INTRODUCTION

• The primary data structure is a list, e.g.,
  (A B C D E)
  (A)
  () empty list or NIL = a special name

• Can represent any entities
  \( x + y \)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \( xy + x + 3 \)

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

• 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
INTRODUCTION

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

• Can represent any entities
  \(x + y\) \hspace{1cm} \texttt{(PLUS \ x \ y)}
  \(\text{first item is operator, remaining items are operands}\)
  \(\text{can have an arbitrary number of arguments}\)
  \(xy + x + 3\)

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

• 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
INTRODUCTION

• The primary data structure is a list, e.g.,
  (A B C D E)
  (A)
  ()  empty list or NIL = a special name

• Can represent any entities
  \(x + y\)  \((PLUS x y)\)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \(xy + x + 3\)  \((PLUS (TIMES x y) x 3)\)

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

• 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
INTRODUCTION

• The primary data structure is a list, e.g.,
  \[(A \ B \ C \ D \ E)\]
  \[(A)\]
  \[]\empty list or \text{NIL} = \text{a special name}\]
• Can represent any entities
  \[x + y\quad (PLUS \ x \ y)\]
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \[xy + x + 3 \quad (PLUS \ (TIMES \ x \ y) \ x \ 3)\]
• We can refer to elements of a list by using brackets:
  for \[L = (PLUS \ (TIMES \ x \ y) \ x \ 3)\] we have
  \[L[1] = PLUS\]
  \[L[2] = \quad\]
  \[L[2,2] = \quad\]
  \[L[4] = \quad\]
  \[(\exists x) \ (\forall y) \ P(x) \supset P(y)\]

• An undirected graph

  \[\begin{array}{c}
    C \\
    A \quad B \\
    D \\
    E \quad F
  \end{array}\]

• 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
INTRODUCTION

• The primary data structure is a list, e.g.,
  \((A B C D E)\)
  \((A)\)
  \((())\) empty list or NIL = a special name

• Can represent any entities
  \(x + y\) \((PLUS x y)\)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \(xy + x + 3\) \((PLUS (TIMES x y) x 3)\)

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

• 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
INTRODUCTION

- The primary data structure is a list, e.g.,
  (A B C D E)
  (A)
  () empty list or NIL = a special name
- Can represent any entities
  \[ x + y \] \( (PLUS \ x \ y) \)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \[ xy + x + 3 \] \( (PLUS \ (TIMES \ x \ y) \ x \ 3) \)
- We can refer to elements of a list by using brackets:
  for \( L = (PLUS \ (TIMES \ x \ y) \ x \ 3) \) we have
  \( L[1] = PLUS \)
  \( L[2] = (TIMES \ x \ y) \)
  \( L[2, 2] = x \)
  \( L[4] = \)
  \( (\exists x) \ (\forall y) \ P(x) \cup P(y) \)

- 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
INTRODUCTION

• The primary data structure is a list, e.g.,
  (A B C D E)
  (A)
  () empty list or NIL = a special name

• Can represent any entities
  \( x + y \) \( (PLUS \ x \ y) \)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  \( xy + x + 3 \) \( (PLUS \ (TIMES \ x \ y) \times 3) \)

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

• An undirected graph

  \( \begin{array}{c}
  A \\
  B \\
  C \\
  D \\
  E \\
  F \\
  \end{array} \)

• 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
INTRODUCTION

• The primary data structure is a list, e.g.,
  (A B C D E)
  (A)
  () empty list or NIL = a special name

• Can represent any entities
  x + y (PLUS x y)
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
  xy + x + 3 (PLUS (TIMES x y) x 3)

• We can refer to elements of a list by using brackets:
  for L = (PLUS (TIMES x y) x 3) we have
    L[1] = PLUS
    L[2] = (TIMES x y)
    L[2,2] = x
    L[4] = 3
  (∃x) (∀y) P(x) ⊃ P(y)
  (EXIST x (ALL y (IMPLIES (P x)(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
INTRODUCTION

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

• Can represent any entities
\[x + y\] (PLUS \(x\) \(y\))
  first item is operator, remaining items are operands
  can have an arbitrary number of arguments
\[xy + x + 3\] (PLUS (TIMES \(x\) \(y\)) \(x\) \(3\))

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

• 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
\[
( (A \ B) (B \ A \ C \ D) (C \ B \ D \ E) \\
  (D \ B \ C \ E) (E \ C \ D \ F') (F \ E) )
\]
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:

  (A B C) would be:

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

- Solution: \text{INFO} points to another list!
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:

  ![Diagram of list representation](image)

  where Ω = NIL or

  (A B C) would be:

  ![Diagram of (A B C) representation](image)

- What about \(xy+x+3\) or \((\text{PLUS} \ (\text{TIMES} \ x \ y) \ x \ 3)\) ?
- Solution: INFO points to another list!
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:
  
  ![Diagram of list representation](image)

  where $\Omega = \text{NIL}$ or

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

  ![Diagram of list representation](image)

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

• Solution: *INFO* points to another list!

![Diagram of list representation](image)
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:
  
  A
  (A.B)
  (A. (B.A))
  (3 . 3.4) note convention about decimal point and dot: . is not a decimal point. Space around dot may be omitted if no confusion results:
  (PLUS. (x. (y.NIL)))

- Represented in computer memory by:

```plaintext
segmex pr sex pr

(A.B) →
(A. (B.A)) →

(PLUS. (x. (y.NIL)))
```

- This should be familiar
S-EXPRESSIONS

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

<sexpr> ⇒ <atom> | ( <sexpr> . <sexpr> )

• Examples:
  A
  (A.B)
  (A. (B.A))
  (3 . 3.4) note convention about decimal point and dot: . . is not a decimal point. Space around dot may be omitted if no confusion results:
  (PLUS. (x. (y.NIL)))

• Represented in computer memory by:

  (PLUS. (x. (y.NIL))) →

• This should be familiar
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:
  
  \[
  \begin{align*}
  &A \\
  &(A.B) \\
  &(A.(B.A)) \\
  & (3 . 3.4) \text{ note convention about decimal point and dot: . . is not a decimal point. Space around dot may be omitted if no confusion results:} \\
  & (PLUS.(x.(y.NIL)))
  \end{align*}
  \]

- Represented in computer memory by:

\[
\begin{align*}
  &\text{sexpr sexpr} \\
  &(A.B) \rightarrow \begin{array}{c}
  \text{A} \\
  \text{B}
  \end{array} \\
  &(A.(B.A)) \rightarrow \begin{array}{c}
  \text{A} \\
  \text{B} \\
  \text{A}
  \end{array}
  \end{align*}
\]

\[
\begin{align*}
  & (PLUS.(x.(y.NIL))) \rightarrow \begin{array}{c}
  \text{PLUS} \\
  \text{x} \\
  \text{y}
  \end{array}
  \end{align*}
\]

- This should be familiar
S-EXPRESSIONS

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

<sexpr> ⇒ <atom> | ( <sexpr> . <sexpr> )

- Examples:

A

(A.B)

(A. (B.A))

(3 . 3.4) note convention about decimal point and dot: . . is not a decimal point. Space around dot may be omitted if no confusion results:

(PLUS. (x. (y.NIL))))

- Represented in computer memory by:

