INTRODUCTION

- The primary data structure is a list, e.g.,
  \[(A B C D E)\]
  \[(A)\]
  \[(\text{empty list or } \texttt{NIL} = \text{a special name})\]

- Can represent any entities
  \[x + y \quad (\text{PLUS } x \ y)\]
  First item is operator, remaining items are operands
  Can have an arbitrary number of arguments
  \[xy + x + 3 \quad (\text{PLUS } (\text{TIMES } x \ y) \ x \ 3)\]

- We can refer to elements of a list by using brackets:
  for \(L = (\text{PLUS } (\text{TIMES } x \ y) \ x \ 3)\) we have
  \[L[1] = \text{PLUS}\]
  \[L[2] = (\text{TIMES } x \ y)\]
  \[L[2,2] = x\]
  \[L[4] = 3\]
  \[(\exists x) \ (\forall y) \ P(x) \supset P(y)\]
  \[\text{WEAK} \quad (\exists x) \ (\forall y) \ P(x) \supset P(y)\]

- An undirected graph

- Questions:
  1. How would we represent it?
  2. What do we want to know?
  3. What node is connected to what node?

- Solution: list of lists where first element of each
  list is connected to rest
  \[(\text{(A B) (B A C D) (C B D E) (D B C E) (E C D F') (F E)}})\]
REPRESENTATION OF A LIST

• Components of lists can be *atoms*
  1. any sequence of characters not including spaces or parentheses
  2. examples: x y 345 A37 A-B-C
     376-80-5763 80.8 ...

• How would we represent a list?

• In earlier work we used:

\[
\begin{array}{c}
\text{INFO} \\
\end{array} \rightarrow \begin{array}{c}
\text{INFO} \\
\end{array} \rightarrow \begin{array}{c}
\text{INFO} \\
\end{array} \rightarrow \ldots \rightarrow \begin{array}{c}
\Omega \\
\end{array}
\]

where \( \Omega = \text{NIL} \) or \( \begin{array}{c} \text{NIL} \end{array} \)...

\((A \ B \ C)\) would be:

\[
\begin{array}{c}
A \\
\end{array} \rightarrow \begin{array}{c}
B \\
\end{array} \rightarrow \begin{array}{c}
C \\
\end{array} \rightarrow \begin{array}{c}
\Omega \\
\end{array}
\]

• What about \( xy+x+3 \) or \( \text{PLUS} (\text{TIMES} \ x \ y) \ x \ 3 \) ?

• Solution: \( \text{INFO} \) points to another list!

\[
\begin{array}{c}
\text{PLUS} \\
\end{array} \rightarrow \begin{array}{c}
\text{X} \\
\end{array} \rightarrow \begin{array}{c}
3 \\
\end{array} \rightarrow \begin{array}{c}
\text{NIL} \\
\end{array}
\]

\[
\begin{array}{c}
\text{TIMES} \\
\end{array} \rightarrow \begin{array}{c}
\text{X} \\
\end{array} \rightarrow \begin{array}{c}
\text{Y} \\
\end{array}
\]

Copyright © 2003 by Hanan Samet
OBSERVATIONS ABOUT LISTS

• There is really no need for INFO field

• There are two link fields, say LLINK and RLINK

• INFO is now an atom, which is a link to a property list
  1. value of the atom
  2. print name

• Notation
  1. use lower-case letters at the end of the alphabet (e.g., x, y, z) to describe variables and upper-case letters at the start of the alphabet (e.g., A, B, C, D) to denote data
  2. atom represented by address of its property list
  3. list referred to by address of its first element

• Note a curious asymmetry:
  1. LLINK can refer to atom or list, but
  2. RLINK can only refer to a list or the empty list (equivalent to the atom NIL)
S-EXPRESSIONS

- An atom or a pair of s-expressions separated by . and surrounded by parentheses

\[
<\text{sexpr}> \Rightarrow <\text{atom}> | ( <\text{sexpr}> . <\text{sexpr}> )
\]

- Examples:

\[
\text{A} \\
\text{(A.B)} \\
\text{(A.(B.A))} \\
\text{(3 . 3.4)} \text{ note convention about decimal point and dot: . . is not a decimal point. Space around dot may be omitted if no confusion results:}
\]

\[
\text{(PLUS.(x.(y.NIL)))}
\]

