NIL  > (cdr '(x y))  (Y)  > (cdr '(x (y z)))// list of the list Y Z  ((Y Z))  > (nth 0 '(x y z))  X  > (nth 2 '(x y z))  Z  > (nth 3 '(x y z))  NIL  > (cadr '(x y z)) //car of the cdr  Y  > (cddr '(x y z)) //cdr of the cdr  (Z) |
| **Functions Creating Lists**  (**cons** *expr**list*) - returns a new list with its car being *expr* and its cdr being *list. //cons = construct*  (**append** *list1**list2*) - returns a new list with the items from *list1* followed by the items of *list2*.  (**list** *expr1**expr2*) - returns a new list with the value of *expr1* followed by the value of *expr2*. | > (cons 'x '(y z))  (X Y Z)  > (cons '(x) '(y z))  ((X) Y Z))  > (cons () '(y z))  (() Y Z)) or shown as (NIL Y Z)  > (cons 'x ())  (X) - this is important in building lists  > (append '(Q R S) '(X Y Z)) //takes off parenthesis after S and before X  (Q R S X Y Z)  > (append 'X '(Y Z))  Error: X is not a list  > (list 'x 'y)  (X Y)  > (list 'x '(y))  (X (Y))  > (list '(x) '(y))  ((X) (Y))  > (APPEND (LIST ‘X) ‘(Y Z)  (X Y Z) |
| **Predicate Functions**  (**ATOM** *expr*) - returns T if *expr* is an ATOM. Otherwise, NIL.  (**NULL** *expr*) - returns T if *expr* is NIL.  (**EQL** *expr1 expr2*) - returns T if the expressions are   * the same symbol * numeric values are the same type and values * two lists have the same address. //tests/compare the addresses //use EQUAL FOR VALUES   Note: use **EQL** when comparing atoms!  (**EQUAL** *expr1 expr2*) - returns T like EQL; however, it also compares lists element by element. The shape and values must match. EQL performs better than EQUAL. | > (ATOM 'x)  T  > (ATOM NIL)  T  > (ATOM '(X Y)) //nil- for null pointer, empty list  NIL  > (ATOM 5)  T  > (ATOM "hello world") //string literal = atom  T  > (NULL NIL)  T  > (NULL 'x)  NIL  > (EQL 'x 'x)  T  > (EQL 'x 'y)  NIL  > (EQL 'X "X")  NIL ;;; 'X is a symbol, "X" is a string  > (EQL 4 (+ 2 2))  T  > (EQL 4 (+ 2.0 2)) //different data types so NIL  NIL  > (EQL '(X Y) '(X Y)) //constructed two different lists- nil  NIL  > (EQUAL '(X Y) '(X Y))  T  > (EQUAL '(X Y) '(Y X)) // not same order  NIL  > (SETF A '(X Y))  (X Y)  > (SETF B A)  (X Y) ;;; B and A point to the same list  > (EQL A B)  T  NIL = FALSE/ F |
| **Predicate Functions (continued)**  (**NUMBERP** *expr*) - returns T if *expr* is a NUMBER. Otherwise, NIL.  (**ZEROP** *expr*) - returns T if *expr* is zero. Otherwise, NIL.  (**<** *num1 num2*) - returns T if the num1 < num2; otherwise NIL.  There are also functions for **>, >=,** and **<=.** | > (NUMBERP 'X) //checking symbol x, not the value  NIL  > (NUMBERP 5)  T  >(SETF X 4)  >(NUMBERP X)  T  > (< 4 (+ 8 10))  T |
| **Logic Functions //first thing to do – does it short circuit**  (**AND** *expr1 expr2 ...*)  Evaluates each *exprk* in order left to right. If any expression is NIL, it short circuits and returns NIL. Otherwise, it returns T.  // looking for non-nil values//considered to be true  (**OR** *expr1 expr2 ...*)  Evaluates each *exprk* in order left to right. If any expression is T, it short circuits and returns T. Otherwise, it returns NIL.  (**NOT** *expr*)  Evaluates the expression. If it is NIL, NOT returns T. If it isn't NIL, NOT returns NIL. | > (setf lastName 'prince) //strings, not symbols//spaces are delimiters  > (setf age 24)  > (and (EQL lastName 'king) //short circuits here, doesn’t bother cont.  (< age 26))  NIL  > (or ((EQL lastName 'king)  (< age 26))  T  > (not lastName) //Is lastName nil? No? Then nil.  NIL  > (not NIL)  T |