(A.B) → A B

(A. (B.A)) → A B

(PLUS. (x. (y.NIL))) → PLUS x y

- This should be familiar (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:
     \[
     (\text{CDR} \ \text{(QUOTE} \ \text{(A} \ \text{B} \ \text{C})\text{)})
     \]
     \[
     (\text{CDR} \ ' \ (A \ B \ C))
     \]
     \[
     \text{CDR} \ (\text{' (A} \ \text{B} \ \text{C})\text{)}
     \]
     \[
     \text{CDR} [ \ (\text{' (A} \ \text{B} \ \text{C})\text{)}
     \]
     \[
     \text{CDR} [ \ (\text{QUOTE} \ \text{(A} \ \text{B} \ \text{C})\text{)}\text{)}
     \]
     - use \[ \text{[]} \] 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:**
     \[
     \text{CONS} [ \ '\text{A} \ , \ '\text{(B} \ \text{C} \ \text{D})\text{]} \equiv \ (A \ B \ C \ D) \equiv \text{CONS} [ \ '\text{A} \ , \ '\text{(B} \ . \ \text{(C} . \ \text{(D} . \ \text{NIL})\text{)})\text{)] \equiv \ (A \ . \ (B \ . \ (C \ . \ (D} . \ \text{NIL})\text{)})
     \]
LISP EXAMPLES

\[(A.B)\quad (C.D)\]

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

\[\text{CAR}'((A.B).(C.D))\] =
\[\text{CAR}[\text{CAR}'((A.B).(C.D))\] =
\[\text{CAAR}'((A.B).(C.D))\] =

- note use of \text{CAAR} for \text{CAR}(\text{CAR}(x))
- also \text{CADR}(x) = \text{CAR}(\text{CDR}(x))
- \text{CDR} is performed first followed by \text{CAR}
- can construct any combination needed

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

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

\[\text{Important:}\quad \text{CAR}[\text{CONS}'A,'B]\] =
\[\text{CDR}[\text{CONS}'A,'B]\] =
\[\text{CONS}[\text{CAR}'(A.B),\text{CDR}'(A.B)]\] =
LISP EXAMPLES

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

- note use of CAAR for \text{CAR}( \text{CAR}(x))
- also \text{CADR}(x) = \text{CAR}( \text{CDR}(x))
- \text{CDR} is performed first followed by \text{CAR}
- can construct any combination needed

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

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

Copyright © 2003 by Hanan Samet
LISP EXAMPLES

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

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

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

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

- 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:  
\[
\text{CAR[CONS['A,'B]] =}
\]

\[
\text{CDR[CONS['A,'B]] =}
\]

\[
\text{CONS[CAR['(A.B)],CDR['(A.B)]] =}
\]
LISP EXAMPLES

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

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

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

\[
\text{CAAR}[\text{CONS}[\text{'A, '(B C D)]] = \text{CAAR} \text{ CAR(x) } \text{ Note use of CAAR for } \text{ CAR( CAR(x) )}
\]

\[
\text{Also CADR(x) = CAR( CDR(x) )}
\]

\[
\text{CDR is performed first followed by CAR}
\]

\[
\text{Can construct any combination needed}
\]

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

LISP EXAMPLES

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

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

\[\text{CDR}[(\text{CONS}['A, 'B])] =
\]
\[\text{CONS}[(\text{CAR}[(A.B)], \text{CDR}[(A.B)])] =
\]

Copyright © 2003 by Hanan Samet
LISP EXAMPLES

\[(A.B) \quad (C.D)\]

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

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

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

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

- note use of CAAR for \text{CAR}( \text{CAR}(x) )
- also \text{CADR}(x) = \text{CAR}( \text{CDR}(x) )
- \text{CDR} is performed first followed by \text{CAR}
- can construct any combination needed

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

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

\[\text{Important: } \text{CAR}[\text{CONS}'A,'B'] = \]

\[\text{CDR}[\text{CONS}'A,'B'] = \]

\[\text{CONS}[\text{CAR}'(A.B)',\text{CDR}'(A.B)'] = \]
LISP EXAMPLES

\[
\begin{align*}
(A.B) & \quad (C.D) & \quad \text{CONS}['(A.B),'(C.D)] = \\
A & \quad B & \quad C & \quad D
\end{align*}
\]

\[
\begin{align*}
\text{CAR}['(A.B).(C.D)) & = (A.B) \\
\text{CAR}[\text{CAR }'(A.B).(C.D)) & = A \\
\text{CAAR}['(A.B).(C.D)) & = A
\end{align*}
\]

- 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

\[
\begin{align*}
\text{CONS}['(A.B),'A] & = ((A.B).A) \\
\text{CONS}[A, '(B C D)] & = (A B C D)
\end{align*}
\]

\[
\begin{align*}
\text{Important: } \text{CAR}[\text{CONS }'[A,'B]] & = A \\
\text{CDR}[\text{CONS }'[A,'B]] & = \\
\text{CONS}[\text{CAR }'[A.B],\text{CDR }'[A.B]] & =
\end{align*}
\]

Copyright © 2003 by Hanan Samet
LISP EXAMPLES

\[(A\cdot B)\] \hspace{1cm} \[(C\cdot D)\]

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

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

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

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

• note use of CAAR for \(\text{CAR(CAR(x))}\)
• also \(\text{CADR(x)} = \text{CAR(CDR(x))}\)
• \(\text{CDR}\) is performed first followed by \(\text{CAR}\)
• can construct any combination needed

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

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

\[\text{Important: CAR[CONS['A,'B]]} = A\]

\[\text{CDR[CONS['A,'B]]} = B\]

\[\text{CONS[CAR['(A\cdot B)],CDR['(A\cdot B)]]} =\]
LISP EXAMPLES

\[(A.B)\] \[\downarrow\] \[\downarrow\] \[\downarrow\] \[\downarrow\] \[\downarrow\] \[\downarrow\] \[\downarrow\] \[\downarrow\]
\[A\] \[B\] \[C\] \[D\]

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

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

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

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

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

• Lists may be shared:

\[ x \leftarrow \text{CONS 'A 'B} \]

is the same as \((A.B).((A.B).(D.NIL)))\)

which can also be represented as:

\[ y \leftarrow \text{CONS 'A 'B} \]

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

\[ x \leftarrow \]

i.e.,

\[ y \leftarrow \]
SHARING OF LISTS

• Lists may be shared:

\[
\begin{align*}
\text{x} & \quad \text{A} \quad \text{B} \\
\text{C} & \\
\text{D} & \\
\end{align*}
\]

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

which can also be represented as:

\[
\begin{align*}
\text{y} & \quad \text{A} \quad \text{B} \\
\text{C} & \\
\text{D} & \\
\end{align*}
\]

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

\[
\begin{align*}
\text{x} & \leftarrow (\text{CONS z (CONS 'C (CONS z (CONS 'D NIL)))))} \\
\text{y} & \leftarrow \\
\end{align*}
\]

i.e.,
SHARING OF LISTS

• Lists may be shared:

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:

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

i.e.,

\[ y \leftarrow (\text{CONS (CONS 'A 'B) (CONS 'C (CONS (CONS 'A 'B) (CONS 'D NIL))))} \]
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
     \[
     \text{EQ}[\text{CAR}(x), \text{CADDR}(x)] = \text{atom denoting value True}
     \]
     \[
     \text{EQ}[\text{CAR}(y), \text{CADDR}(y)] = \text{just like False}
     \]
  2. atoms are uniquely represented
     \[
     \text{EQ}[\text{CADR}(x), \text{CADR}(y)] = \text{while}
     \]
     \[
     \text{EQ}[\text{CAR}(x), \text{CAR}(y)] = \text{and}
     \]
     \[
     \text{EQ}[\text{CAAR}(x), \text{CAAR}(y)] = = \text{EQ}[\text{CDAR}(x), \text{CDAR}(y)]
     \]

