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 = (PLUS (TIMES x y) x 3)` we have

  `L[1] =`

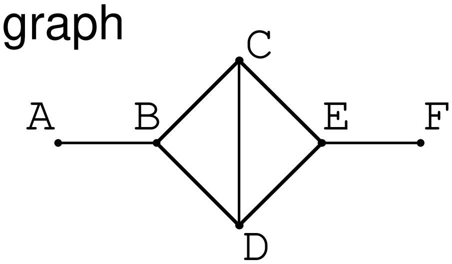
  `L[2] =`

  `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 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`

- We can refer to elements of a list by using brackets:
  for `L = (PLUS (TIMES x y) x 3)` we have

  `L[1] =`

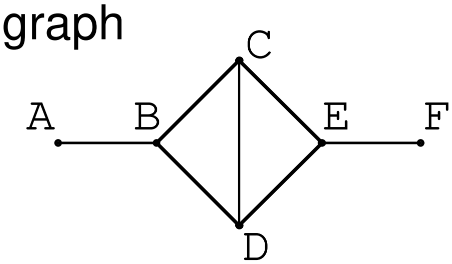
  `L[2] =`

  `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 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] =`

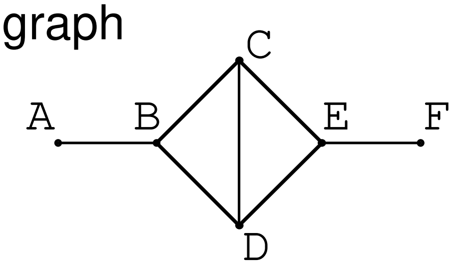
  `L[2] =`

  `L[2,2] =`

  `L[4] =`

  $(\exists x)(\forall y)\ P(x) \supset P(y)$

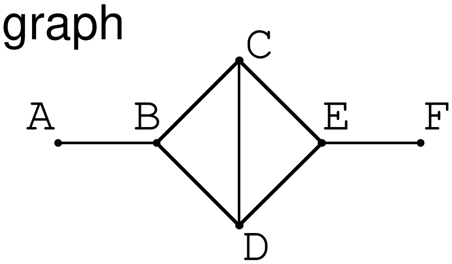
- An undirected graph



- 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 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] =`
  `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

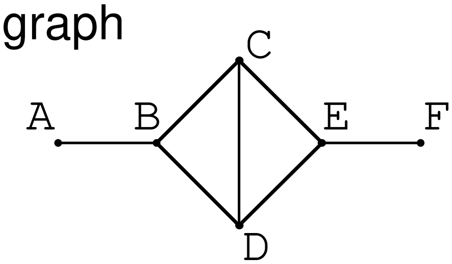
  L[2] = (TIMES x y)

  L[2,2] =

  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`

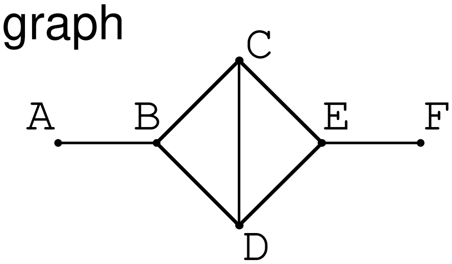
  `L[2] = (TIMES x y)`

  `L[2,2] = x`

  `L[4] =`

  $(\exists x)(\forall y)\ P(x) \supset P(y)$

- 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`

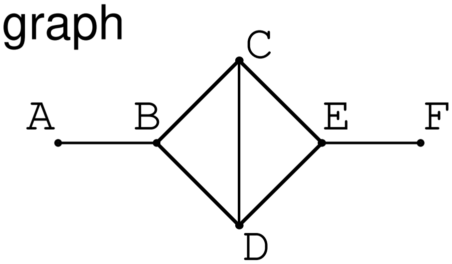
  `L[2] = (TIMES x y)`

  `L[2,2] = x`

  `L[4] = 3`

  $(\exists x)(\forall y)\ P(x) \supset P(y)$

- 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`

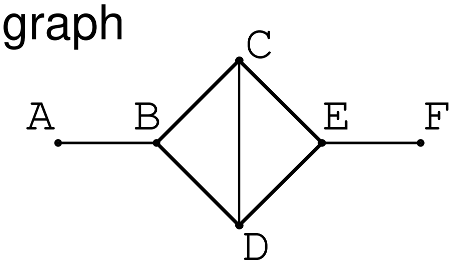
  `L[2] = (TIMES x y)`

  `L[2,2] = x`

  `L[4] = 3`

  $(\exists x)(\forall y)\ P(x) \supset P(y)$

  `(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 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)

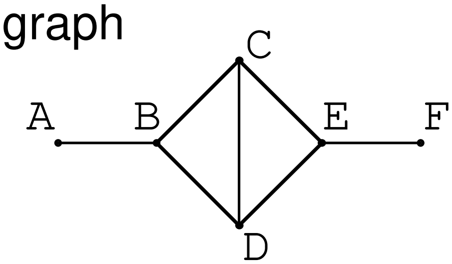
  L[2,2] = x

  L[4] = 3

  $(\exists x)(\forall y)\ P(x) \supset P(y)$

  (EXIST x (ALL y (IMPLIES (P x)(P y))))

- An undirected graph



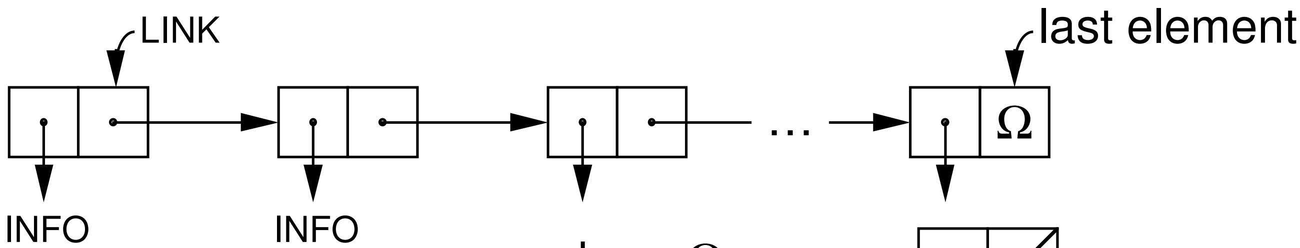
- Questions:
  1. How would we represent it?
  2. What do we want to know?
  3. What node is connected to what node?

- Solution: list of lists where first element of each
  list is connected to rest

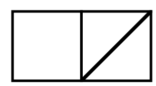
  ((A B)(B A C D)(C B D E)
  (D B C E)(E C D F)(F E))

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:



where Ω = NIL or

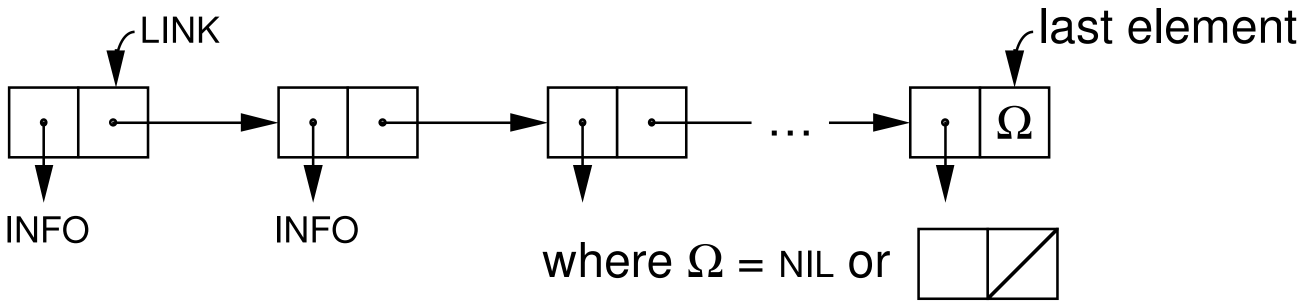


`(A B C)` would be:

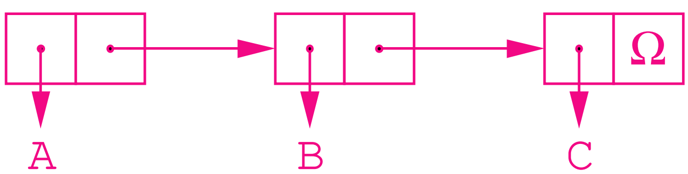
- What about `xy+x+3` or `(PLUS (TIMES x y) x 3)` ?
- Solution: INFO points to another list!

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:



`(A B C)` would be:



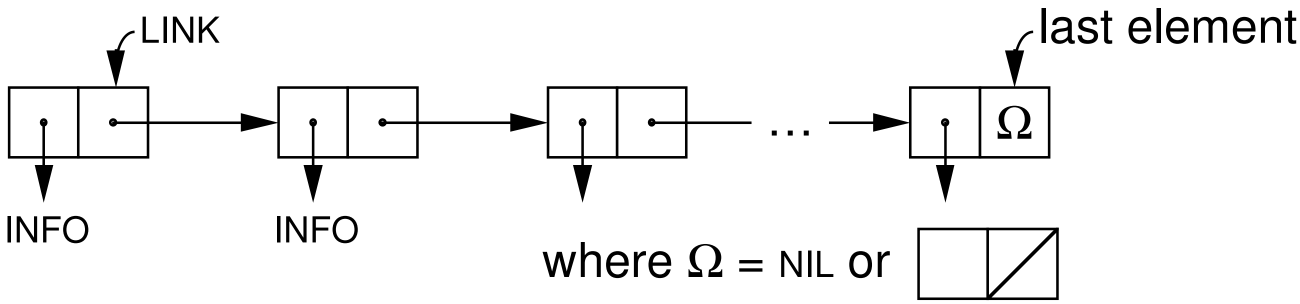
- What about `xy+x+3` or `(PLUS (TIMES x y) x 3)` ?
- Solution: INFO points to another list!

REPRESENTATION OF A LIST

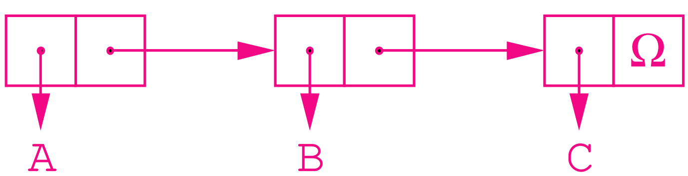
- Components of lists can be *atoms*
  1. any sequence of characters not including spaces or parentheses
  2. examples: `x y 345 A37 A-B-C`
     `376-80-5763 80.8 ...`

- How would we represent a list?

- In earlier work we used:

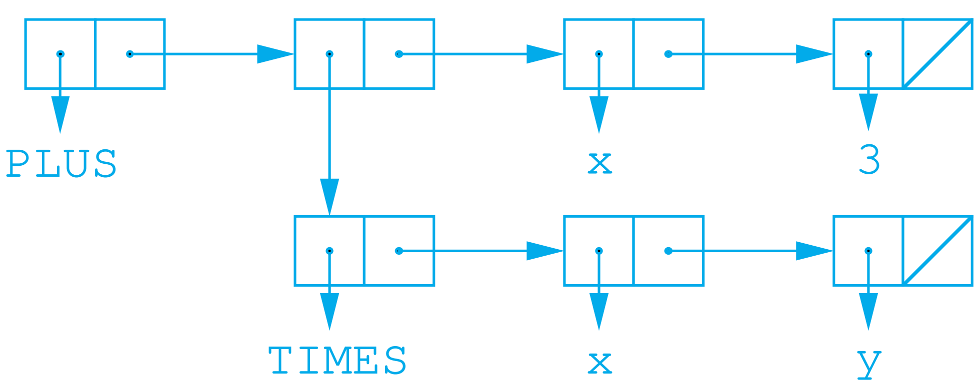


`(A B C)` would be:



- What about `xy+x+3` or `(PLUS (TIMES x y) x 3)` ?