- Represented in computer memory by:

\[
\text{(PLUS x y)}
\]
THE LISP PROGRAMMING LANGUAGE

• Easy to learn – just a few primitive operations
  1. **CAR** (Contents of Address Register)
     • first element of list
     • sometimes called *head*
     • sometimes written as \(a x\)
     • refers to left part of an s-expression
  2. **CDR** (Contents of Decrement Register)
     • remainder of list after removing first element
     • sometimes called *tail*
     • sometimes written as \(d x\)
     • refers to right part of an s-expression
     • pushes left paren one element to right
       \[\text{CDR of } (A \ (B \ C)) \rightarrow (B \ C)\]
     • CDR (and CAR) technically undefined for atoms
     • sometimes CDR of atom is its property list
  3. **QUOTE** prevents the usual evaluation of arguments
     Notationally the following are equivalent:
     • \((\text{CDR} (\text{QUOTE} \ (A \ B \ C)))\)
     • \((\text{CDR} \ ' (A \ B \ C))\)
     • \(\text{CDR} (\ ' (A \ B \ C))\)
     • \(\text{CDR}[ ' (A \ B \ C) ]\)
     • \(\text{CDR}[ (\text{QUOTE} \ (A \ B \ C)) ]\)
     • use \([\ ]\) when args quoted or in definition of recursive function, use \((\ )\) otherwise
  4. **CONS** (CONStruct)
     • creates an s-expression from two s-expressions
     • alternatively, adds atom or list to head of another list
     **Ex:** \(\text{CONS}[ 'A, ' (B \ C \ D)] \equiv (A \ B \ C \ D) \equiv \text{CONS}[ 'A, ' (B \ (C \ (D.NIL)))]\)
       \((A \ (B \ (C \ (D.NIL))))\)
LISP EXAMPLES

\[(A.B) \quad (C.D) \quad \text{CONS}['(A.B),'(C.D)] = \]

\[\text{CAR}['((A.B).(C.D))'] = (A.B)\]
\[\text{CAR}[\text{CAR } '((A.B).(C.D))'] = A\]
\[\text{CAAR}['((A.B).(C.D))'] = A\]
- note use of CAAR for CAR( CAR(x) )
- also CADR(x) = CAR( CDR(x) )
- CDR is performed first followed by CAR
- can construct any combination needed

\[\text{CONS}['(A.B),'A] = ((A.B).A)\]
\[\text{CONS}['A,'(B C D)] = (A B C D)\]

Important: \[\text{CAR}[\text{CONS}['A,'B]] = A\]
\[\text{CDR}[\text{CONS}['A,'B]] = B\]
\[\text{CONS}[	ext{CAR}['(A.B)],\text{CDR}['(A.B)]] = (A.B)\]
SHARING OF LISTS

• Lists may be shared:

\[
\text{is the same as } ((A.B).(C.((A.B).(D.NIL))))
\]

which can also be represented as:

• Difference is that given \( z \leftarrow (\text{CONS } 'A 'B) \) then:

\[
x \leftarrow (\text{CONS } z (\text{CONS } 'C (\text{CONS } z (\text{CONS } 'D \text{ NIL}))))
\]

i.e.,

\[
y \leftarrow (\text{CONS} (\text{CONS } 'A 'B) (\text{CONS } 'C (\text{CONS} (\text{CONS } 'A 'B) (\text{CONS } 'D \text{ NIL}))))
\]
STRUCTURAL EQUIVALENCE

• Can we test to see if any sharing exists? (for example, if first and third elements of x identical?)
  1. the EQ predicate performs this test
     \[ EQ[\text{CAR}(x), \text{CADDR}(x)] = \text{T} \] atom denoting value True
     \[ EQ[\text{CAR}(y), \text{CADDR}(y)] = \text{NIL} \] just like False
  2. atoms are uniquely represented
     \[ EQ[\text{CADR}(x), \text{CADR}(y)] = \text{T} \] while
     \[ EQ[\text{CAR}(x), \text{CAR}(y)] = \text{NIL} \] and
     \[ EQ[\text{CAAR}(x), \text{CAAR}(y)] = \text{T} \]
     =\[ EQ[\text{CDAR}(x), \text{CDAR}(y)] \]

• s-expressions x and y are structurally equivalent
  1. can we write function EQUAL to test for this?
  2. smallest indivisible unit is the atom
     base case: \[ \text{atom}(x) \Rightarrow \text{atom}(y) \]
     otherwise NIL
     • if either x or y are atoms, then \[ EQ(x, y) \]
     otherwise \[ \text{EQUAL} \] first parts and \[ \text{EQUAL} \] second parts