• s-expressions \( x \) and \( y \) are \textit{structurally equivalent}
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case:

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

• Can we test to see if any sharing exists? (for example, if first and third elements of \( x \) identical?)
  1. the \( \text{EQ} \) predicate performs this test

\[
\text{EQ}\left[\text{CAR}\left( x \right), \text{CADDR}\left( x \right) \right] = \text{T} \quad \text{atom denoting value True}
\]

\[
\text{EQ}\left[\text{CAR}\left( y \right), \text{CADDR}\left( y \right) \right] = \text{just like False}
\]

2. atoms are uniquely represented

\[
\text{EQ}\left[\text{CADR}\left( x \right), \text{CADR}\left( y \right) \right] = \text{while}
\]

\[
\text{EQ}\left[\text{CAR}\left( x \right), \text{CAR}\left( y \right) \right] = \text{and}
\]

\[
\text{EQ}\left[\text{CAAR}\left( x \right), \text{CAAR}\left( y \right) \right] = \text{=} \text{EQ}\left[\text{CDAR}\left( x \right), \text{CDAR}\left( y \right) \right]
\]

• \( x \) and \( y \) are *structurally equivalent*
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom

   base case:

   3. need way to find out if something is an atom
   • use the \( \text{ATOM} \) function

   4. thus \( \text{EQUAL}\left[ x, y \right] = \)

• this should be familiar from our discussion of similarity and equivalence of binary trees
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
     
     \[
     EQ[\text{CAR}(x), \text{CADDR}(x)] = \text{atom denoting value True} \\
     EQ[\text{CAR}(y), \text{CADDR}(y)] = \text{NIL just like False}
     \]

  2. atoms are uniquely represented
     
     \[
     EQ[\text{CADR}(x), \text{CADR}(y)] = \text{while} \\
     EQ[\text{CAR}(x), \text{CAR}(y)] = \text{and} \\
     EQ[\text{CAAR}(x), \text{CAAR}(y)] = \text{=}EQ[\text{CDAR}(x), \text{CDAR}(y)]
     \]

• s-expressions \( x \) and \( y \) are structurally equivalent
  
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case:

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
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
     
     $EQ[CAR(x), CADDR(x)] = T$ atom denoting value True
     $EQ[CAR(y), CADDR(y)] = NIL$ just like False

  2. atoms are uniquely represented
     
     $EQ[CADR(x), CADR(y)] = T$ while
     $EQ[CAR(x), CAR(y)] = \text{and}$
     $EQ[CAAR(x), CAAR(y)] = =EQ[CDAR(x), CDAR(y)]$

• 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:

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

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

- s-expressions \( x \) and \( y \) are \textit{structurally equivalent}
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case:

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

- this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

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

• \( s \)-expressions \( x \) and \( y \) are structurally equivalent
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case:

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

• Can we test to see if any sharing exists? (for example, if first and third elements of \( x \) identical?)

  1. the \( \text{EQ} \) predicate performs this test
     \[
     \text{EQ}[\text{CAR}(x), \text{CADDR}(x)]= \text{T} \quad \text{atom denoting value True}
     \]
     \[
     \text{EQ}[\text{CAR}(y), \text{CADDR}(y)]= \text{NIL} \quad \text{just like False}
     \]

  2. atoms are uniquely represented
     \[
     \text{EQ}[\text{CADR}(x), \text{CADR}(y)]= \text{T} \quad \text{while}
     \]
     \[
     \text{EQ}[\text{CAR}(x), \text{CAR}(y)]= \text{NIL} \quad \text{and}
     \]
     \[
     \text{EQ}[\text{CAAR}(x), \text{CAAR}(y)]= \text{T} = \text{EQ}[\text{CDAR}(x), \text{CDAR}(y)]
     \]

• \( x \) and \( y \) are structurally equivalent

  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case: \( \text{atom}(x) \Rightarrow \text{atom}(y) \)
     otherwise \( \text{NIL} \)

     • if either \( x \) or \( y \) are atoms, then \( \text{EQ}(x, y) \)
     otherwise \( \text{EQUAL} \) first parts and \( \text{EQUAL} \) second parts

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

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

• s-expressions \( x \) and \( y \) are *structurally equivalent*
  1. can we write function \( \text{EQUAL} \) to test for this?
  2. smallest indivisible unit is the atom
     base case: \( \text{atom}(x) \Rightarrow \text{atom}(y) \)
     otherwise \( \text{NIL} \)

• if either \( x \) or \( y \) are atoms, then \( \text{EQ}(x, y) \)
  otherwise \( \text{EQUAL} \) first parts and \( \text{EQUAL} \) second parts

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

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

• this should be familiar from our discussion of similarity and equivalence of binary trees
STRUCTURAL EQUIVALENCE

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

• s-expressions $x$ and $y$ are structurally equivalent
  1. can we write function $\text{EQUAL}$ to test for this?
  2. smallest indivisible unit is the atom
     base case: $\text{atom}(x) \Rightarrow \text{atom}(y)$
     otherwise $\text{NIL}$

• if either $x$ or $y$ are atoms, then $\text{EQ}(x, y)$
  otherwise $\text{EQUAL}$ first parts and $\text{EQUAL}$ second parts

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

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

• 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) \text{ 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 \langle x, y, z \rangle
REPRESENTING TREES

• Linear: 

• More tree-like representation: 

• More balanced: 

• In unbalanced representation: 
  
  \[
  \text{CAR}(L_1) = \text{CADR}(L_1) = \text{CADDR}(L_1) = \text{CADDDR}(L_1) = \frac{1+2+3+4}{4} = 2.5 \text{ operations}
  \]

• In balanced representation: 
  
  \[
  \text{CAAR}(L_2) = \text{CDAR}(L_2) = \text{CADR}(L_2) = \text{CDDR}(L_2) = \frac{2+2+2+2}{4} = 2 \text{ operations}
  \]

• Advantage of \(L_1\): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

• Linear:

```
L → ⬇️ ⬇️ ⬇️ ⬇️
A B C D
```

• More tree-like representation: $L_1$
  note all information appears at terminal nodes only

```
L1 → ⬇️ ⬇️ ⬇️
A
```

• More balanced:
  only two ops to get to any particular node

```
L2 → ⬇️ ⬇️
A B

C D
```

• In unbalanced representation:

  $\text{CAR}(L_1) = A$
  $\text{CADR}(L_1) = $
  $\text{CADDR}(L_1) = $
  $\text{CADDDR}(L_1) = $

  average of $\frac{1+2+3+4}{4} = 2.5$ operations

• In balanced representation:

  $\text{CAAR}(L_2) = $
  $\text{CDAR}(L_2) = $
  $\text{CADR}(L_2) = $
  $\text{CDDR}(L_2) = $

  average of 2 operations

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

• Linear:  

- More tree-like representation: \( L_1 \)
  - Note all information appears at terminal nodes only

- More balanced:
  - Only two ops to get to any particular node

• In unbalanced representation:

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

• In balanced representation:

  \[
  \begin{align*}
  \text{CAAR}(L_2) &= \\
  \text{CDAR}(L_2) &= \\
  \text{CADR}(L_2) &= \\
  \text{CDDR}(L_2) &= \\
  \text{Average of 2 operations}
  \end{align*}
  \]

• Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

• Linear:

![Linear representation diagram]

- More tree-like representation: \( L_1 \)
  - Note all information appears at terminal nodes only

- More balanced:
  - Only two ops to get to any particular node

- In unbalanced representation:
  - \( \text{CAR}(L_1) = A \)
  - \( \text{CADR}(L_1) = B \)
  - \( \text{CADDR}(L_1) = C \)
  - \( \text{CADDDR}(L_1) = \)
  - Average of \( \frac{1+2+3+4}{4} = 2.5 \) operations

- In balanced representation:
  - \( \text{CAAR}(L_2) = \)
  - \( \text{CDAR}(L_2) = \)
  - \( \text{CADR}(L_2) = \)
  - \( \text{CDDR}(L_2) = \)
  - Average of 2 operations

- Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

• Linear: 

![Linear representation diagram]

- More tree-like representation: 
  
  ![More tree-like representation diagram]
  
  Note all information appears at terminal nodes only.