- Solution: INFO points to another list!



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

  <sexpr> $\Rightarrow$ <atom> | (<sexpr> . <sexpr>)

- Examples:

  `A`

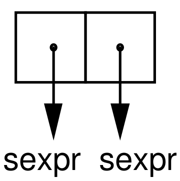
  `(A.B)`

  `(A.(B.A))`

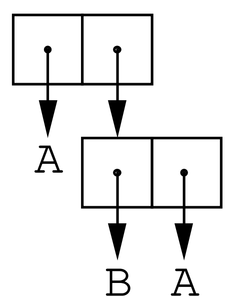
  `(3 . 3.4)` note convention about decimal point and dot: ␣.␣ is not a decimal point. Space around dot may be omitted if no confusion results:

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

- Represented in computer memory by:



`(A.B)` $\rightarrow$ `(A.(B.A))` $\rightarrow$



`(PLUS.(x.(y.NIL)))` $\rightarrow$

- 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`

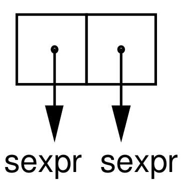
  `(A.B)`

  `(A.(B.A))`

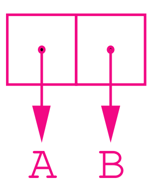
  `(3 . 3.4)` note convention about decimal point and dot: ␣.␣ is not a decimal point. Space around dot may be omitted if no confusion results:

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

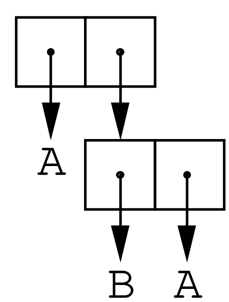
- Represented in computer memory by:



`(A.B)` →



`(A.(B.A))` →



`(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> $\Rightarrow$ <atom> | (<sexpr> . <sexpr>)

- Examples:

  `A`

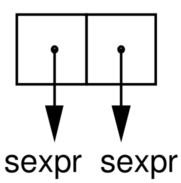
  `(A.B)`

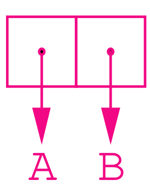
  `(A.(B.A))`

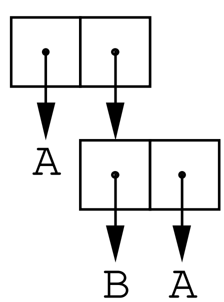
  `(3 . 3.4)` note convention about decimal point and dot: ␣ . ␣ is not a decimal point. Space around dot may be omitted if no confusion results:

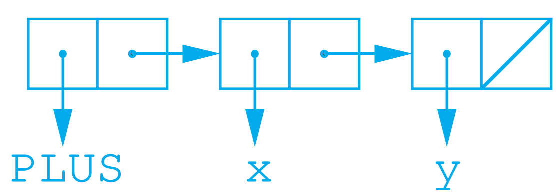
  `(PLUS.(x.(y.NIL)))`

- Represented in computer memory by:



`(A.B)` $\rightarrow$ 

`(A.(B.A))` $\rightarrow$ 

`(PLUS.(x.(y.NIL)))` $\rightarrow$ 

- This should be familiar

S-EXPRESSIONS

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

  <sexpr> $\Rightarrow$ <atom> | (<sexpr> . <sexpr>)

- Examples:

  `A`

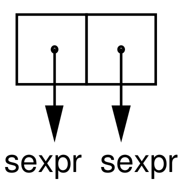
  `(A.B)`

  `(A.(B.A))`

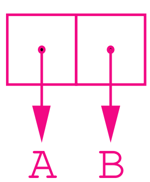
  `(3 . 3.4)` note convention about decimal point and dot: ␣.␣ is not a decimal point. Space around dot may be omitted if no confusion results:

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

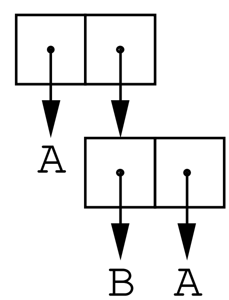
- Represented in computer memory by:



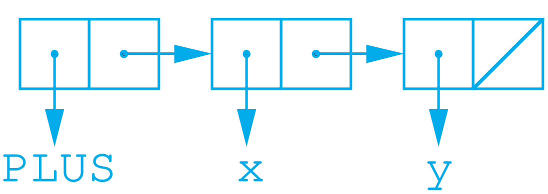
`(A.B)` →



`(A.(B.A))` →



`(PLUS.(x.(y.NIL)))` →



- This should be familiar `(PLUS x y)`

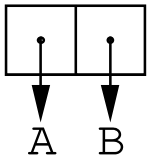
THE LISP PROGRAMMING LANGUAGE

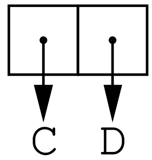
- Easy to learn – just a few primitive operations
  1. CAR (Contents of Address Register)
     - first element of list
     - sometimes called *head*
     - sometimes written as `a x`
     - refers to left part of an s-expression
  2. CDR (Contents of Decrement Register)
     - remainder of list after 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

       CDR of `(A (B C)` → `(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:
     - `(CDR(QUOTE(A B C)))`
     - `(CDR '(A B C))`
     - `CDR('(A B C))`
     - `CDR['(A B C)]`
     - `CDR[(QUOTE (A B C))]`
     - use [ ] when args quoted or in definition of recursive function, use ( ) otherwise
  4. CONS (CONStruct)
     - creates an s-expression from two s-expressions
     - alternatively, adds atom or list to head of another list

       Ex: `CONS['A,'(B C D)]` ≡ `(A B C D)` ≡
       `CONS['A,'(B.(C.(D.NIL)))]` ≡
       `(A.(B.(C.(D.NIL))))`

LISP EXAMPLES

`(A.B)` 

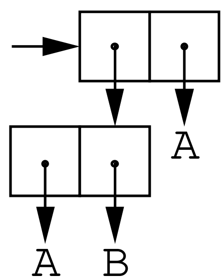
`(C.D)` 

`CONS['(A.B),'(C.D)] =`

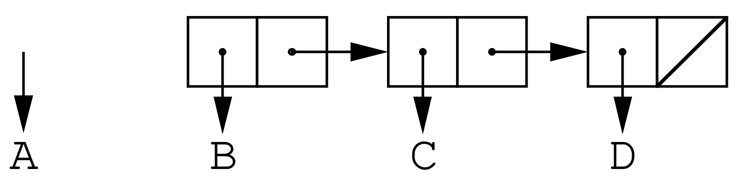
```
CAR['((A.B).(C.D))] =
CAR[CAR '((A.B).(C.D))] =
CAAR['((A.B).(C.D))] =
```

- 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

`CONS['(A.B),'A] = ((A.B).A)`

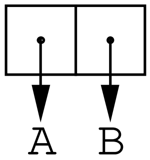


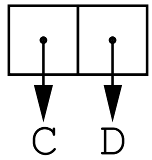
`CONS['A,'(B C D)] = (A B C D)`



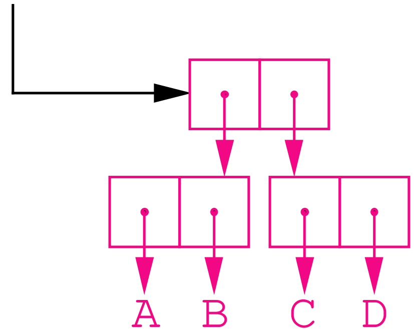
Important:
```
CAR[CONS['A,'B]] =
CDR[CONS['A,'B]] =
CONS[CAR['(A.B)],CDR['(A.B)]] =
```

LISP EXAMPLES

(A.B) 

(C.D) 

CONS['(A.B),'(C.D)] =



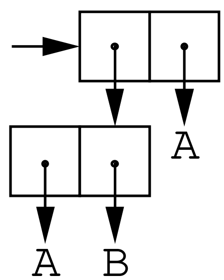
CAR['((A.B).(C.D))] =

CAR[CAR '((A.B).(C.D))] =

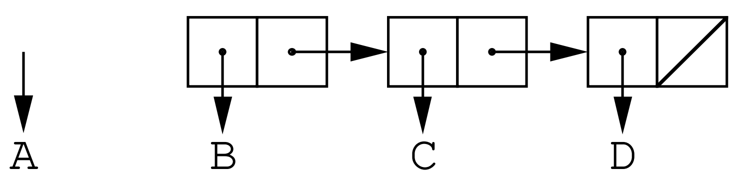
CAAR['((A.B).(C.D))] =

- 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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

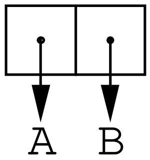
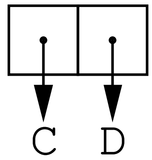


Important: CAR[CONS['A,'B]] =

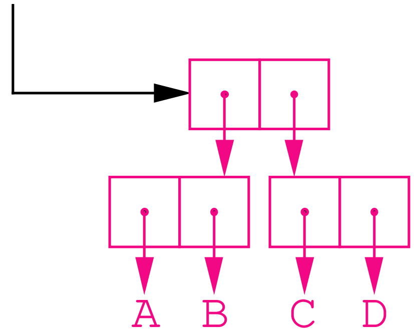
CDR[CONS['A,'B]] =

CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

`(A.B)`  `(C.D)` 

`CONS['(A.B),'(C.D)] =`



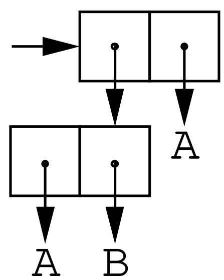
`CAR['((A.B).(C.D))] = (A.B)`

`CAR[CAR '((A.B).(C.D))] =`

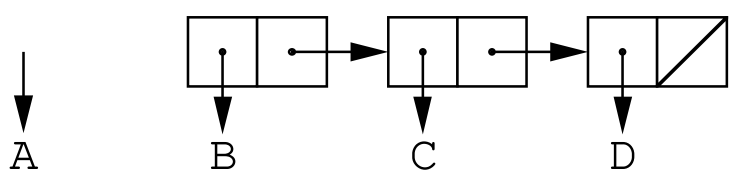
`CAAR['((A.B).(C.D))] =`

- 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

`CONS['(A.B),'A] = ((A.B).A)`



`CONS['A,'(B C D)] = (A B C D)`



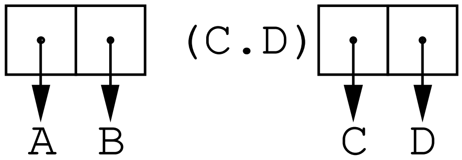
Important: `CAR[CONS['A,'B]] =`

`CDR[CONS['A,'B]] =`

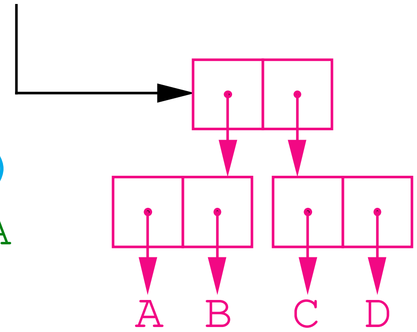
`CONS[CAR['(A.B)],CDR['(A.B)]] =`

LISP EXAMPLES

(A.B) (C.D)