| **Numeric Functions**  (**+** *num1 num2 …*) - returns the sum of the arguments  (**\*** *num1 num2 …*) - returns the product of the arguments  (**-** *num1 num2*) - returns the difference of *num1* and *num2*  (**/** *num1 num2*) - returns the quotient of *num1* and *num2*  (1**+** *num*) - returns the sum of 1 + *num* | > (setf width 5 length 4)  4  > (setf perimeter (+ (\* 2 width) (\* 2 length)))  18 |
| **Conditional Functions**  (**if** *condExpr exprTrue exprFalse*) - if *condExpr* is not-NIL, *exprTrue* is evaluated and returned; otherwise, *exprFalse* is evaluated and returned.  //very simple, use cond for else if’s  (**cond (***condExpr1 resExpr1*) // conditional  **(***condExpr2 resExpr2*)  *…*  **(***condExprN resExprN*))  COND is passed unevaluated pairs of conditional expressions and resulting expressions. It evaluates each conditional expression in order until it encounters one which is non-NIL and then returns the corresponding resulting expression. If all the conditional expressions are NIL, it returns NIL.  Most of my examples will use COND instead of IF because it supports multiple cases and I learned LISP using COND. | > (setf a 10)  10  > (if (eql a 10) 'x 'y)  X  > (cond ( (eql a 10) 'x )  ( T 'y ) )// if else statement structure  X  > (setf b 5)  5  (cond (( eql a b) (+ a b) //will evaluate first and unless nil will then evaluate the next  > (cond ( (eql a b) 'equal ) //if a = b print equal  ( (< a b) 'less ) // else if a less than b print less  ( T 'greater ) )  GREATER |
| **Defining Functions**  (**defun** *funcName* (*parmList*) *bodyExpr*)  Define a function named *funcName.* *parmList* is a list of zero or more formal parameters. *bodyExpr* is the expression evaluated as the body of the function.  When the defined function is used, its arguments are passed into it **by value**. // arguments are being evaluated every time | > (defun FIRST (L) // list is called L  (CAR L) // prints first element in list  )  FIRST  > (defun SECOND (L) // only defining, not invoking functions  (CADR L) // car of the cdr (second function)  )  SECOND  > (defun MAX2 (x y)  (cond ( (> x y) x ) // if x > y return VALUE of x  ( T y )  )  )  MAX2  > (SETF A 10 B 5)  (MAX2 A B) // invoking function above  10  > (MAX2 A (\* B 5))  25 |
| **Using Recursion**  Recursion is an important looping approach for LISP. Many functions will be a COND listing the cases. The first case is the base case. | ;;; factorial of N  > (defun FACTORIAL (n)  (cond ( (ZEROP N) 1 ) // if zero return 1  ( T (\* N  (FACTORIAL (- N 1))  )  )  )  ) |
| **Exercise:** count the number of top-level elements in a list.   * (cardinality '(x y z)) is 3 * (cardinality '( (x y) x)) is 2 * (cardinality '() ) is 0 | > (defun CARDINALITY (L)  (COND ( (NULL L) 0)  ( T (1+ (CARDINALITY (CDR L)))  )  )  Same function but in C:  Int cardinality ( Node \*p)  {  If (p == NULL)  Return 0;  Else  Return 1 + (cardinality (p->pNext));  } |
| **Exercise**: get the last item in a list,   * We get the rest of the list using CDR. * Recursively taking the CDR will eventually reach a (). * If the CDR is NIL, return the CAR //AT THE LAST ELEMENT SO YOU WANT TO RETURN THE CAR( SINCE ITS NOW THE FIRST) * We will name this function LAST1 | > (defun LAST1 (L)  (COND ((NULL L) NIL)  ((NULL (CDR L)) (CAR L)  (T (LAST1 (CDR L) )  )  )    > (LAST1 ' (A B D))  D |
| **Exercise:** see if an atom is a top-level member of a set   * (memset 'x '(x y z)) is T * (memset 'w '(x y z)) is NIL | > (defun MEMSET (atm L) // binary tree like traversal  (cond ( (NULL L) NIL )  ( (EQL atm (CAR L)) T )  ( T (MEMSET atm (CDR L)) )  )  )  Practice tracing using Larry's style for (MEMSET 'A '(B A R T)) |