- More balanced: 
  
  ![More balanced representation diagram]
  
  Only two ops to get to any particular node.

- In unbalanced representation:
  
  \[
  \begin{align*}
  \text{CAR}(L_1) &= A \\
  \text{CADR}(L_1) &= B \\
  \text{CADDR}(L_1) &= C \\
  \text{CADDDR}(L_1) &= D \\
  \end{align*}
  \]

  Average of \( \frac{1+2+3+4}{4} \) = 2.5 operations.

- In balanced representation:
  
  \[
  \begin{align*}
  \text{CAAR}(L_2) &= \\
  \text{CDAR}(L_2) &= \\
  \text{CADR}(L_2) &= \\
  \text{CDDR}(L_2) &= \\
  \end{align*}
  \]

  Average of 2 operations.

- Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop.
REPRESENTING TREES

• Linear:  

```
  L → A → B → C → D
```

• More tree-like representation:  

```
  L_1 → A → B → C → D
```

  note all information appears at terminal nodes only

• More balanced:  

```
  L_2 → A → B → C → D
```

  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:

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

• Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

- Linear: 

```
L  
A  B  C  D
```

- More tree-like representation: 

```
L_1  
A  B  C  D
```

  note all information appears at terminal nodes only

- More balanced: 

```
L_2  
A  B  C  D
```

  only two ops to get to any particular node

- In unbalanced representation:

  \[ \begin{align*}
  \text{CAR}(L_1) &= A \\
  \text{CADR}(L_1) &= B \\
  \text{CADDR}(L_1) &= C \\
  \text{CADDR}(L_1) &= D \\
  \end{align*} \]

  average of \( \frac{1+2+3+4}{4} \) = 2.5 operations

- In balanced representation:

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

  average of 2 operations

- Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

• Linear:

• More tree-like representation: note all information appears at terminal nodes only

• More balanced: 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:

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

• Advantage of \( L_1 \): if searching for a particular element and list is not a fixed size, then we know when to stop
REPRESENTING TREES

• Linear: 

• More tree-like representation: 

• More balanced: 

• In unbalanced representation:
  \[\text{CAR}(L_1) = A\]
  \[\text{CADR}(L_1) = B\]
  \[\text{CADDR}(L_1) = C\]
  \[\text{CADDDR}(L_1) = D\]
  average of \(\frac{1+2+3+4}{4}\) = 2.5 operations

• In balanced representation:
  \[\text{CAAR}(L_2) = A\]
  \[\text{CDAR}(L_2) = B\]
  \[\text{CADR}(L_2) = C\]
  \[\text{CDDR}(L_2) = D\]
  average of 2 operations

• 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?
  2. induction:

  3. \( \text{member}[x, l] = \)

• How to write function in LISP?
• Need to assign a function body to the function name
  \[
  \text{(DEF fname (LAMBDA (arg1 arg2...argn) fbody))}
  \]
• For example:
  \[
  \text{member}[x, l] =
  \]
MEMBERSHIP IN LIST

• How would we search for x in list L₁?
  1. base case: how do we know when we are done?
    check for null list: if nl then nil
  2. induction:

3. member[x,l] =

• How to write function in LISP?
• Need to assign a function body to the function name
  (DEF fname (LAMBDA (arg1 arg2...argn) fbody))

• For example:
  member[x,l] =
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 \( n_1 \) then \( \text{nil} \) } \)
     check first element: \( \text{if \( a_1 \) eq \( x \) then \( T \) } \)
  2. induction:

• How to write function in LISP?
• Need to assign a function body to the function name
  (DEF fname (LAMBDA (arg1 arg2...argn) fbody))
• For example:
  member\([x, l]\) =
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: \( \text{if } nl \text{ then nil} \)
     check first element: \( \text{if } al \text{ eq } x \text{ then T} \)

  2. induction:

     check rest of list: \( dl \)

• member\([x, dl]\)

  3. member\([x, l]\) =

• How to write function in LISP?

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

• For example:
  \( \text{member}[x, l] = \)
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: if $n_l$ then nil
      check first element: if $a_l$ eq $x$ then T
  2. induction:
      check rest of list: $d_l$

• $\text{member}[x, d_l]$

3. $\text{member}[x, l] = \begin{cases} 
\text{if } n_l \text{ then nil} \\
\text{else if } a_l \text{ eq } x \text{ then T} \\
\text{else member}[x, d_l] 
\end{cases}$

• How to write function in LISP?
• Need to assign a function body to the function name
  (DEF fname (LAMBDA (arg1 arg2...argn) fbody))

• For example:
  $\text{member}[x, l] = $
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: $\text{if } n_1 \text{ then } \text{nil}$
     check first element: $\text{if } a_1 \text{ eq } x \text{ then } T$
  2. induction:
     check rest of list: $d_1$
     $\text{member}[x,d_1]$

3. $\text{member}[x,l] = \begin{cases} 
\text{if } n_l \text{ then } \text{nil} \\
\text{else if } a_l \text{ eq } x \text{ then } T \\
\text{else } \text{member}[x,d_l] 
\end{cases}$

• 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

\[
\text{membersexpr}[x,s] =
\]

- 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)?

\[
\text{members}[x,s] =
\]

- 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

Copyright © 2003 by Hanan Samet
MEMBERSHIP IN S-EXPRESSION

• How would we search for x in s-expression s?
• Analogous to searching terminal nodes of a tree

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

• 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)?

\[
\text{members}[x, s] = 
\]

• 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

Copyright © 2003 by Hanan Samet
MEMBERSHIP IN S-EXPRESSION

• How would we search for \(x\) in s-expression \(s\)?
• Analogous to searching terminal nodes of a tree

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

• 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)?

\[
\text{atom } s \quad \Rightarrow x \text{ eq } s
\]

• 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
MEMBERSHIP IN S-EXPRESSION

• How would we search for \( x \) in s-expression \( s \)?
• Analogous to searching terminal nodes of a tree

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

• 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)?

\[
\begin{align*}
\text{atom } s & \Rightarrow x \text{ eq } s \\
\text{not atom } s & \Rightarrow x \text{ equal } s \text{ or } \\
& \quad \text{members}[x,al] \text{ or members}[x,dl]
\end{align*}
\]

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

• 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: \texttt{L3}?