CONS['(A.B),'(C.D)] =



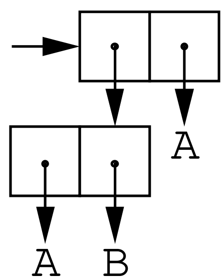
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

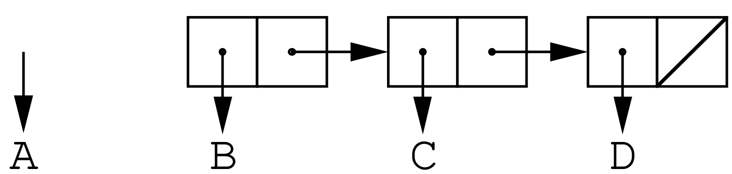
CAAR['((A.B).(C.D))] =

- 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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

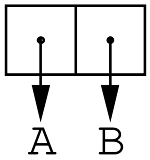


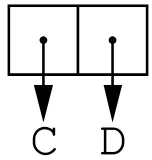
Important: CAR[CONS['A,'B]] =

CDR[CONS['A,'B]] =

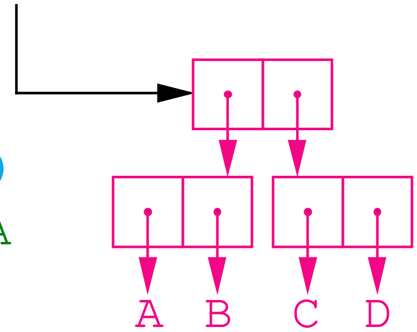
CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

(A.B) 

(C.D) 

CONS['(A.B),'(C.D)] =



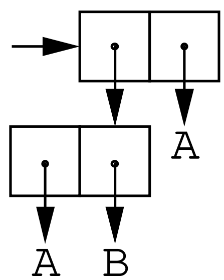
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

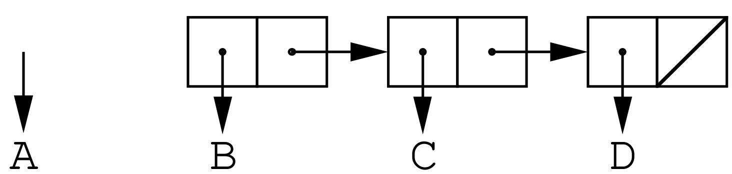
CAAR['((A.B).(C.D))] = A

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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

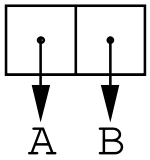


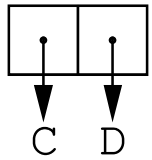
Important: CAR[CONS['A,'B]] =

CDR[CONS['A,'B]] =

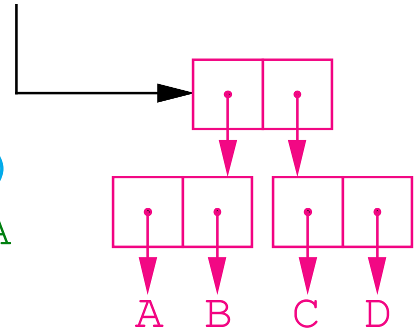
CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

(A.B) 

(C.D) 

CONS['(A.B),'(C.D)] =



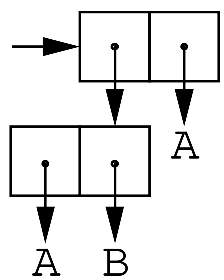
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

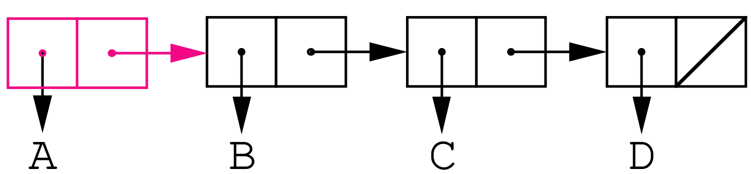
CAAR['((A.B).(C.D))] = A

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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

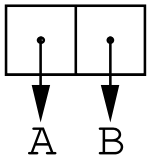
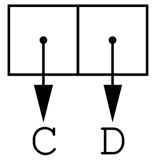


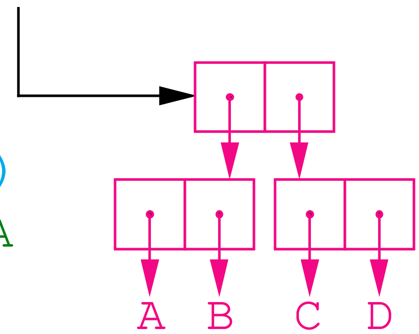
Important: CAR[CONS['A,'B]] =

CDR[CONS['A,'B]] =

CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

(A.B)  (C.D)  CONS['(A.B),'(C.D)] =



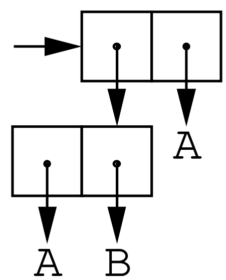
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

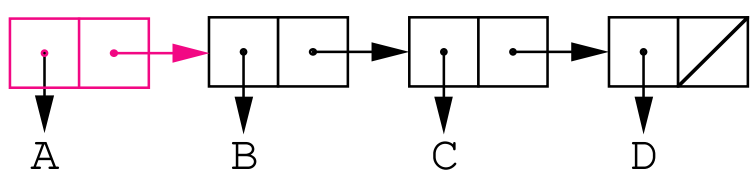
CAAR['((A.B).(C.D))] = A

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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

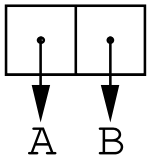


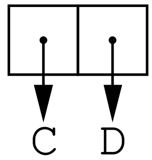
Important: CAR[CONS['A,'B]] = A

CDR[CONS['A,'B]] =

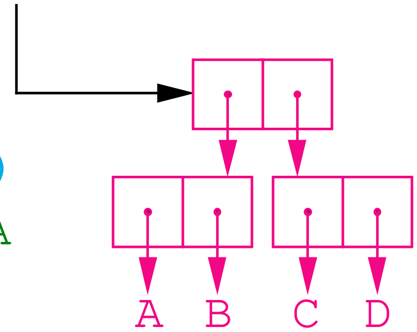
CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

(A.B) 

(C.D) 

CONS['(A.B),'(C.D)] =



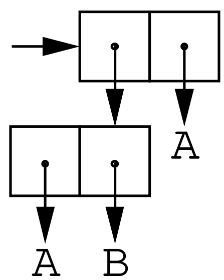
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

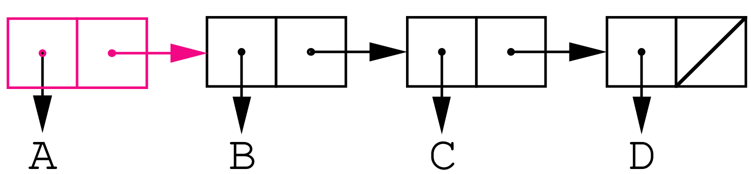
CAAR['((A.B).(C.D))] = A

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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)

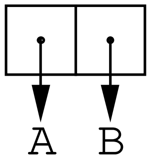


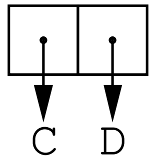
Important: CAR[CONS['A,'B]] = A

CDR[CONS['A,'B]] = B

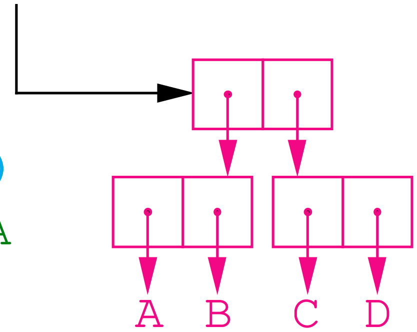
CONS[CAR['(A.B)],CDR['(A.B)]] =

LISP EXAMPLES

(A.B) 

(C.D) 

CONS['(A.B),'(C.D)] =



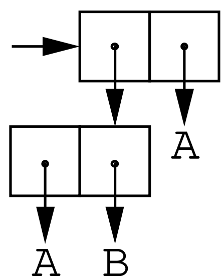
CAR['((A.B).(C.D))] = (A.B)

CAR[CAR '((A.B).(C.D))] = A

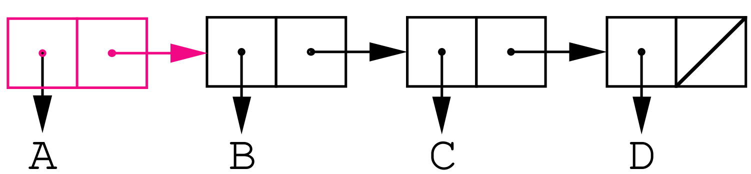
CAAR['((A.B).(C.D))] = A

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

CONS['(A.B),'A] = ((A.B).A)



CONS['A,'(B C D)] = (A B C D)



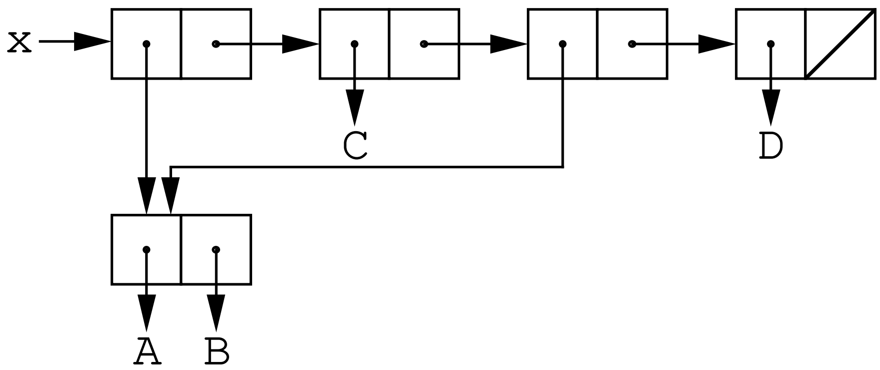
Important: CAR[CONS['A,'B]] = A

CDR[CONS['A,'B]] = B

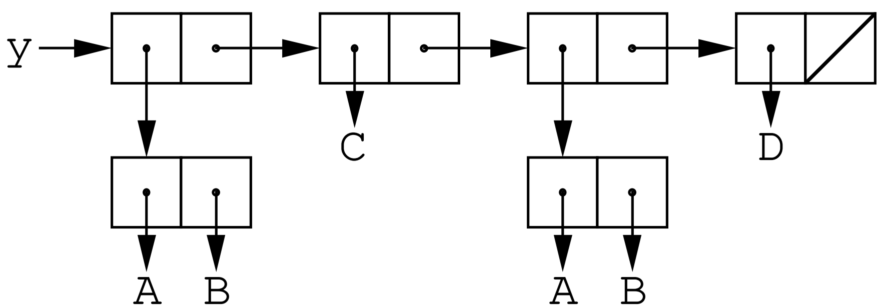
CONS[CAR['(A.B)],CDR['(A.B)]] = (A.B)