3. need way to find out if something is an atom
   • use the ATOM function
4. thus \[ \text{EQUAL}[x, y] = \]

\[
\begin{align*}
\text{if ATOM}(x) \text{ or ATOM}(y) \text{ then eq}(x, y) \\
\text{else } \text{EQUAL}[\text{CAR}(x), \text{CAR}(y)] \text{ and } \\
\text{EQUAL}[\text{CDR}(x), \text{CDR}(y)]
\end{align*}
\]

• this should be familiar from our discussion of similarity and equivalence of binary trees
COMBINATIONS OF LISP PRIMITIVES

- Three primitive functions: CAR CDR CONS
- Two primitive predicates: ATOM EQ
- Predicate is just function returning either NIL or non-NIL
- All other functions are combinations of these five primitives

Example:

\[
\text{EQUAL}(x, y) \\
\text{NULL}(x) \text{ which is } \text{EQ}(x, \text{NIL}) \text{ also written as } n_x
\]

- Other abbreviations:
  \[
  a_x \text{ for } \text{CAR}(x) \\
  a_d_x \text{ for } \text{CAR}(\text{CDR}(x)) \\
  x.y \text{ for } \text{CONS}(x, y)
  \]

- The LIST function
  1. takes arbitrary number of arguments and returns a list containing these arguments
  2. Ex: \( \text{LIST}(x, y, z) \) is \( (x \ y \ z) \)
  3. corresponds to composition of CONS operations
     \[
     \text{LIST}(x) \text{ is } \text{CONS}(x, \text{NIL}) \\
     \text{LIST}(x, y) \text{ is } \text{CONS}(x, \text{CONS}(y, \text{NIL}))
     \]
  4. also written as \( <x, y, z> \)
REPRESENTING TREES

• Linear:

```
L -> A -> B -> C -> D
```

• More tree-like representation: $L_1$

```
A -> A
```

note all information appears at terminal nodes only

• More balanced:

```
L_2 -> B -> C
```

only two ops to get to any particular node

• In unbalanced representation:

\[
\begin{align*}
\text{CAR}(L_1) &= A \\
\text{CADR}(L_1) &= B \\
\text{CADDR}(L_1) &= C \\
\text{CADDDR}(L_1) &= D \\
\text{average of } \frac{1+2+3+4}{4} &= 2.5 \text{ operations}
\end{align*}
\]

• In balanced representation:

\[
\begin{align*}
\text{CAAR}(L_2) &= A \\
\text{CDAR}(L_2) &= B \\
\text{CADR}(L_2) &= C \\
\text{CDDR}(L_2) &= D \\
\text{average of } 2 \text{ operations}
\end{align*}
\]

• Advantage of $L_1$: if searching for a particular element and list is not a fixed size, then we know when to stop
MEMBERSHIP IN LIST

• How would we search for $x$ in list $L_1$?

1. base case:
   how do we know when we are done?

   check for null list: $\text{if } \text{null } \text{then } \text{nil}$
   check first element: $\text{if } \text{first element } \text{eq } x \text{ then } \text{T}$

2. induction:

   check rest of list: $\text{dl}$

   • $\text{member}[x, \text{dl}]$

3. $\text{member}[x, L] = \text{if } \text{null } \text{then } \text{nil}$
   $\text{else if } \text{first element } \text{eq } x \text{ then } \text{T}$
   $\text{else } \text{member}[x, \text{rest of list}]$

• How to write function in LISP?

• Need to assign a function body to the function name
  $(\text{DEF } \text{fname} \ (\text{LAMBDA } (\text{arg1 arg2...argn}) \ \text{fbody}))$

• For example:

  $\text{member}[x, L] =$

  $(\text{DEF MEMBER} \ (\text{LAMBDA } X \ L))$
  $(\text{COND} \ ((\text{NULL } L) \ \text{NIL})$
  $((\text{EQ } X \ (\text{CAR } L)) \ \text{T})$
  $(\text{T} \ (\text{MEMBER } X \ (\text{CDR } L))))$
MEMBERSHIP IN S-EXPRESSION

• How would we search for $x$ in s-expression $s$?

• Analogous to searching terminal nodes of a tree

\[
\text{membersexpr}[x,s] = \\
\text{if ats then } x \text{ eq } s \\
\text{else membersexpr}[x,as] \text{ or membersexpr}[x,ds]
\]

• Base case is a node corresponding to atom

• Otherwise, check left subtree followed by right subtree

• Observations on the LISP s-expression tree:
  1. tree is being traversed in preorder
  2. information is only stored in terminal nodes
  3. each non-leaf node contains two pointers, \text{CAR} and \text{CDR}, to left and right subtrees, respectively

• Can we search for occurrence of an entire s-expression?