2. Work backwards:
   \begin{align*}
   \text{CDR (L3)} &= \ \\
   \text{CDAR (L3)} &= \ \\
   \text{CDAAR (L3)} &= \ \\
   \text{CDAAAR (L3)} &= \\
   \end{align*}

• Circular structures
  1. a list \textit{could} point back to component of itself
     \begin{align*}
     \text{CAR (L4)} &= \ \\
     \text{CADR (L4)} &= \ \\
     \text{CADDR (L4)} &= \ \\
     \text{CDDDR (L4)} &= \ \\
     \text{CADDDR (L4)} &= \\
     \end{align*}

2. thus the s-expression is not tree-like
3. we will in general not be dealing with such structures
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:
     \begin{align*}
     \text{CDR} \left( L_3 \right) &= A \\
     \text{CDAR} \left( L_3 \right) &= \\
     \text{CDAAR} \left( L_3 \right) &= \\
     \text{CDAAAR} \left( L_3 \right) &=
     \end{align*}

• Circular structures
  1. a list \textit{could} point back to component of itself
     \begin{align*}
     \text{CAR} \left( L_4 \right) &= \\
     \text{CADR} \left( L_4 \right) &= \\
     \text{CADDR} \left( L_4 \right) &= \\
     \text{CDDDR} \left( L_4 \right) &= \\
     \text{CADDDR} \left( L_4 \right) &=
     \end{align*}
  2. thus the s-expression is not tree-like
  3. we will in general not be dealing with such structures
ALTERNATIVE LIST REPRESENTATIONS

• Suppose we organize list by CDR instead of by CAR?
  1. What is lisp representation of this list: 

  2. Work backwards:
     \[ \text{CDR (L3)} = A \]
     \[ \text{CDAR (L3)} = B \]
     \[ \text{CDAAR (L3)} = \]
     \[ \text{CDAAAR (L3)} = \]

• Circular structures
  1. A list can point back to a component of itself
     \[ \text{CAR (L4)} = \]
     \[ \text{CADR (L4)} = \]
     \[ \text{CADDR (L4)} = \]
     \[ \text{CDDDR (L4)} = \]
     \[ \text{CADDDR (L4)} = \]
  2. Thus the s-expression is not tree-like
  3. We will in general not be dealing with such structures
ALTERNATIVE LIST REPRESENTATIONS

• Suppose we organize list by CDR instead of by CAR?
  1. What is lisp representation of this list: \[ L_3 \]

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

• Circular structures
  1. a list could point back to component of itself
   \[ \text{CAR}(L_4) = \]
   \[ \text{CADR}(L_4) = \]
   \[ \text{CADDR}(L_4) = \]
   \[ \text{CDDDR}(L_4) = \]
   \[ \text{CADDDR}(L_4) = \]

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

• Suppose we organize list by CDR instead of by CAR?
  1. What is lisp representation of this list: $L_3$

2. Work backwards:
   - $\text{CDR}(L3) = A$
   - $\text{CDAR}(L3) = B$
   - $\text{CDAAR}(L3) = C$
   - $\text{CDAAAR}(L3) = D$

• Circular structures
  1. a list could point back to component of itself
     - $\text{CAR}(L4) =$
     - $\text{CADR}(L4) =$
     - $\text{CADDR}(L4) =$
     - $\text{CDDDR}(L4) =$
     - $\text{CADDDR}(L4) =$

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

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

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

• Circular structures
  1. a list \textit{could} point back to component of itself
     CAR\( (L_4) = A \)
     CADR\( (L_4) = \)
     CADDR\( (L_4) = \)
     CDDDR\( (L_4) = \)
     CADDDDR\( (L_4) = \)

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

- Suppose we organize list by CDR instead of by CAR?
  1. What is lisp representation of this list: \[ L_3 \]

2. Work backwards:
   - CDR (L3) = A
   - CDAR (L3) = B
   - CDAAR (L3) = C
   - CDAAAR (L3) = D

- Circular structures
  1. A list could point back to component of itself
     - CAR (L4) = A
     - CADR (L4) = B
     - CADDR (L4) =
     - CDDDR (L4) =
     - CADDDR (L4) =

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

• Suppose we organize list by `cdr` instead of by `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 *could* point back to component of itself
     - $\text{CAR}(L_4) = A$
     - $\text{CADR}(L_4) = B$
     - $\text{CADDR}(L_4) = C$
     - $\text{CDDDR}(L_4) =$
     - $\text{CADDDR}(L_4) =$

2. thus the s-expression is not tree-like
3. we will in general not be dealing with such structures
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:
\[
\begin{align*}
\text{CDR (L3)} &= A \\
\text{CDAR (L3)} &= B \\
\text{CDAAR (L3)} &= C \\
\text{CDAAAR (L3)} &= D \\
\end{align*}
\]

• Circular structures
  1. a list \textit{could} point back to component of itself
\[
\begin{align*}
\text{CAR (L4)} &= A \\
\text{CADR (L4)} &= B \\
\text{CADDR (L4)} &= C \\
\text{CDDDR (L4)} &= L_4 \\
\text{CADDDR (L4)} &= \\
\\end{align*}
\]
  2. thus the s-expression is not tree-like
  3. we will in general not be dealing with such structures
ALTERNATIVE LIST REPRESENTATIONS

• Suppose we organize list by CDR instead of by 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 *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 \textit{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: \textsc{null}(x)
  with extended lists: \textsc{atom}(\textit{cdr}(x))
- Note that \texttt{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:
  \[ \text{if } P \text{ then } a \text{ else } b \] (\(P\) is known as a \textit{predicate})

- In SLP such a test is equivalent to writing:
  \[ \text{if not}(\text{NULL}(P)) \text{ then } a \text{ else } b \]

- 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 SLP we have:
  \[
  \text{(COND } (P \ a) \\
  (T \ b)) \} \text{ if } P \text{ then } a \text{ else } b \\
  \text{(COND } (P \ a) \\
  (Q \ b \ c) \\
  \ldots \\
  (S \ d) \\
  (T \ e)) \} \text{ if } P \text{ then } a \text{ else if } Q \text{ then } b \text{ also } c \\
  \ldots \text{ else } \ldots \) \text{ else if } S \text{ then } d \text{ else } e \\

- Ex: \(-\infty < x < -1 \Rightarrow \text{tri}(x) = 0 \)
  \(-1 \leq x < 0 \Rightarrow \text{tri}(x) = 1 + x \)
  \(0 \leq x < 1 \Rightarrow \text{tri}(x) = 1 - x \)
  \(1 \leq x < \infty \Rightarrow \text{tri}(x) = 0 \)

\[
\text{TRI}[x] =
\]
CONDITIONAL EXPRESSIONS

- Statements of the form:
  \[ \text{if } P \text{ then } a \text{ else } b \] (\(P\) is known as a \textit{predicate})

- In LISP such a test is equivalent to writing:
  \[ \text{if not} \left( \text{NULL}(P) \right) \text{ then } a \text{ else } b \]

- Note we are \textit{not} testing for true, just not false (i.e., not NIL)

- More generally:
  \[
  (\text{COND} \ (P_1 \ e_{11}) \ \\
  \quad (P_2 \ e_{21}) \ \\
  \quad (P_3 \ e_{31} \ e_{32} \ e_{33}) \ \\
  \quad (P_4 \ e_4) \ \\
  \quad (T \ e_5))
  \]