| **Coding Functions where we must traverse to both the CAR and the CDR**  **Exercise:** see if an atom is anywhere in a list   * (memall 'x '( (Y X) Z)) is T * Don't just take the CAR. We must check for ATOM | > (defun MEMALL (atm L)  (cond ( (NULL L) NIL )  ( (ATOM L) (EQL atm L) )  ( T (OR (MEMALL atm (CAR L) )  (MEMALL atm (CDR L) ) ) )  )  )  Practice tracing using Larry's style for (MEMALL 'A '(B (A R) T)) |
| **Coding Functions Which Must Construct a List**  **Exercise:** remove the last top-level element in a list   * (remlast '(x y z)) returns (x y) * Use CONS to construct a list of the (CAR L) with the REMLAST of the (CDR L). | > (defun REMLAST (L)  (cond ( (NULL L) NIL )  ( (NULL (CDR L)) NIL )  ( T (CONS (CAR L) (REMLAST (CDR L)) ) )  )  )  Practice tracing using Larry's style. |
| **Example:** code the function, **SETUNION,** which is passed two lists that are simply high-level sets. The result has entries from both lists; however, an entry which is in both sets, only appears once in the SETUNION.   * (setunion '(x y z) '(v w x z)) is (y v w x z) * Approach (recursively traversing set1):   + See if a high-level item of set1 is in set2. Did we create a function for that?   + If it isn't in set2, cons it to the result.   + After we have traversed all of set1, include set2 in the result. | > (defun SETUNION (set1 set2)  (cond ( (NULL set1) set2 )  ( (MEMSET (CAR set1) set2)  (SETUNION (CDR set1) set2)  )  ( T (CONS (CAR set1)  (SETUNION (CDR set1) set2)  )  )  )  )  > (setunion '(x y z) '(v w x z)) |
| **Exercise:** code the function, **SETINTER,** which is passed two lists that are simply high-level sets. The result has entries which are the intersection of the two sets.   * (setinter '(x y z) '(v w x z)) is (x z) * Approach (recursively traversing set1):   + See if a high-level item of set1 is in set2.   + If not, don't include it in the result, but continue with the rest of set1.   + If it is, cons it to the result.   + After we have traversed all of set1, return NIL. | > (defun SETINTER (set1 set2)  (cond ( (NULL set1) NIL )  ( (MEMSET (CAR set1) set2)  (CONS ( CAR set1)  (SETINTER (CDR set1) set2) ) )  ( T (SETINTER (CDR set1) set2) )  )  )  > (setinter '(x y z) '(v w x z)) |
| **Exercise:** code the function, **REPTOP,** which is passed two atoms and a list. It replaces every top-level occurrence of the first argument in the list with the second argument.   * (reptop 'i 'o '(m i s s i s s i p p i)) is (m o s s o s s o p p o) * (reptop 'i 'o '(m i s s (i s s) i p p i) is (m o s s (i s s) o p p o) | > (defun REPTOP (match rep L)  (COND ( (NULL L) NIL)  ( ( EQL (CAR L) match)  (CONS rep (REPTOP match rep (CDR L)))  )  (T (CONS (CAR L) (REPTOP match rep (CDR L)))  )  )  )  > (reptop 'i 'o '(m i s s i s s i p p i))  > (reptop 'i 'o '(m i s s (i s s) i p p i)) |
| **Exercise:** code the function, **TOTALTOP**, which totals the numeric values at the top-level in a list. Assume the list only has numeric values at the top-level.   * (totaltop '(5 10 15)) is 30 | > (defun TOTALTOP (L)  (COND ( ( NULL L) 0)  ( T (+ (CAR L)  (TOTALTOP (CDR L)) ))  )  ) |
| **Exercise:** code the function, **TOTALALL,** which totals the numeric values at any level in a list.   * (totalall '(( 1 2 ) () ((5 10) 15)) is 33 | > (defun TOTALALL (L)  (COND (( NULL L) 0)  ( ( NUMBERP (CAR L))  (+ (CAR L) (TOTALALL( CDR L))) )  (T (+ ( TOTALALL (CAR L))  ( TOTALALL (CDR L)) ) )  )  )  Practice tracing using Larry's style for (TOTALALL '(1 (2 3) 4)) |