• What is the terminating case (or cases)?

\[
\text{atom } s \implies x \text{ eq } s \\
\text{not atom } s \implies x \text{ equal } s \text{ or } \\
\quad \text{members}[x,al] \text{ or members}[x,dl]
\]

\[
\text{members}[x,s] = \\
\text{if ats then } x \text{ eq } s \\
\text{else } x \text{ equal } s \text{ or } \\
\quad \text{members}[x,al] \text{ or members}[x,dl]
\]

• Note use of \text{EQUAL} to check equality of s-expressions because we want to test for equivalent substructures i.e., same terminal atomic nodes
ALTERNATIVE LIST REPRESENTATIONS

• Suppose we organize list by \texttt{CDR} instead of by \texttt{CAR}?
  1. What is lisp representation of this list: \( L_3 \)

2. Work backwards:
   - \( \text{CDR} (L_3) = A \)
   - \( \text{CDAR} (L_3) = B \)
   - \( \text{CDAAR} (L_3) = C \)
   - \( \text{CDAAAR} (L_3) = D \)

• Circular structures
  1. a list \emph{could} point back to component of itself
     - \( \text{CAR} (L_4) = A \)
     - \( \text{CADR} (L_4) = B \)
     - \( \text{CADDR} (L_4) = C \)
     - \( \text{CDDDR} (L_4) = L_4 \)
     - \( \text{CADDDR} (L_4) = A \)

2. thus the s-expression is not tree-like
3. we will in general not be dealing with such structures
EXTENDED LIST NOTATION

- Next to last element has its `cdr` point to last element

- Sometimes used when desperate to save space
- Complicates many recursive algorithms by requiring a special check for the last element
- Empty list difficult to represent in a consistent manner with lists we have: `null(x)` with extended lists: `atom(cdr(x))`
- Note that `nil` is the empty list so adding element to it is just like adding element to a normal list
CONDITIONAL EXPRESSIONS

- Statements of the form:
  \[ \text{if } P \text{ then } a \text{ else } b \quad (P \text{ is known as a } \textit{predicate}) \]
- In LISP such a test is equivalent to writing:
  \[ \text{if not(NULL}(P)) \text{ then } a \text{ else } b \]
- Note we are \textit{not} testing for true, just not false (i.e., not NIL)
- More generally:
  \[ (\text{COND} \ (P_1 \ e_{11}) \ (P_2 \ e_{21}) \ (P_3 \ e_{31} \ e_{32} \ e_{33}) \ (P_4 \ e_4) \ (T \ e_5)) \]
  1. basically find first non-NIL \( P_i \) and evaluate \( e_{i1}, e_{i2}, \ldots e_{in} \)
  2. return the value of the last of the \( e_i \)'s – i.e., \( e_{in} \)
  3. \( T \) denotes the final else
  4. any of the \( P_i \) or \( e_{ij} \) could themselves be \text{COND} forms
- When writing conditional expression in LISP we have:
  \[ (\text{COND} \ (P \ a) \ (T \ b)) \quad \{ \text{if } P \text{ then } a \text{ else } b \} \]
  \[ (\text{COND} \ (P \ a) \ (Q \ b \ c) \ \ldots \ (S \ d) \ (T \ e)) \quad \{ \text{if } P \text{ then } a \text{ else if } Q \text{ then } b \text{ also } c \text{ else } \ldots \text{ else if } S \text{ then } d \text{ else } e \} \]
- Ex: \(-\infty < x < -1 \Rightarrow \text{tri}(x) = 0\)
  \[-1 \leq x < 0 \Rightarrow \text{tri}(x) = 1+x\]
  \[0 \leq x < 1 \Rightarrow \text{tri}(x) = 1-x\]
  \[1 \leq x < \infty \Rightarrow \text{tri}(x) = 0\]
  \[\text{TRI}[x] = \begin{cases} 
  0 & \text{if } x < -1 \\
  1+x & \text{if } x < 0 \\
  1-x & \text{if } x < 1 \\
  0 & \text{else} 
\end{cases} \]
SPECIAL FORMS

• Special forms imply special handling by EVAL

• SETQ is special form for binding values to variables does not evaluate its first argument

• SET is like SETQ except that all arguments are evaluated

\[
\text{(SETQ L1 (CAR A))} \equiv \text{(SET (QUOTE L1) (CAR A))}
\]