  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:
  \[
  (\text{COND} \ (P \ a) \ \\
  \quad (T \ b)) \]
  \[
  \text{if } P \text{ then } a \ \\
  \text{else } b
  \]

  \[
  (\text{COND} \ (P \ a) \ \\
  \quad (Q \ b \ c) \ \\
  \quad \ldots \ \\
  \quad (S \ d) \ \\
  \quad (T \ e))
  \]
  \[
  \text{if } P \text{ then } a \ \\
  \text{else if } Q \text{ then } b \text{ also } c \ \\
  \text{else } \ldots \ \\
  \text{else if } S \text{ then } d \ \\
  \text{else } e
  \]

- Ex: \(-\infty < x < -1 \Rightarrow \text{tri}(x) = 0\)
  \(-1 \leq x < 0 \Rightarrow \text{tri}(x) = 1+x\)
  \(0 \leq x < 1 \Rightarrow \text{tri}(x) = 1-x\)
  \(1 \leq x < \infty \Rightarrow \text{tri}(x) = 0\)

  \[\text{TRI}[x] = \begin{cases} \text{if } x<-1 \text{ then } 0 \\ \text{else if } x<0 \text{ then } 1+x \\ \text{else if } x<1 \text{ then } 1-x \\ \text{else } 0 \end{cases}\]
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
  \[(SETQ \, L1 \, (\text{CAR} \, A)) \equiv (\text{SET} \, (\text{QUOTE} \, L1) \, (\text{CAR} \, A))\]

• Generally LISP evaluates in call-by-value fashion arguments evaluated left-to-right then function invoked e.g., \((\text{PLUS} \, (\text{TIMES} \, 2 \, 3) \, 4)\)
  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

  \[
  \begin{align*}
  \text{EVAL}[l] &= \text{if } l[1] \text{ eq 'COND then EVALCOND(CDR } l) \\
  \cdot \\
  \text{EVALCOND}[l] &= \text{if NULL}(l) \text{ then NIL} \\
  &\quad \text{else if EVAL}(l[1,1]) \text{ then EVLIST(CDR } l[1]) \\
  &\quad \text{else EVALCOND(CDR } l) \\
  \text{EVLIST}[l] &= \text{if NULL}(\text{CDR}(l)) \text{ then EVAL}(l[1]) \\
  &\quad \text{else EVAL(EVLIST)}(l[1]) \text{ also EVLIST(CDR } l) 
  \end{align*}
  \]
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 AND B: B is not evaluated if A is known to be NIL
  A OR B: B is not evaluated if A is known to be non-NIL

- Can also represent AND and OR in terms of conditionals:

  \[A_1 \land A_2 \land A_3 \ldots A_n\]

  if \(A_1\) then
  if \(A_2\) then
  if \(A_3\) then
  \ldots
  if \(A_{n-1}\) then \(A_n\)
  else nil
  else nil
  else nil
  else nil

- Better is:

- Similarly for OR:

  \[A_1 \lor A_2 \lor A_3 \ldots A_n\]
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 \texttt{NIL}
  - \( A \text{ OR } B \): \( B \) is not evaluated if \( A \) is known to be \texttt{non-NIL}

- Can also represent \texttt{AND} and \texttt{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
        - \ldots
      - 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\( (A_2) \) then nil
  - else \ldots
  - else if not\( (A_{n-1}) \) then nil
  - else \( A_n \)

- Similarly for \texttt{OR}:
  \[ A_1 \text{ OR } A_2 \text{ OR } A_3 \ldots \text{ OR } A_n \]
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 *NIL*
  $A \text{ OR } B$: $B$ is not evaluated if $A$ is known to be *non-NIL*.

- Can also represent **AND** and **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:
    if not($A_1$) then nil
    else if not($A_2$) then nil
    else ...
    else if not($A_{n-1}$) then nil
    else $A_n$

- Similarly for **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-expr)^n to s-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: $\text{ALT}[\text{'(A B C D E)')] \Rightarrow$
  
  $\text{ALT}[\text{'(A B)')] \Rightarrow$
  
  $\text{ALT}[\text{'(A)')] \Rightarrow$
  
  $\text{ALT}[\text{'( )')] \Rightarrow$

• $\text{ALT}[x] = $
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)'] ⇒
  ALT['(A)'] ⇒
  ALT['()'] ⇒

• ALT[x] =
RECURSION EXAMPLE

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

• Ex: \text{ALT}['(A \ B \ C \ D \ E)'] \Rightarrow (A \ C \ E)
  \text{ALT}['(A \ B)'] \Rightarrow (A)
  \text{ALT}['(A)'] \Rightarrow
  \text{ALT}['()'] \Rightarrow

• \text{ALT}[x] =
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] =
RECURSION EXAMPLE

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

• Ex: \( \text{ALT}'(A \ B \ C \ D \ E)' \Rightarrow (A \ C \ E) \)
  \( \text{ALT}'(A \ B)' \Rightarrow (A) \)
  \( \text{ALT}'(A)' \Rightarrow (A) \)
  \( \text{ALT}'()' \Rightarrow () \)

• \( \text{ALT}[x] = \)
RECURSION EXAMPLE

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

• Ex: $\text{ALT}['(A B C D E)'] \Rightarrow (A C E)$
  $\text{ALT}['(A B)'] \Rightarrow (A)$
  $\text{ALT}['(A)'] \Rightarrow (A)$
  $\text{ALT}['()'] \Rightarrow ()$

• $\text{ALT}[x] = \text{if } nx \text{ or } ndx \text{ then } x$
  $\text{else } ax \cdot \text{alt}[ddx]$

Copyright © 2003 by Hanan Samet
EXAMPLES OF ALT

- To see that this really works:

\[
\begin{align*}
\text{ALT['(A B)']} &= \text{if } n'(A B) \lor nd'(A B) \text{ then } ' (A B) \\
& \quad \text{ else } a'(A B) \cdot \text{ALT[dd'}(A B)] \\
& = \text{if } NIL \lor nd'(A B) \text{ then } ' (A B) \\
& \quad \text{ else } a'(A B) \cdot \text{ALT[dd'}(A B)] \\
& = \text{if } NIL \text{ then } ' (A B) \\
& \quad \text{ else } a'(A B) \cdot \text{ALT[dd'}(A B)] \\
& = a'(A B) \cdot \text{ALT[dd'}(A B)] \\
& = 'A \cdot \text{ALT[NIL]} \\
& = 'A \cdot [ \text{if } nNIL \lor ndNIL \text{ then } NIL \\
& \quad \text{ else } aNIL \cdot \text{ALT[ddNIL]}] \\
& = 'A \cdot [ \text{if } T \text{ then } NIL \\
& \quad \text{ else } aNIL \cdot \text{ALT[ddNIL]}] \\
& = 'A \cdot [NIL] \\
& = '(A)
\end{align*}
\]

- A briefer example:

\[
\begin{align*}
\text{ALT['(A B C D E)']} &= 'A . \text{ALT['(C D E)']} \\
& = 'A . ['C . \text{ALT['(E)']} \\
& = 'A . ['C . (E)] \\
& = '(A C E)
\end{align*}
\]

- 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:

induction case:

\[ \text{LAST}[x] = \]

• 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 \( \text{LAST} \) to return a value of NIL if the last
  element of the list is an atom
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:

  \[ \text{LAST}[x] = \]

• 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 \textsc{Last} to return a value of NIL if the last element of the list is an atom
LAST ATOM OF A LIST

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

\[
\text{LAST}[x] =
\]

• 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
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:

  inductive case:

  $\text{SUBST}[x, y, z] = \ldots$
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, z] = $
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: \text{(((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: if atz then
  \hspace{1cm} if z eq y then x
  \hspace{1cm} else z

  inductive case: subst[x,y,az], subst[x,y,dz]