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←(CONS 'A 'B)` then:

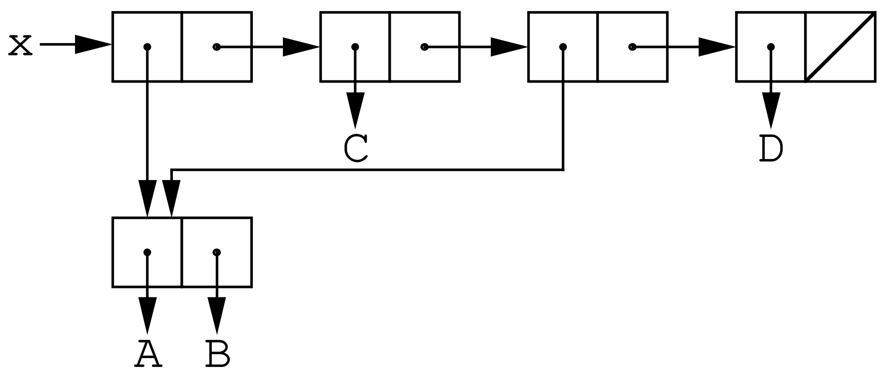
`x←`

i.e.,

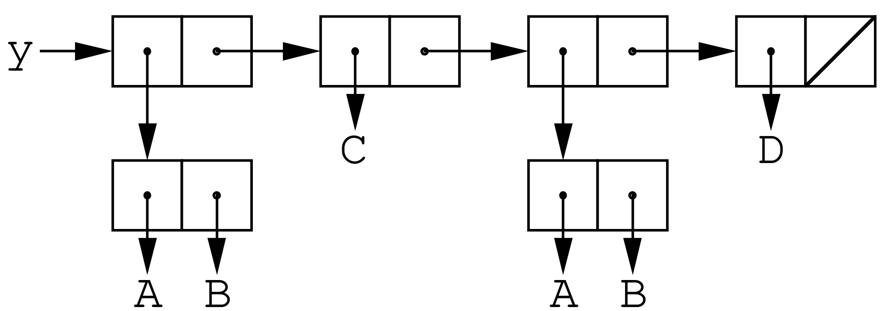
`y←`

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←(CONS 'A 'B)` then:

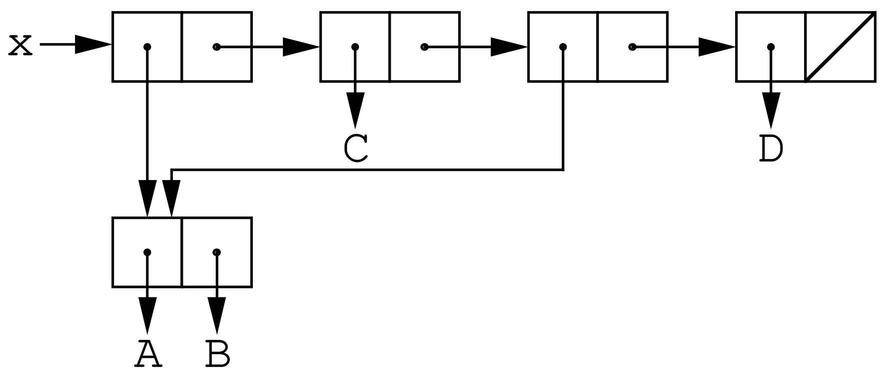
`x←(CONS z (CONS 'C (CONS z (CONS 'D NIL))))`

i.e.,

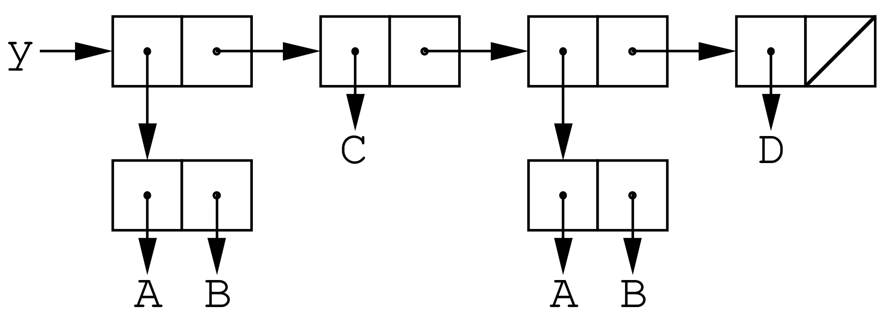
`y←`

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←(CONS 'A 'B)` then:

`x←(CONS z (CONS 'C (CONS z (CONS 'D NIL))))`

i.e.,

`y←(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
     `EQ[CAR(x),CADDR(x)]=` atom denoting value True
     `EQ[CAR(y),CADDR(y)]=` just like False
  2. atoms are uniquely represented
     `EQ[CADR(x),CADR(y)]=` while
     `EQ[CAR(x),CAR(y)]=` 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] =`

     - 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)]=` just like False
  2. atoms are uniquely represented
     `EQ[CADR(x),CADR(y)]=` while
     `EQ[CAR(x),CAR(y)]=` 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] =`

     - 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)]=` while
     `EQ[CAR(x),CAR(y)]=` 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] =`

     - 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)]=` 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] =`

     - 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)]= NIL` 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] =`

     - 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)]= NIL` and
     `EQ[CAAR(x),CAAR(y)]= T =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] =`

     - 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)]= NIL` and
     `EQ[CAAR(x),CAAR(y)]= T =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: `atom(x)` $\Rightarrow$ `atom(y)`
     otherwise `NIL`

     - if either `x` or `y` are atoms, then `EQ(x,y)`
       otherwise `EQUAL` first parts and `EQUAL` second parts

  3. need way to find out if something is an atom
     - use the ATOM function
  4. thus `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)]= NIL` and
     `EQ[CAAR(x),CAAR(y)]= T =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: `atom(x)` ⇒ `atom(y)`
     otherwise `NIL`

     - if either `x` or `y` are atoms, then `EQ(x,y)`
       otherwise `EQUAL` first parts and `EQUAL` second parts
  3. need way to find out if something is an atom
     - use the ATOM function
  4. thus `EQUAL[x,y] =`

```
if ATOM(x) or ATOM(y) then eq(x,y)
else EQUAL[CAR(x),CAR(y)] and
     EQUAL[CDR(x),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 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)]= NIL` and
     `EQ[CAAR(x),CAAR(y)]= T =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: `atom(x)` ⇒ `atom(y)`
     otherwise `NIL`

     - if either `x` or `y` are atoms, then `EQ(x,y)`
       otherwise `EQUAL` first parts and `EQUAL` second parts

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

```
if ATOM(x) or ATOM(y) then eq(x,y)
else EQUAL[CAR(x),CAR(y)] and
     EQUAL[CDR(x),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:

  `EQUAL(x,y)`

  `NULL(x)` which is `EQ(x,NIL)` also written as <u>n</u> x

- Other abbreviations:

<u>a</u> x	for	`CAR(x)`
<u>a</u> <u>d</u> x	for	`CAR(CDR(x))`
`x.y`	for	`CONS(x,y)`

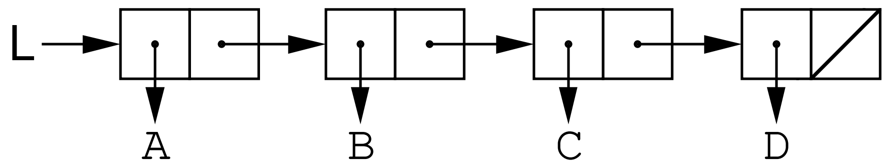
- The LIST function
  1. takes arbitrary number of arguments and returns a list containing these arguments
  2. Ex: `LIST(x,y,z)` is `(x y z)`
  3. corresponds to composition of CONS operations

     `LIST(x)` is `CONS(x,NIL)`

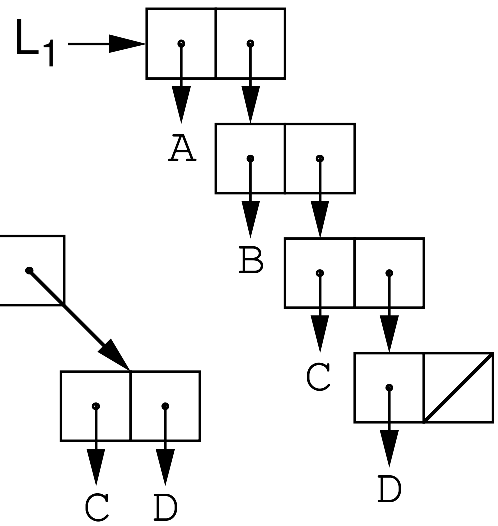
     `LIST(x,y)` is `CONS(x,CONS(y,NIL))`
  4. also written as `<x,y,z>`

REPRESENTING TREES

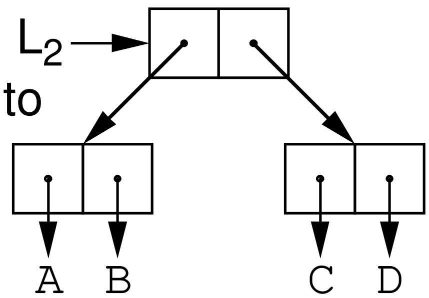
- Linear:



- 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:

  `CAR(L1) =`

  `CADR(L1) =`

  `CADDR(L1) =`

  `CADDDR(L1) =`

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

- In balanced representation:

  `CAAR(L2) =`

  `CDAR(L2) =`

  `CADR(L2) =`

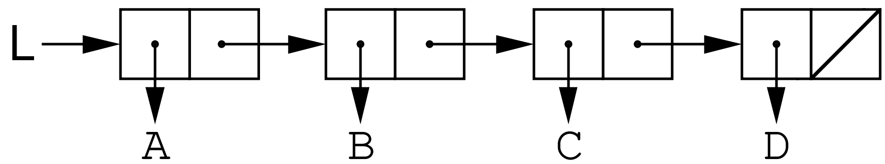
  `CDDR(L2) =`

  average of 2 operations

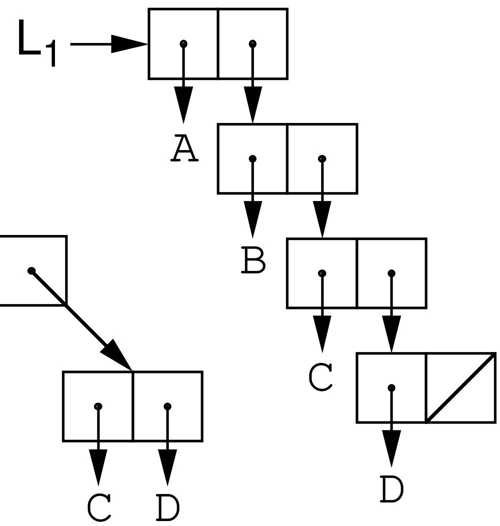
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