• Generally LISP evaluates in call-by-value fashion arguments evaluated left-to-right then function invoked e.g., \((\text{PLUS (TIMES 2 3) 4})\)
  1. multiply 2 and 3 to get 6
  2. invoke PLUS on 6 and 4 to yield 10

• COND is special form – args evaluated until TRUE found

\[
\begin{align*}
\text{EVAL}[l] &= \text{if } l[1] \text{ eq } \text{COND} \text{ then } \text{EVALCOND}(\text{CDR } l) \\
& \quad \vdots \\
\text{EVALCOND}[l] &= \text{if } \text{NULL}(l) \text{ then } \text{NIL} \\
& \quad \text{else if } \text{EVAL}(l[1,1]) \text{ then } \text{EVLIST}(\text{CDR } l[1]) \\
& \quad \text{else } \text{EVALCOND}[\text{CDR } l] \\
\text{EVLIST}[l] &= \text{if } \text{NULL}(\text{CDR}(l)) \text{ then } \text{EVAL}(l[1]) \\
& \quad \text{else } \text{EVAL}(l[1]) \text{ also } \text{EVLIST}[\text{CDR } l]
\end{align*}
\]
SHORT-CIRCUITING OF BOOLEAN CONNECTIVES

- LISP does *short-circuit* evaluation of Boolean expressions as soon as predicate’s value is determined, evaluation ends
  - Ex: \( A \text{ AND } B \): B is not evaluated if A is known to be \text{NIL}
  - \( A \text{ OR } B \): B is not evaluated if A is known to be \text{non-NIL}

- Can also represent \text{AND} and \text{OR} in terms of conditionals:
  \[ A_1 \text{ AND } A_2 \text{ AND } A_3 \ldots \text{ AND } A_n \]
  - if \( A_1 \) then
    - if \( A_2 \) then
      - if \( A_3 \) then
        - ... if \( A_{n-1} \) then \( A_n \)
        - else nil
      - else nil
    - else nil
  - else nil

- Better is:
  \[ \text{if not}(A_1) \text{ then nil else if not}(A_2) \text{ then nil else ... else if not}(A_{n-1}) \text{ then nil else } A_n \]

- Similarly for \text{OR}:
  \[ A_1 \text{ OR } A_2 \text{ OR } A_3 \ldots \text{ OR } A_n \]
  - if \( A_1 \) then \( T \)
  - else if \( A_2 \) then \( T \)
  - else ...
  - else if \( A_{n-1} \) then \( T \)
  - else \( A_n \)
RECURSION

• We have already seen *recursive* functions (MEMBER, etc.)

• Until now we have only constructed *predicates*, i.e.,
  functions that return only TRUE or FALSE (*T* or *NIL*)

• General LISP function maps from \((s\text{-expr})^n\) to \(s\text{-expr}\)

• We will now construct functions that return lists or
  general s-expressions
RECURSION EXAMPLE

• Given a list, return a list consisting of every other element in the input list starting with the first element

• Ex: ALT['(A B C D E)] ⇒ (A C E)
  ALT['(A B)] ⇒ (A)
  ALT['(A)] ⇒ (A)
  ALT['()] ⇒ ()

• ALT[x] = if nx or ndx then x
  else ax . alt[ddx]
EXAMPLES OF ALT

• To see that this really works:

\[
\text{ALT}[(A \ B)] = \begin{cases} 
\text{n'(A \ B) \lor nd'(A \ B) & \text{then } (A \ B) \\
\text{else } a'(A \ B) \cdot \text{ALT}[dd'(A \ B)]
\end{cases}
\]

\[
= \begin{cases} 
\text{if NIL \lor nd'(A \ B) then } (A \ B) \\
\text{else } a'(A \ B) \cdot \text{ALT}[dd'(A \ B)]
\end{cases}
\]

\[
= \begin{cases} 
\text{if NIL then } (A \ B) \\
\text{else } a'(A \ B) \cdot \text{ALT}[dd'(A \ B)]
\end{cases}
\]

\[
= a'(A \ B) \cdot \text{ALT}[dd'(A \ B)]
\]

\[
= 'A \cdot \text{ALT}[NIL]
\]

\[
= 'A \cdot [ \begin{cases} 
\text{if nNIL \lor ndNIL then NIL} \\
\text{else } aNIL \cdot \text{ALT}[ddNIL]
\end{cases}
\]

\[
= 'A \cdot [ \begin{cases} 
\text{if T then NIL} \\
\text{else } aNIL \cdot \text{ALT}[ddNIL]
\end{cases}
\]

\[
= 'A \cdot [ \text{NIL}]
\]

\[
= '(A)
\]