However, we want the s-expression as our result
• CONS the results of subst on the head and tail of z

\text{SUBST[x,y,z] =}
**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:** \( \text{if } \text{atz} \text{ then } \)
  
  \( \text{if } z \text{ eq } y \text{ then } x \)
  
  \( \text{else } z \)

  **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 \( \text{subst} \) on the head and tail of \( z \)

\[\text{SUBST}[x,y,z] = \]

\[\text{if } \text{atz} \text{ then }\]

\[\text{if } z \text{ eq } y \text{ then } x \]

\[\text{else } z \]

\[\text{else subst}[x,y,az].\text{subst}[x,y,dz] \]
APPEND FUNCTION

- Takes two lists as arguments and concatenates them
  \[ x \rightarrow \cdots \rightarrow \cdots \rightarrow y \]

- 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:

  induction:

  \[ \text{APPEND}[x, y] = \]
APPEND FUNCTION

• Takes two lists as arguments and concatenates them

\[
\begin{align*}
  x & \rightarrow \ldots \\
  y & \rightarrow \ldots
\end{align*}
\]

• 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:

\[
\text{APPEND}[x, y] =
\]
APPEND FUNCTION

- Takes two lists as arguments and concatenates them
  \[ x \rightarrow \ldots \rightarrow \]  
  \[ y \rightarrow \ldots \rightarrow \]  

- 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: \([ax].append[dx, y]\) 

\[ \text{APPEND}[x, y] = \]
APPEND FUNCTION

• Takes two lists as arguments and concatenates them

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

Copyright © 2003 by Hanan Samet
REVERSE FUNCTION

• Use auxiliary function to simplify task
  \[
  \text{REVERSE}[x] =
  \]

\[
\text{REVERSE}_1[x, y] =
\]

• Variable \( y \) serves as place-holder to contain result
• Also possible to define using only one function and \text{APPEND}
  \[
  \text{REVERSE}[x] =
  \]

• Using an auxiliary function is more efficient than using \text{APPEND}
  1. no need to postpone operations until return from recursion
  2. avoids repeated calls to \text{APPEND} to make new lists
REVERSE FUNCTION

• Use auxiliary function to simplify task

\[ \text{REVERSE}[x] = \text{REVERSE1}[x, \text{nil}] \]

\[ \text{REVERSE1}[x, y] = \]

• Variable \( y \) serves as place-holder to contain result
• Also possible to define using only one function and APPEND

\[ \text{REVERSE}[x] = \]

• 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
REVERSE FUNCTION

• Use auxiliary function to simplify task
  \[ \text{REVERSE}[x] = \text{REVERSE}1[x, \text{nil}] \]

\[ \text{REVERSE}1[x, y] = \begin{cases} \text{if } x \text{ then } y \\ \text{else } \text{REVERSE}1[dx, <ax>.y] \end{cases} \]

• Variable \( y \) serves as place-holder to contain result
• Also possible to define using only one function and \texttt{APPEND}
  \[ \text{REVERSE}[x] = \]

• Using an auxiliary function is more efficient than using \texttt{APPEND}
  1. no need to postpone operations until return from recursion
  2. avoids repeated calls to \texttt{APPEND} to make new lists
REVERSE FUNCTION

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

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

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

• 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:

  induction:

\[
\text{FLATTEN}[x] = \\
\text{FLAT}[x, y] =
\]

- 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:

\[
\text{FLATTEN}[x] =
\]
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: \(\text{if atx then x.y}\)

induction:

\[
\text{FLATTEN} [x] = \\
\text{FLAT} [x, y] =
\]

• 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:

\[
\text{FLATTEN} [x] =
\]
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 at $x$ then $x.y$

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

FLATTEN[$x$] =

FLAT[$x,y$] =

• 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[$x$] =
FLATTEN FUNCTION

• Make flat list of all atoms in a given s-expression
• Use auxiliary function $\text{FLAT}[x,y]$ 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: \( \text{if atx then x.y} \)

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

\[
\text{FLATTEN}[x] = \text{FLAT}[x, \text{nil}]
\]
\[
\text{FLAT}[x,y] = \text{if atx then x.y else FLAT}[ax, \text{FLAT}[dx,y]]
\]

• 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:
  \[
  \text{FLATTEN}[x] =
  \]
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: \texttt{if atx then x.y}

  induction: first flatten tail then head (to preserve order)

  \[
  \text{flat}[ax, \text{flat}[dx, y]]
  \]

  \[
  \text{FLATTEN}[x] = \text{FLAT}[x, \text{nil}]
  \]

  \[
  \text{FLAT}[x, y] = \texttt{if atx then x.y} \quad \texttt{else FLAT}[ax, \text{FLAT}[dx, y]]
  \]

• 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:

  \[
  \text{FLATTEN}[x] = \texttt{if atx then x} \quad \texttt{else FLATTEN}[ax] * \text{FLATTEN}[dx]
  \]
TRADEOFF: EXTRA ARG VS EFFICIENCY

• Common when creating LISP functions
  Ex: factorial:
  \[ FACT\[x\]= \]

  with the addition of a second argument:
  \[ FACT1\[x, y\]= \]

• The general case
  1. note similarity of transformation in \textsc{reverse} and \textsc{fact}
  2. consider these schemas
    \[
    f(x) = \text{if } p(x) \text{ then } a \quad \Rightarrow \quad h(x, y) = \text{if } p(x) \text{ then } y \oplus a
    \\
    \text{else } b \oplus f(g(x)) \quad \text{else } (h(g(x), y \oplus b)
    \]
    \[
    f(x) = \text{if } p(x) \text{ then } a \quad \Rightarrow \quad h(x, y) = \text{if } p(x) \text{ then } a \oplus y
    \\
    \text{else } f(g(x)) \oplus b \quad \text{else } (h(g(x), b \oplus y)
    \]
    with \( f(x) \equiv h(x, \text{id} \oplus) \) and \text{id} \oplus is identity element of \oplus op
  3. when are these transformations valid?
    • \( \oplus \) must be

  4. Ex: \textsc{reverse}:
    \( p \) is null
    \( a \) is \text{NIL}
    \( b \) is \text{CAR x}
    \( g \) is \text{CDR}
    \( \oplus \) is \text{APPEND}
    \( b \oplus y \equiv \text{CAR x} \oplus \text{APPEND y} \equiv a \times \text{CONS y} \) (since \( a \times x \) is atom)

  5. Ex: \textsc{factorial}:
    \( p \) is \( x \) eq 1
    \( a \) is 1
    \( b \) is \( x \)
    \( g \) is \( x - 1 \)
    \( \oplus \) is multiplication
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:
  \[
  \text{FACT1}[x, y] =
  \]