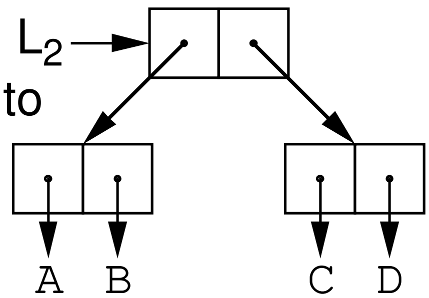
- Linear:



- 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:

  `CAR(L1) = A`

  `CADR(L1) =`

  `CADDR(L1) =`

  `CADDDR(L1) =`

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

- In balanced representation:

  `CAAR(L2) =`

  `CDAR(L2) =`

  `CADR(L2) =`

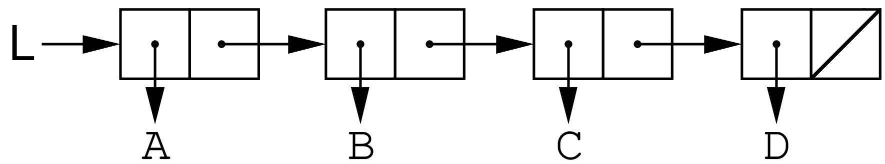
  `CDDR(L2) =`

  average of 2 operations

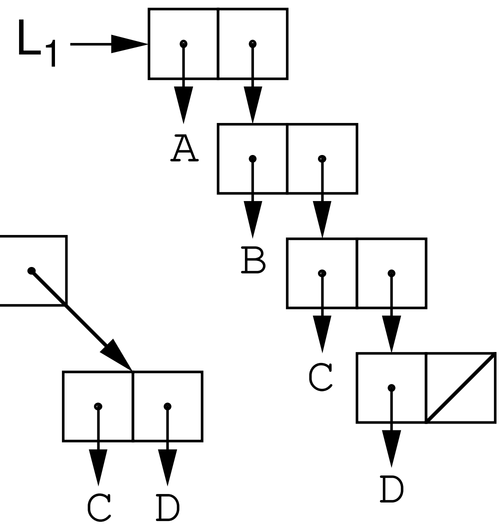
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

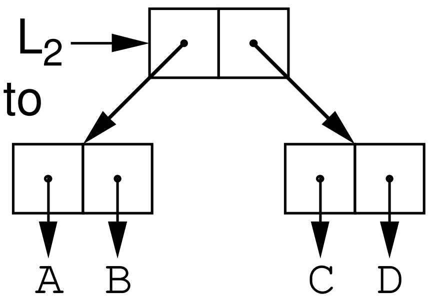
- Linear:



- 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:

  CAR(L1) = A

  CADR(L1) = B

  CADDR(L1) =

  CADDDR(L1) =

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

- In balanced representation:

  CAAR(L2) =

  CDAR(L2) =

  CADR(L2) =

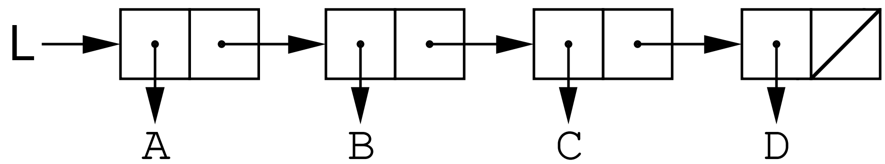
  CDDR(L2) =

  average of 2 operations

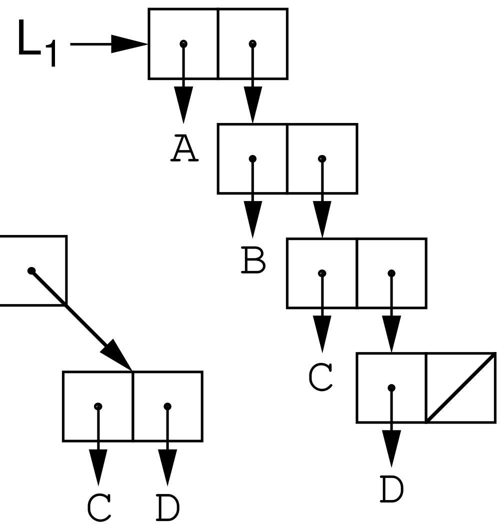
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

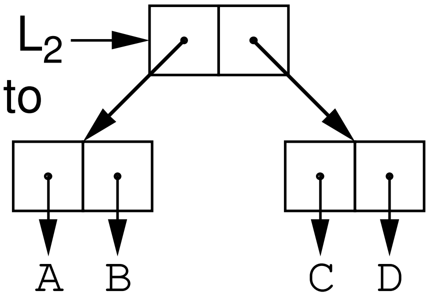
- Linear:



- 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:

  CAR(L1) = A

  CADR(L1) = B

  CADDR(L1) = C

  CADDDR(L1) =

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

- In balanced representation:

  CAAR(L2) =

  CDAR(L2) =

  CADR(L2) =

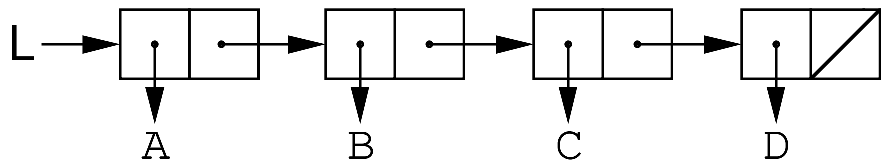
  CDDR(L2) =

  average of 2 operations

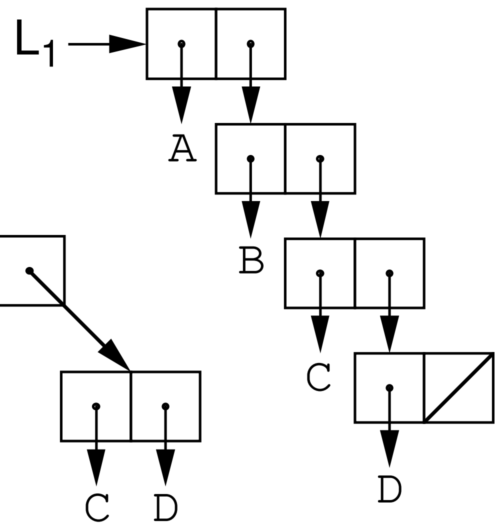
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

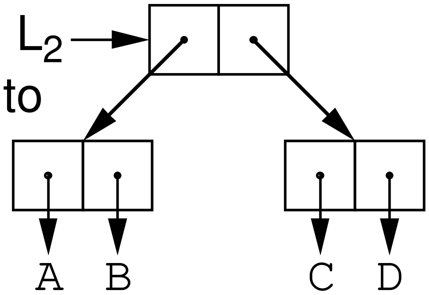
- Linear:



- 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:

  `CAR(L1) = A`

  `CADR(L1) = B`

  `CADDR(L1) = C`

  `CADDDR(L1) = D`

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

- In balanced representation:

  `CAAR(L2) =`

  `CDAR(L2) =`

  `CADR(L2) =`

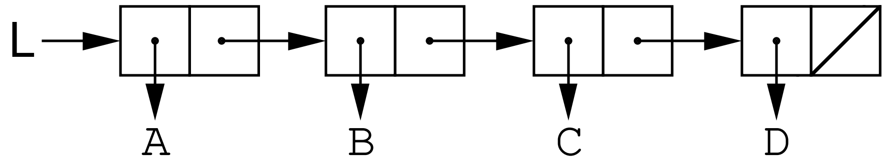
  `CDDR(L2) =`

  average of 2 operations

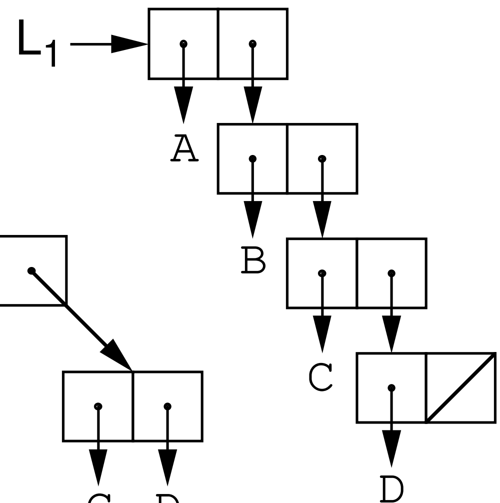
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

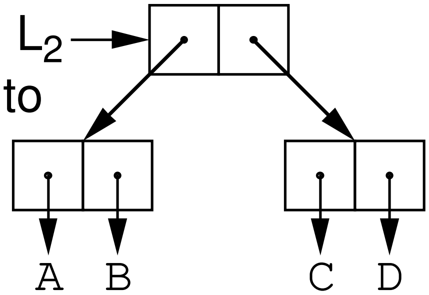
- Linear:



- 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:

  `CAR(L1) = A`

  `CADR(L1) = B`

  `CADDR(L1) = C`

  `CADDDR(L1) = D`

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

- In balanced representation:

  `CAAR(L2) = A`

  `CDAR(L2) =`

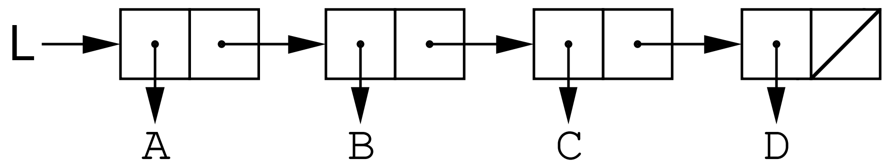
  `CADR(L2) =`

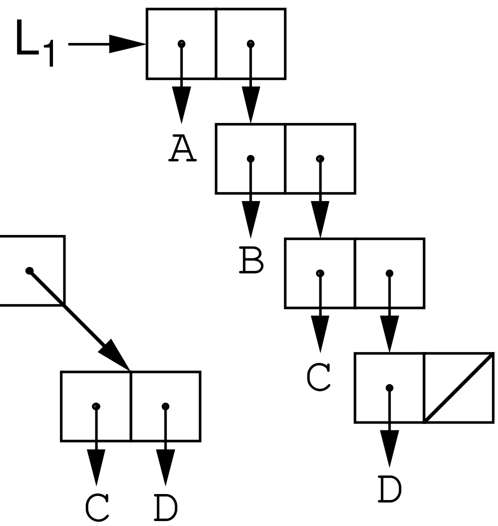
  `CDDR(L2) =`

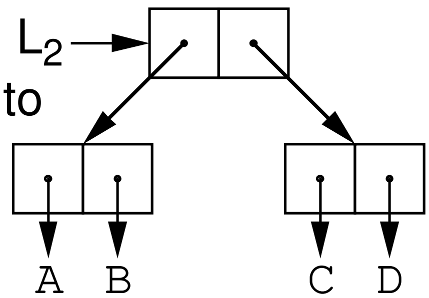
  average of 2 operations

- 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:

  `CAR(L1)  =  A`

  `CADR(L1)  =  B`

  `CADDR(L1)  =  C`

  `CADDDR(L1)  =  D`

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

- In balanced representation:

  `CAAR(L2)  =  A`

  `CDAR(L2)  =  B`

  `CADR(L2)  =`

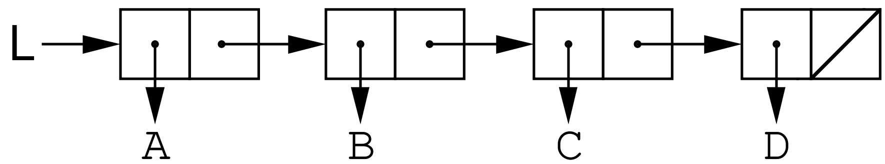
  `CDDR(L2)  =`

  average of 2 operations

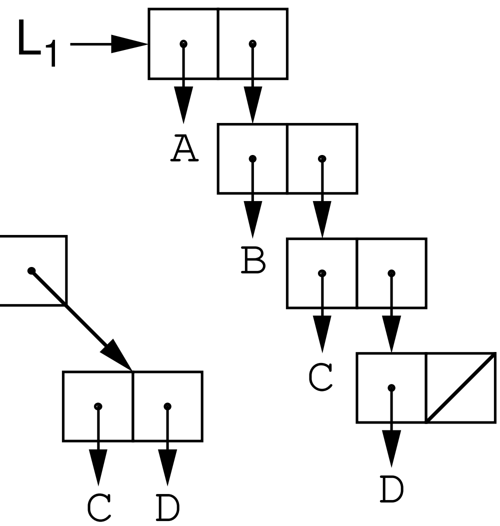
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

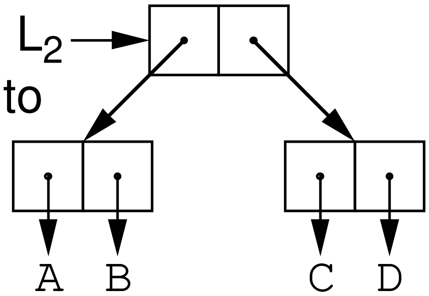
- Linear:



- 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:

  `CAR(L1) = A`

  `CADR(L1) = B`

  `CADDR(L1) = C`

  `CADDDR(L1) = D`

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

- In balanced representation:

  `CAAR(L2) = A`

  `CDAR(L2) = B`

  `CADR(L2) = C`

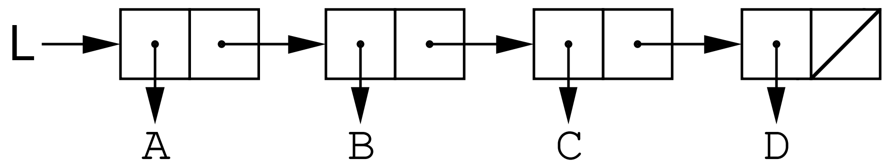
  `CDDR(L2) =`

  average of 2 operations

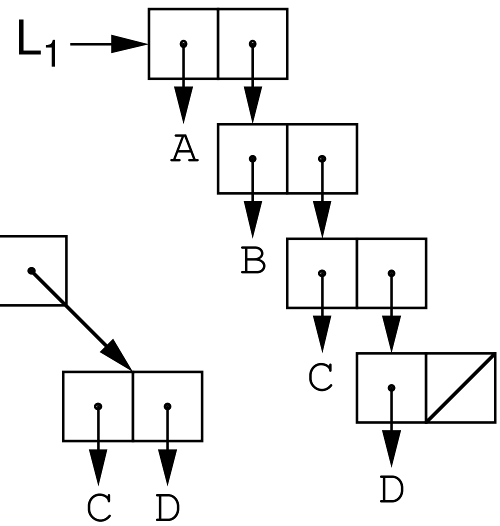
- Advantage of L_1: if searching for a particular element and list is not a fixed size, then we know when to stop

REPRESENTING TREES

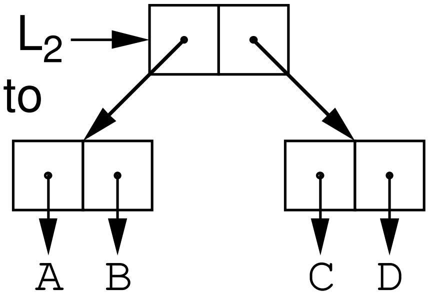
- Linear:



- 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:

  CAR(L1) = A

  CADR(L1) = B

  CADDR(L1) = C

  CADDDR(L1) = D

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

- In balanced representation:

  CAAR(L2) = A

  CDAR(L2) = B

  CADR(L2) = C

  CDDR(L2) = 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. `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: `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: `if nl then nil`

     check first element: `if al eq x then T`
  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: `if nl then nil`

     check first element: `if al eq x 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
  `(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: `if nl then nil`

     check first element: `if al eq x then T`
  2. induction:

     check rest of list: `dl`

     - `member[x,dl]`
  3.
```
member[x,l] =  if nl then nil
               else if al eq x then T
               else member[x,dl]
```

- 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: `if nl then nil`

     check first element: `if al eq x then T`
  2. induction:

     check rest of list: `dl`

     - `member[x,dl]`
  3.
```
member[x,l] =  if nl then nil
               else if al eq x then T
               else member[x,dl]
```

- 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] =
(DEF MEMBER (LAMBDA X L)
            (COND ((NULL L) NIL)
                  ((EQ X (CAR L)) T)
                  (T (MEMBER X (CDR L))))))
```

MEMBERSHIP IN S-EXPRESSION

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

```
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, CAR and CDR, to left and right subtrees, respectively

- Can we search for occurence of an entire s-expression?
- What is the terminating case (or cases)?

```
members[x,s]=
```

- Note use of 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

```
membersexpr[x,s]=
if ats then x eq s
else membersexpr[x,as] 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, CAR and CDR, to left and right subtrees, respectively

- Can we search for occurence of an entire s-expression?
- What is the terminating case (or cases)?

```
members[x,s]=
```

- Note use of 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

```
membersexpr[x,s]=
if ats then x eq s
else membersexpr[x,as] 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,
     CAR and CDR, to left and right subtrees, respectively

- Can we search for occurence of an entire s-expression?
- What is the terminating case (or cases)?

atom s $\Rightarrow$ `x eq s`

```
members[x,s]=
   if ats then x eq s
```

- Note use of 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

```
membersexpr[x,s]=
if ats then x eq s
else membersexpr[x,as] 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, CAR and CDR, to left and right subtrees, respectively

- Can we search for occurence of an entire s-expression?
- What is the terminating case (or cases)?

atom s $\Rightarrow$ `x eq s`

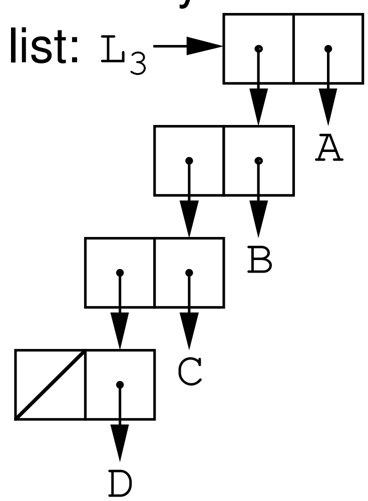
not atom s $\Rightarrow$ `x equal s or`
`members[x,al] or members[x,dl]`