• A briefer example:

\[
\text{ALT}['(A \ B \ C \ D \ E)] = 'A \cdot \text{ALT}['(C \ D \ E)]
\]

\[
= 'A \cdot [ 'C \cdot \text{ALT}['(E)]]
\]

\[
= 'A \cdot [ 'C \cdot (E)]
\]

\[
= '(A \ C \ E)
\]

• Observations:

1. rules for evaluating Boolean conditions are important since if evaluation of OR continued after finding one NIL we would then evaluate dNIL which is undefined

2. we can build a list result by returning from recursion
LAST ATOM OF A LIST

• Construct a function to return the last atom of a list
  base case: nothing after current element?
    if ndx then ax
  
  induction case: get last of rest of list

LAST[x] = if ndx then ax
    else last[dx]

• Is there a problem with this definition?
  1. what happens when called on an atom?
    • if CDR of atom is property list we may never terminate
    • if CDR of atom is NIL then we get CAR of atom which
      is also probably not what we want
    • this definition only works if x is a list
  2. what if the list is empty?
    • same problem, as empty is represented by NIL atom
    • could explicitly check for empty list and return NIL

• Exercise: modify LAST to return a value of NIL if the last
  element of the list is an atom
SUBSTITUTE FUNCTION

• Substitute s-expression \( x \) for all occurrences of atom \( y \) in the s-expression \( z \)
  for example: \( \text{SUBST}[(A.B), 'Y', ((Y.A).Y)] \)
  yields: \(((A.B).A).(A.B))\)

• One approach is to check each item in \( z \) for equality with the atom \( y \) and if so replace by s-expression \( x \)

  base case: \( \text{if atz then} \)
              \( \text{if z eq y then x} \)
              \( \text{else z} \)

  inductive case: \( \text{subst}[x,y,az], \text{subst}[x,y,dz] \)

  However, we want the s-expression as our result
  • \text{CONS} the results of subst on the head and tail of \( z \)

\[
\text{SUBST}[x,y,z] = \begin{cases} 
\text{if atz then} & \\
  \text{if z eq y then x} & \\
  \text{else z} & \\
  \text{else subst}[x,y,az].\text{subst}[x,y,dz] & 
\end{cases}
\]
APPEND FUNCTION

- Takes two lists as arguments and concatenates them
  \[
  \begin{array}{c}
  x \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \cdots \quad \rightarrow \\
  y \rightarrow \quad \rightarrow \quad \cdots \quad \rightarrow \\
  \end{array}
  \]

- Could march down first list to last element then change link to point to second list

- Since argument lists may be shared with other data structures we instead make a copy of the first list

- Form a list consisting of all elements of \(x\) until reach end of \(x\) at which time attach \(y\)

  base case: \(\text{if } nx \text{ then } y\)

  induction: \([ax].\text{append}[dx, y]\)

\[
\text{APPEND}[x, y] = \begin{cases} 
  \text{if } nx \text{ then } y \\
  \text{else } [ax].\text{APPEND}[dx, y] 
\end{cases}
\]
REVERSE FUNCTION

- Use auxiliary function to simplify task
  \[
  \text{REVERSE}[x] = \text{REVERSE1}[x, \text{nil}]
  \]

\[
\text{REVERSE1}[x, y] = \begin{cases} 
  \text{if } nx \text{ then } y \\
  \text{else } \text{REVERSE1}[dx, <ax>.y]
\end{cases}
\]

- Variable \( y \) serves as place-holder to contain result
- Also possible to define using only one function and APPEND
  \[
  \text{REVERSE}[x] = \begin{cases} 
  \text{if } nx \text{ then } \text{nil} \\
  \text{else } \text{REVERSE}[dx]*<ax>
\end{cases}
  \]

- Using an auxiliary function is more efficient than using APPEND
  1. no need to postpone operations until return from recursion
  2. avoids repeated calls to APPEND to make new lists
FLATTEN FUNCTION

• Make flat list of all atoms in a given s-expression

• Use auxiliary function FLAT[x,y] where y accumulates the atoms

• Result list will contain atoms encountered from left to right

• Whenever an atom is encountered we add it to y

  base case: if atx then x.y

  induction: first flatten tail then head (to preserve order)
              flat[ax, flat[dx, y]]

  FLATTEN[x]=FLAT[x,nil]

  FLAT[x,y]=if atx then x.y
             else FLAT[ax,FLAT[dx,y]]