• The general case
  1. note similarity of transformation in REVERSE and FACT
  2. consider these schemas
     \[
     f(x) = \begin{cases} 
     \text{if } p(x) \text{ then } a \\
     b \oplus f(g(x)) 
     \end{cases} \quad \Rightarrow \quad h(x, y) = \begin{cases} 
     \text{if } p(x) \text{ then } y \oplus a \\
     (h(g(x), y \oplus b) \text{ else})
     \end{cases}
     \]
     \[
     f(x) = \begin{cases} 
     \text{if } p(x) \text{ then } a \\
     f(g(x)) \oplus b 
     \end{cases} \quad \Rightarrow \quad h(x, y) = \begin{cases} 
     \text{if } p(x) \text{ then } a \oplus y \\
     (h(g(x), b \oplus y) \text{ else})
     \end{cases}
     \]
     with \( f(x) \equiv h(x, \text{id} \oplus) \) and \( \text{id} \oplus \) is identity element of \( \oplus \) op
  3. when are these transformations valid?
     • \( \oplus \) must be
  4. Ex: REVERSE: \( p \) is null
     \( a \) is NIL
     \( b \) is \( \text{<CAR X> \oplus} \)
     \( g \) is CDR
     \( \oplus \) is APPEND
     \( b \oplus y \equiv \text{<CAR X> \oplus} \text{APPEND} y \equiv a \text{ CONS} y \) (since \( a \text{ CONS} y \) is atom)
  5. Ex: FACTORIAL: \( p \) is \( x \) eq 1
     \( a \) is 1
     \( b \) is \( x \)
     \( g \) is \( x-1 \)
     \( \oplus \) is multiplication
TRADEOFF: EXTRA ARG VS EFFICIENCY
• Common when creating LISP functions
  Ex: factorial:
  \[ \text{FACT}[x] = \begin{cases} 
  1 & \text{if } x \text{ eq } 1 \\
  x \times \text{FACT}[x-1] & \text{else}
  \end{cases} \]

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

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

• The general case
  1. note similarity of transformation in \textsc{REVERSE} and \textsc{FACT}
  2. consider these schemas
     \[ f(x) = \begin{cases} 
  a & \text{if } p(x) \text{ then } \\
  b & \text{else}
  \end{cases} \]
     \[ h(x, y) = \begin{cases} 
  y & \text{if } p(x) \text{ then } \\
  b & \text{else}
  \end{cases} \]

     \[ f(x) = \begin{cases} 
  a & \text{if } p(x) \text{ then } \\
  b & \text{else }
  \end{cases} \]
     \[ h(x, y) = \begin{cases} 
  y & \text{if } p(x) \text{ then } \\
  b & \text{else}
  \end{cases} \]

  with \( f(x) \equiv h(x, \text{id} \oplus) \) and \( \text{id} \oplus \) is identity element of \( \oplus \) \( \text{op} \)

  3. when are these transformations valid?
     • \( \oplus \) must be

  4. Ex: \textsc{REVERSE}:
     \( p \) is null
     \( a \) is NIL
     \( b \) is \( \langle \text{CAR} \ x \rangle \)
     \( g \) is CDR
     \( \oplus \) is APPEND

     \( b \oplus y \equiv \langle \text{CAR} \ x \rangle \text{ APPEND} y \equiv a \times \text{CONS} y \) (since \( a \ x \) is atom)

  5. Ex: \textsc{FACTORIAL}:
     \( p \) is \( x \text{ eq } 1 \)
     \( a \) is 1
     \( b \) is \( x \)
     \( g \) is \( x-1 \)
     \( \oplus \) is multiplication
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 \text{REVERSE} and \text{FACT}
  
  2. consider these schemas
    
    \[
    f(x) = \begin{cases} 
    \text{if } p(x) \text{ then } a \\
    b \oplus f(g(x))
    \end{cases}
    \Rightarrow
    h(x, y) = \begin{cases} 
    \text{if } p(x) \text{ then } y \oplus a \\
    (h(g(x), y \oplus b)
    \end{cases}
    \]

    \[
    f(x) = \begin{cases} 
    \text{if } p(x) \text{ then } a \\
    f(g(x)) \oplus b
    \end{cases}
    \Rightarrow
    h(x, y) = \begin{cases} 
    \text{if } p(x) \text{ then } a \oplus y \\
    (h(g(x), b \oplus y)
    \end{cases}
    \]

    with \(f(x) \equiv h(x, \text{id} \oplus)\) and \(\text{id} \oplus\) is identity element of \(\oplus\ op\)

  3. when are these transformations valid?
    
    - \(\oplus\) must be \textit{associative}

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

  5. Ex: \text{FACTORIAL}:
    
    \(p\) is \(x\) eq \(1\)
    \(a\) is \(1\)
    \(b\) is \(x\)
    \(g\) is \(x-1\)
    \(\oplus\) is multiplication
GREATEST COMMON DENOMINATOR

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

• recursively:

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

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 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 NIL is returned

final case:
base case:
induction step:

ASSOC[x, d] =

- 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 significantly more complex
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)\)
     \(((x \text{ PLUS A B})(y \cdot c)(z \text{ TIMES U V}))\)

- Lookup using \textsc{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 \textsc{nil} is returned

final case: \textbf{if} \textsc{nl} \textbf{then} \textsc{nil}

base case:

induction step:

\textsc{assoc} [ x, d ] =

- 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
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 \texttt{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 \texttt{NIL} is returned

  final case: \texttt{if nl then nil}
  base case: \texttt{if x eq aal then al}
  induction step:

  \texttt{ASSOC}[x,d] =

• 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
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 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 NIL is returned

  final case: if nl then nil
  base case: if x eq aal then al
  induction step: ASSOC[x,dl]

  ASSOC[x,d]=

• 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 significantly more complex
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 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 NIL is returned

  final case: if nl then nil
  base case: if x eq aal then al
  induction step: ASSOC[x,dl]

  ASSOC[x,d] = if nl then nil
               else if x eq aal then al
               else ASSOC[x,dl]

- 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 significantly more complex
INTERNAL LAMBDA

• Avoid computing a function twice
• Compute once and store for future reference
• Ex: ((LAMBDA (x y) (PLUS (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 ASSOC(x,l) yields a non-NIL result and we want the actual definition of x
  2. recall ASSOC returns NIL or entire entry including the atom x that we looked up
  3. \lambda(pair); if n pair then NIL else d pair; (ASSOC(x,d))
  }  this is a segment

• Redefine SUBST so CONS only happens if there indeed was a substitution

SUBST2 [x, y, z] =
INTERNAL LAMBDA

• Avoid computing a function twice
• Compute once and store for future reference
• Ex: ((LAMBDA (x y) (PLUS (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\times x + y

• Ex: using ASSOC to substitute a dictionary value
  1. if use of ASSOC(x,l) yields a non-NIL result and we want the actual definition of x
  2. recall ASSOC returns NIL or entire entry including the atom x that we looked up
  3. λ(pair); if n pair then NIL else d pair; (ASSOC(x,d)) this is a segment

• Redefine SUBST so CONS only happens if there indeed was a substitution

\[
\text{SUBST2}[x, y, z] = \\
\text{if atz then} \\
\quad \text{if z eq y then x} \\
\quad \text{else z} \\
\quad \text{else LAMBDA (head, tail);} \\
\quad \text{if head eq az and tail eq dz then z} \\
\quad \text{else head.tail;} \\
\quad (\text{SUBST2}[x, y, az], \text{SUBST2}[x, y, dz])
\]
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 ← substz(x,y,car(z));
      tail ← 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:
  - **Value cell** contains value bound to the atom
    i.e. \( \text{(SETQ } A \text{ (QUOTE (JOHN MARY))} \) means that the value of \( A \) is \( \text{(JOHN MARY)} \)
  - **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:

    ```
    Value Cell → "Mary Jones" → Rank Home Captain California
    Single
    Years College Branch
    8 Maryland Marines
    ```

    is represented as:

```
```

Copyright © 2003 by Hanan Samet
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 \texttt{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 \texttt{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.,
   \( (\text{UNFLAG (QUOTE CASE) (QUOTE RECESSED)}) \)