• This technique is useful for applying an arbitrary function to both the head and tail of a given s-expression

• Could also be constructed without using auxiliary function:
  FLATTEN[x]= if atx then x
               else FLATTEN[ax]*FLATTEN[dx]
TRADEOFF: EXTRA ARG VS EFFICIENCY

- Common when creating LISP functions
  
  **Ex: factorial:**
  
  \[
  \text{FACT}[x] = \begin{cases} 
  1 & \text{if } x \text{ eq } 1 \\
  x \cdot \text{FACT}[x-1] & \text{else}
  \end{cases}
  \]

  with the addition of a second argument:
  
  \[
  \text{FACT}[x] = \text{FACT1}[x,1]
  \]

  \[
  \text{FACT1}[x,y] = \begin{cases} 
  y & \text{if } x \text{ eq } 1 \\
  \text{FACT1}[x-1,x \cdot y] & \text{else}
  \end{cases}
  \]

- The general case
  
  1. note similarity of transformation in `REVERSE` and `FACT`
  2. consider these schemas
     
     \[
     f(x) = \begin{cases} 
     a & \text{if } p(x) \\
     b \oplus f(g(x)) & \text{else}
     \end{cases} \quad \Rightarrow \quad h(x,y) = \begin{cases} 
     a & \text{if } p(x) \\
     b \oplus h(g(x),y) & \text{else}
     \end{cases}
     \]
     
  3. when are these transformations valid?
     
     - \( \oplus \) must be associative

  4. **Ex: `REVERSE`:**
     
     \[
     p \text{ is null} \\
     a \text{ is } \text{NIL} \\
     b \text{ is } \text{<CAR x>} \\
     g \text{ is } \text{CDR} \\
     \oplus \text{ is } \text{APPEND}
     \]
     
     \[
     b \oplus y \equiv \text{<CAR x>} \text{ APPEND } y \equiv a \times \text{CONS } y \text{ (since } a \times \text{ is atom)}
     \]

  5. **Ex: `FACTORIAL`:**
     
     \[
     p \text{ is } x \text{ eq } 1 \\
     a \text{ is } 1 \\
     b \text{ is } x \\
     g \text{ is } x-1 \\
     \oplus \text{ is multiplication}
     \]
GREATER COMMON DENOMINATOR

- highest number that divides both \( m \) and \( n \)

- recursively:

\[
\text{GCD}(m,n) = \begin{cases} 
\text{GCD}(n,m) & \text{if } m > n \\
\text{GCD}(n \mod m, m) & \text{else if } m = 0 \\
\text{GCD}(n \mod m, m) & \text{else}
\end{cases}
\]

where:

\[
n \mod m = \begin{cases} 
n & \text{if } n < m \\
(n - m) \mod m & \text{else}
\end{cases}
\]

i.e., subtract until number between 0 and \( \min(m,n) - 1 \)
ASSOCIATION LISTS

- Common data structure in recursive programming
- Representation of dictionary as a list of s-expressions
  1. first element of each s-expression is a single atom
  2. rest of s-expression is atom’s definition or associated value
  3. Ex: \( x \) is associated to \(' (PLUS A B)\)
     \( y \) is associated to \(' c \)
     \( z \) is associated to \(' (TIMES U V)\)
     \(((x \, PLUS \, A \, B) \, (y \, . \, c) \, (z \, TIMES \, U \, V))\)
- Lookup using \( \text{ASSOC}(x,d) \)
  1. \( x \) is the atom to be looked up and \( d \) is the dictionary list
  2. if \( x \) is in the dictionary then the entire entry is returned
  3. if \( x \) is not in the dictionary then \( \text{NIL} \) is returned

  final case: \( \text{if } n1 \text{ then } n1 \)

  base case: \( \text{if } x \text{ eq } aal \text{ then } al \)

  induction step: \( \text{ASSOC}[x,dl] \)

\[ \text{ASSOC}[x,d] = \begin{cases} 
  \text{if } n1 \text{ then } n1 \\
  \text{else if } x \text{ eq } aal \text{ then } al \\
  \text{else } \text{ASSOC}[x,dl] 
\end{cases} \]

- Disadvantage is sequential search through entire list
  (since list is not kept in sorted order)
- We could represent the dictionary as a tree, but then
  lookup would be more complex, and insertion and deletion
  would be \textit{significantly} more complex
INTERNAL LAMBDA

- Avoid computing a function twice
- Compute once and store for future reference
- Ex: \(((\text{LAMBDA}(x\ y)\ \text{(PLUS} (\text{TIMES} 2 x)\ y))\) 3 4)  
  1. like a function without a name  
  2. binds 3 to x and 4 to y  
  3. computes \(2 \cdot x + y\)
- Ex: using ASSOC to substitute a dictionary value  
  1. if use of \(\text{ASSOC}(x,l)\) yields a non-\(\text{NIL}\) result and we want the actual definition of \(x\)  
  2. recall \(\text{ASSOC}\) returns \(\text{NIL}\) or entire entry including the atom \(x\) that we looked up  
  3. \(\lambda\)\(\text{(pair)}; \text{if } n\ \text{pair then } \text{NIL} \)
    \(\text{else } d\ \text{pair};\) \(\text{(ASSOC}(x,d))\) \(\}\) this is a segment
- Redefine \text{SUBST} so CONS only happens if there indeed was a substitution

\[
\text{SUBST2}[x,y,z]= \\
\text{if atz then} \\
\quad \text{if } z\ \text{eq } y\ \text{then } x \\
\quad \text{else } z \\
\text{else } \text{LAMBDA} (\text{head},\text{tail}); \\
\quad \text{if } \text{head}\ \text{eq } az\ \text{and } \text{tail}\ \text{eq } dz\ \text{then } z \\
\quad \text{else } \text{head}.\text{tail}; \\
\quad (\text{SUBST2}[x,y,az],\text{SUBST2}[x,y,dz])
\]
EXAMPLE OF THE USE OF INTERNAL LAMBDA

sexpr procedure substz(x,y,z); // Subst x for y in z
begin // Copy made only if a
  if atom(z) then // subst instance found
    if eq(y,z) then return(x)
    else return(z)
  else
    begin
      head \leftarrow substz(x,y,car(z));
      tail \leftarrow substz(x,y,cdr(z));
      if equal(head,car(z)) and
        equal(tail,cdr(z)) then return(z)
      else return(cons(head,tail));
    end;
end;

(CSETQ SUBSTZ (LAMBDA (X Y Z)
  (COND [(ATOM Z)
    (COND ((EQ Y Z) X)
      (T Z))]
  [T (< LAMBDA(HEAD TAIL)
    (COND [(AND (EQUAL HEAD (CAR Z))
      (EQUAL TAIL (CDR Z))) Z]
    [T (CONS HEAD TAIL)])>
    (SUBSTZ X Y (CAR Z))
    (SUBSTZ X Y (CDR Z)))])))
PROPERTY LISTS

- **Wisconsin LISP** represents an atom:

  ![Property List Diagram]

  - **Value cell** contains value bound to the atom i.e. `(SETQ A (QUOTE (JOHN MARY)))
  means that the value of A is (JOHN MARY)

  - **Print name** is atom’s name as a sequence of characters

  - **Property list**
    1. data structure storing two levels of information on atom
    2. like association list with addition of flag atoms
    3. Ex:

      ![Property List Example]

      is represented as:

      ![Property List Representation Diagram]
PROPERTY LIST FUNCTIONS

- Programmer need not (and should not) be aware of exact representation of the property list

- Functions to access property list

1. \(\text{(PUT } x \ y \ z)\) put property \(y\) on atom \(x\)’s property list with property value \(z\), e.g.,
   \(\text{(PUT (QUOTE AL) (QUOTE HAIR) (QUOTE RED))}\)

2. \(\text{(GET } x \ y)\) fetch property value associated with property \(y\) on atom \(x\)’s property list, e.g.,
   \(\text{(GET (QUOTE CHARLES) (QUOTE ADDRESS))}\)
   • just like ASSOC

3. \(\text{(REMPROP } x \ y)\) removes property \(y\) and its associated property value from atom \(x\)’s list, e.g.,
   \(\text{(REMPROP (QUOTE ANGOLA) (QUOTE COLONY))}\)

4. \(\text{(FLAG } x \ y)\) places flag \(y\) on atom \(x\)’s prop list, e.g.,
   \(\text{(FLAG (QUOTE MARY) (QUOTE MARRIED))}\)

5. \(\text{(IFFLAG } x \ y)\) returns TRUE if atom \(x\) has flag \(y\), e.g.,
   \(\text{(IFFLAG (QUOTE JOE) (QUOTE CITIZEN))}\)

6. \(\text{(UNFLAG } x \ y)\) removes flag \(y\) from atom \(x\)’s list, e.g.,
   \(\text{(UNFLAG (QUOTE CASE) (QUOTE RECESSED))}\)