```
members[x,s]=
   if ats then x eq s
   else x equal s or
         members[x,al] or members[x,dl]
```

- Note use of 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 CDR instead of by CAR?
  1. What is lisp representation of this list:



  2. Work backwards:

     CDR(L3)=

     CDAR(L3)=

     CDAAR(L3)=

     CDAAAR(L3)=

- Circular structures
  1. a list *could* point back to component of itself

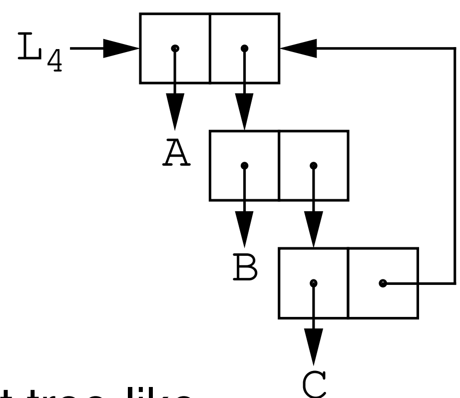
     CAR(L4)=

     CADR(L4)=

     CADDR(L4)=

     CDDDR(L4)=

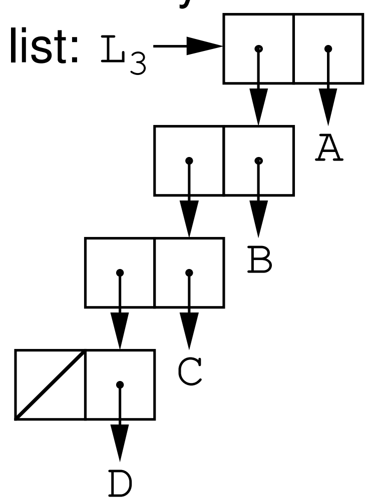
     CADDDR(L4)=



  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:

     `CDR(L3)=` A

     `CDAR(L3)=`

     `CDAAR(L3)=`

     `CDAAAR(L3)=`

- Circular structures
  1. a list *could* point back to component of itself

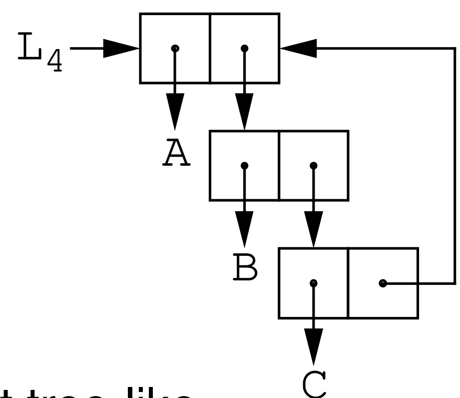
     `CAR(L4)=`

     `CADR(L4)=`

     `CADDR(L4)=`

     `CDDDR(L4)=`

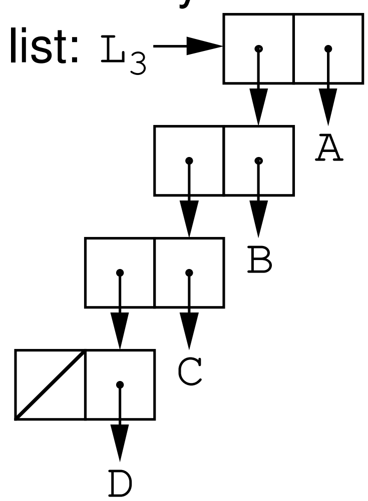
     `CADDDR(L4)=`



  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:

     CDR(L3)= A

     CDAR(L3)= B

     CDAAR(L3)=

     CDAAAR(L3)=

- Circular structures
  1. a list *could* point back to component of itself

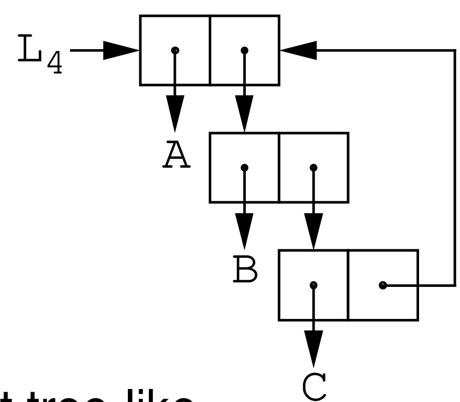
     CAR(L4)=

     CADR(L4)=

     CADDR(L4)=

     CDDDR(L4)=

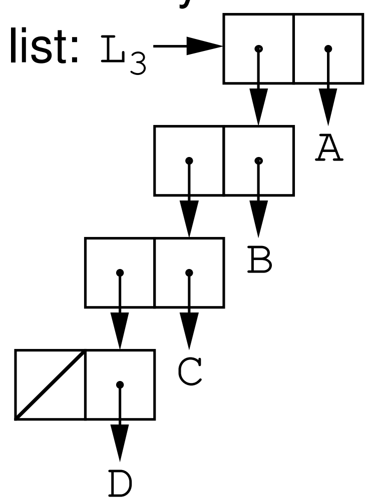
     CADDDR(L4)=



  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:



  2. Work backwards:

     CDR(L3)= A

     CDAR(L3)= B

     CDAAR(L3)= C

     CDAAAR(L3)=

- Circular structures
  1. a list *could* point back to component of itself

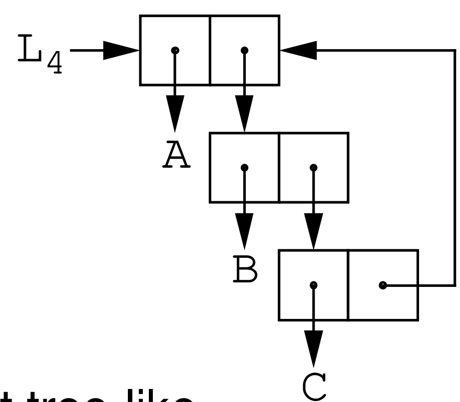
     CAR(L4)=

     CADR(L4)=

     CADDR(L4)=

     CDDDR(L4)=

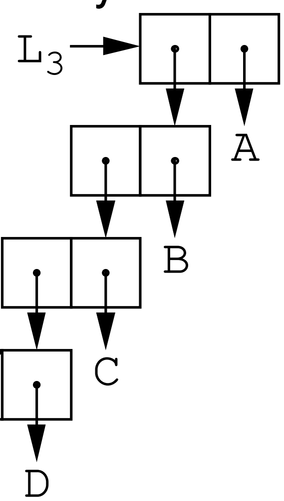
     CADDDR(L4)=



  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:



  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

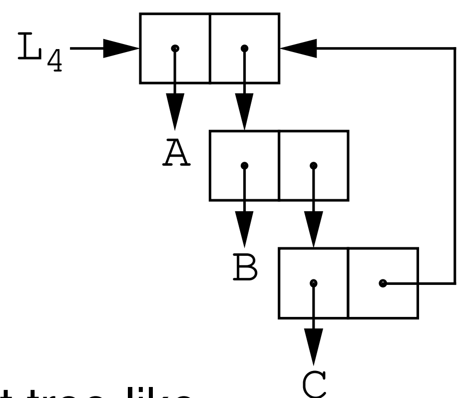
     CAR(L4)=

     CADR(L4)=

     CADDR(L4)=

     CDDDR(L4)=

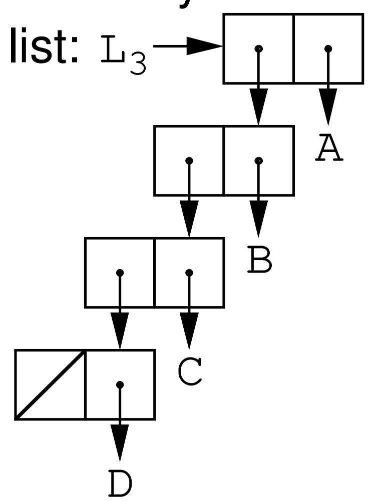
     CADDDR(L4)=



  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:



  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

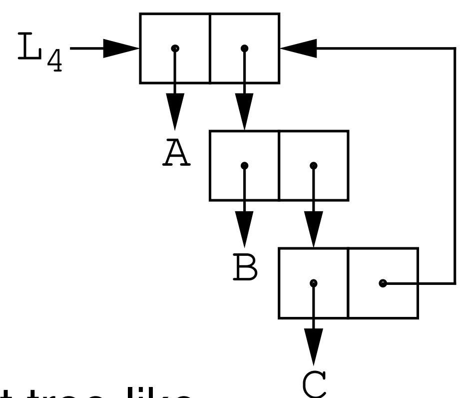
     CAR(L4)= A

     CADR(L4)=

     CADDR(L4)=

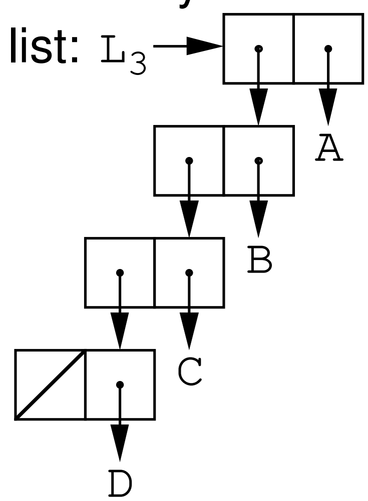
     CDDDR(L4)=

     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: 

  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

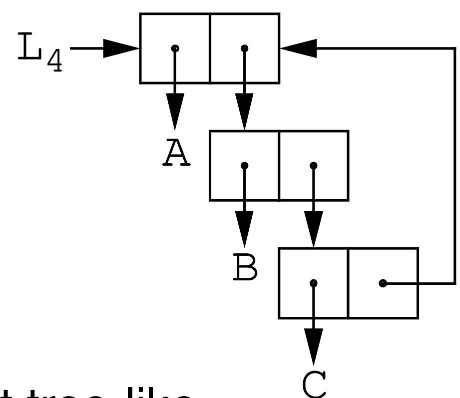
     CAR(L4)= A

     CADR(L4)= B

     CADDR(L4)=

     CDDDR(L4)=

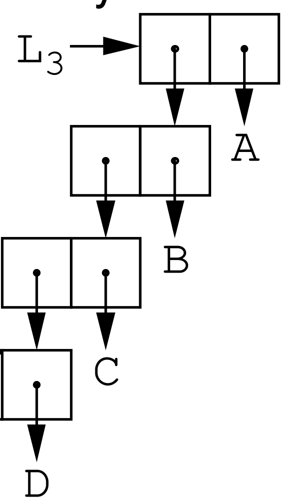
     CADDDR(L4)=



  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:



  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

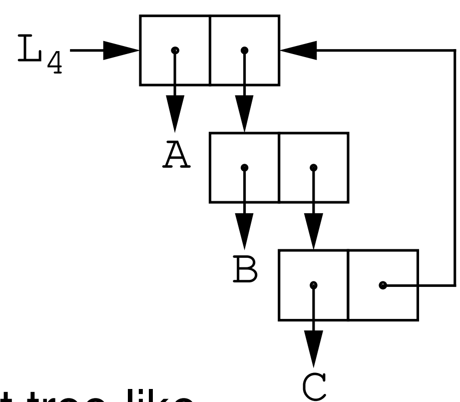
     CAR(L4)= A

     CADR(L4)= B

     CADDR(L4)= C

     CDDDR(L4)=

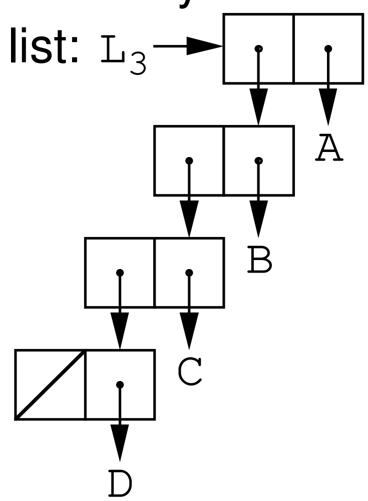
     CADDDR(L4)=



  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:

     CDR(L3)= A

     CDAR(L3)= B

     CDAAR(L3)= C

     CDAAAR(L3)= D

- Circular structures
  1. a list *could* point back to component of itself

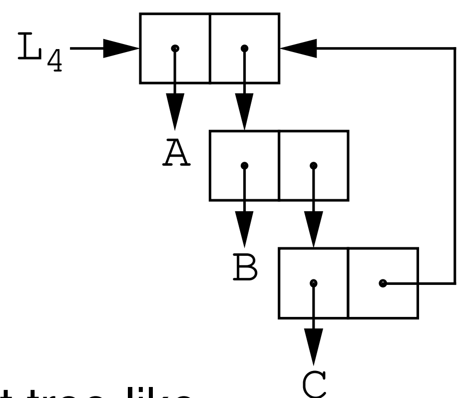
     CAR(L4)= A

     CADR(L4)= B

     CADDR(L4)= C

     CDDDR(L4)= L_4

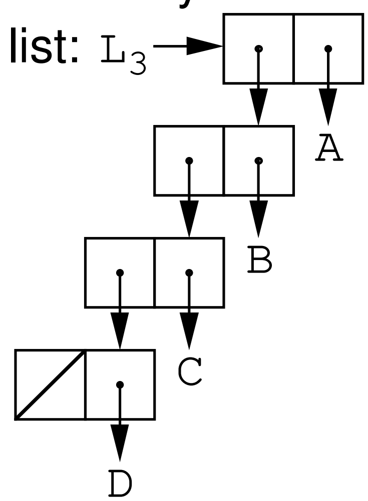
     CADDDR(L4)=



  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(L3)= A

     CDAR(L3)= B

     CDAAR(L3)= C

     CDAAAR(L3)= D

- Circular structures
  1. a list *could* point back to component of itself

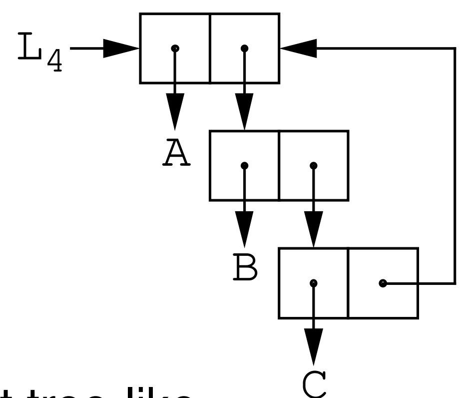
     CAR(L4)= A

     CADR(L4)= B

     CADDR(L4)= C

     CDDDR(L4)= L_4

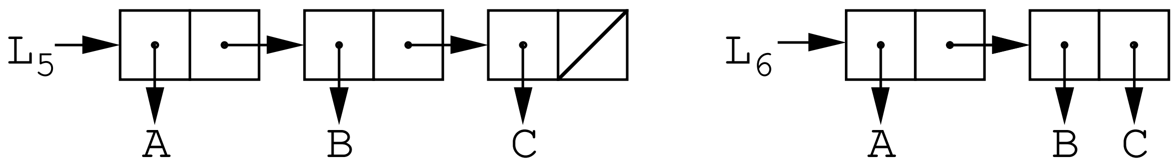
     CADDDR(L4)= A



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

EXTENDED LIST NOTATION

- Next to last element has its CDR point to last element



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

CONDITIONAL EXPRESSIONS

- Statements of the form:
  `if P then a else b` (`P` is known as a *predicate*)
- In LISP such a test is equivalent to writing:
  `if not(NULL(P)) then a else b`
- Note we are *not* testing for true, just not false (i.e., not NIL)
- More generally:

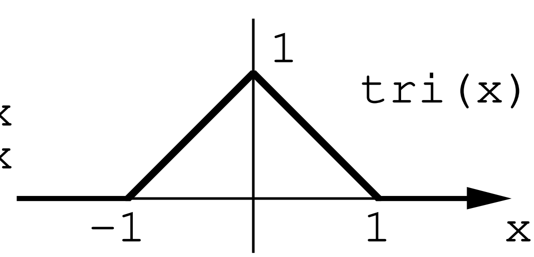
  (COND (P_1 e_{11})
  (P_2 e_{21})
  (P_3 e_{31} e_{32} e_{33})
  (P_4 e_4)
  (T e_5))

  1. basically find first non-NIL P_i and evaluate e_{i1}, e_{i2},…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 COND forms
- When writing conditional expression in LISP we have:

`(COND(P a)` `     (T b))`	if P then a else b

`(COND(P a)`
`     (Q b c)`
`     …`
`     (S d)`
`     (T e))`
	if P then a
	else if Q then b also c
	else …
	else if S then d
	else e

- Ex: $-\infty < x < -1 \Rightarrow \mathrm{tri}(x)=0$
  $-1 \le x < 0 \Rightarrow \mathrm{tri}(x)=1+x$
  $0 \le x < 1 \Rightarrow \mathrm{tri}(x)=1-x$
  $1 \le x < \infty \Rightarrow \mathrm{tri}(x)=0$
  `TRI[x] =`



CONDITIONAL EXPRESSIONS

- Statements of the form:
  `if P then a else b` (`P` is known as a *predicate*)
- In LISP such a test is equivalent to writing:
  `if not(NULL(P)) then a else b`
- Note we are *not* testing for true, just not false (i.e., not NIL)
- More generally:

```
(COND (P1 e11)
      (P2 e21)
      (P3 e31 e32 e33)
      (P4 e4)
      (T  e5))
```

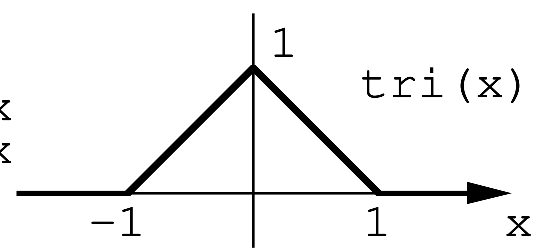
1. basically find first non-NIL P_i and evaluate $e_{i1}, e_{i2}, \ldots e_{in}$
2. return the value of the last of the e_i's – i.e., e_{in}
3. T denotes the final else
4. any of the P_i or e_{ij} could themselves be COND forms

- When writing conditional expression in LISP we have:

`(COND (P a)` `      (T b))`	if P then a else b

`(COND (P a)`
`      (Q b c)`
`      …`
`      (S d)`
`      (T e))`
	if P then a
	else if Q then b also c
	else …
	else if S then d
	else e

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




```
TRI[x] = if x<-1 then 0
         else if x<0 then 1+x
         else if x<1 then 1-x
         else 0
```

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 (CAR A)) ≡ (SET (QUOTE L1) (CAR A))`
- Generally LISP evaluates in call-by-value fashion
  arguments evaluated left-to-right then function invoked
  e.g., `(PLUS (TIMES 2 3) 4)`
  1. multiply 2 and 3 to get 6
  2. invoke PLUS on 6 and 4 to yield 10
- COND is special form – args evaluated until TRUE found

```
EVAL[l] = if l[1] eq 'COND then EVALCOND(CDR l)
              .
              .
              .
EVALCOND[l] = if NULL(l) then NIL
              else if EVAL(l[1,1]) then EVLIST(CDR l[1])
              else EVALCOND[CDR l]

EVLIST[l] = if NULL(CDR(l)) then EVAL(l[1])
            else EVAL(l[1]) also EVLIST[CDR l]
```

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 AND A_2 AND A_3 … AND A_n

```
if A1 then
      if A2 then
            if A3 then
                  :
                  :
                  if An-1 then An
                  else nil
            else nil
      else nil
else nil
```

- Better is:

- Similarly for OR:
  A_1 OR A_2 OR A_3 … 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 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 AND A_2 AND A_3 … AND A_n

```
if A1 then
    if A2 then
        if A3 then
            :
            :
            if An–1 then An
            else nil
        else nil
    else nil
else nil
```

- Better is:

```
if not(A1) then nil
else if not(A2) then nil
else …
else if not(An–1) then nil
else An
```

- Similarly for OR:
  A_1 OR A_2 OR A_3 … 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 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 AND A_2 AND A_3 … AND A_n

```
if A1 then
    if A2 then
        if A3 then
            :
            :
            if An–1 then An
            else nil
        else nil
    else nil
else nil
```

- Better is:

```
if not(A1) then nil
else if not(A2) then nil
else …
else if not(An–1) then nil
else An
```

- Similarly for OR:
  A_1 OR A_2 OR A_3 … OR A_n

```
if A1 then T
else if A2 then T
else …
else if An–1 then T
else An
```

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 $(\text{s-expr})^n$ 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: ALT['(A B C D 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: 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: ALT['(A B C D E)] ⇒ (A C E)`

  `ALT['(A B)] ⇒ (A)`

  `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: 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: 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:
```
Ex: ALT['(A B C D E)] ⇒ (A C E)
    ALT['(A B)] ⇒ (A)
    ALT['(A) ] ⇒ (A)
    ALT['()] ⇒ ()
```
- ALT[x] =
```
ALT[x] =  if nx or ndx then x
          else ax . alt[ddx]
```

EXAMPLES OF ALT

- To see that this really works:

```
ALT['(A B)] = if n'(A B) ∨ nd'(A B) then'(A B)
                else a'(A B) . ALT[dd'(A B)]
            = if NIL ∨ nd'(A B) then '(A B)
                else a'(A B) . ALT[dd'(A B)]
            = if NIL then '(A B)
                else a'(A B) . ALT[dd'(A B)]
            = a'(A B) . ALT[dd'(A B)]
            = 'A . ALT[NIL]
            = 'A . [ if nNIL ∨ ndNIL then NIL
                      else aNIL . ALT[ddNIL]]
            = 'A . [ if T then NIL
                      else aNIL . ALT[ddNIL]]
            = 'A . [ NIL]
            = '(A)
```

- A briefer example:

```
ALT['(A B C D E)]  = 'A.ALT['(C D E)]
                   = 'A.['C.ALT['(E)]]
                   = 'A.['C.(E)]
                   = '(A C E)
```

- Observations:
  1. rules for evaluating Boolean conditions are important since if evaluation of OR continued after finding one NIL we would then evaluate dNIL which is undefined
  2. we can build a *list* result by returning from recursion

LAST ATOM OF A LIST

- Construct a function to return the last atom of a list
  base case:

  induction case:

  `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:

  `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] =`

- 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: `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:

```
SUBST[x,y,z] =
```

SUBSTITUTE FUNCTION

- Substitute s-expression *x* for all occurrences of atom *y* in the s-expression *z*
  for example: `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:
  ```
  if atz then
      if z eq y then x
      else z
  ```

  inductive case:

  ```
  SUBST[x,y,z] =
  ```

SUBSTITUTE FUNCTION

- Substitute s-expression *x* for all occurrences of atom *y* in the s-expression *z*
  for example: `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:
```
if atz then
     if z eq y then x
     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

```
SUBST[x,y,z] =
```

SUBSTITUTE FUNCTION

- Substitute s-expression *x* for all occurrences of atom *y* in the s-expression *z*
  for example: `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:
```
if atz then
    if z eq y then x
    else z
```

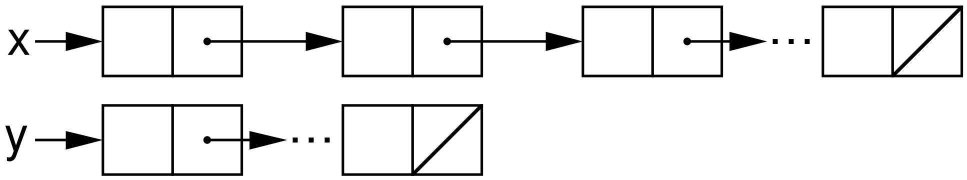
inductive case: `subst[x,y,az], subst[x,y,dz]`

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

```
SUBST[x,y,z] =

  if atz then
     if z eq y then x
     else z
  else subst[x,y,az].subst[x,y,dz]
```

APPEND FUNCTION

- Takes two lists as arguments and concatenates them



- 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*

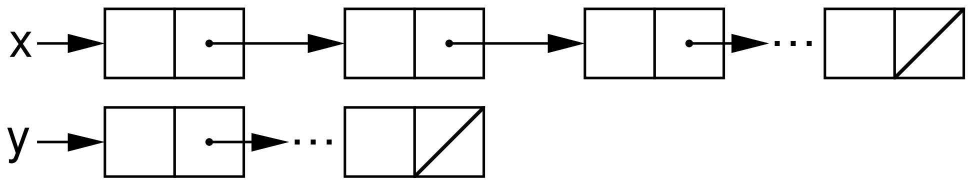
  base case:

  induction:

  `APPEND[x,y] =`

APPEND FUNCTION

- Takes two lists as arguments and concatenates them



- 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*

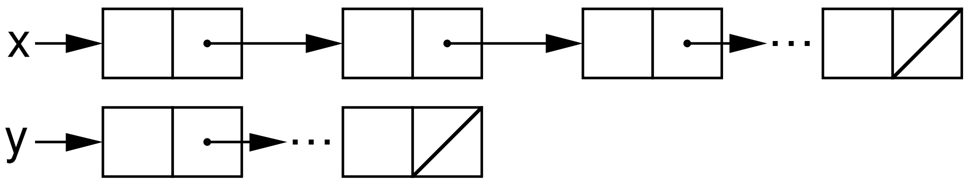
base case: `if nx then y`

induction:

`APPEND[x,y] =`

APPEND FUNCTION

- Takes two lists as arguments and concatenates them



- 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*

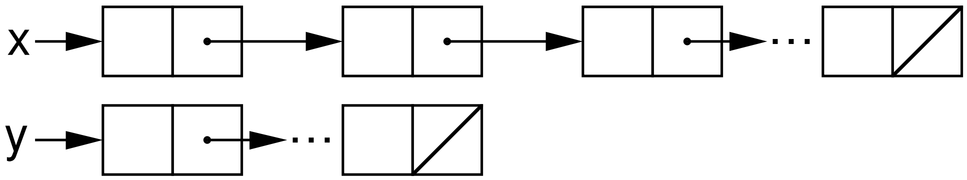
base case: `if nx then y`

induction: `[ax].append[dx,y]`

```
APPEND[x,y] =
```

APPEND FUNCTION

- Takes two lists as arguments and concatenates them



- 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: `if nx then y`

induction: `[ax].append[dx,y]`

```
APPEND[x,y] = if nx then y
              else [ax].APPEND[dx,y]
```

REVERSE FUNCTION

- Use auxiliary function to simplify task

  `REVERSE[x]=`

  `REVERSE1[x,y]=`

- Variable *y* serves as place-holder to contain result
- Also possible to define using only one function and APPEND

  `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

  `REVERSE[x]= REVERSE1[x,nil]`

  `REVERSE1[x,y]=`

- Variable *y* serves as place-holder to contain result
- Also possible to define using only one function and APPEND

  `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

```
REVERSE[x]=REVERSE1[x,nil]

REVERSE1[x,y]=if nx then y
              else REVERSE1[dx,<ax>.y]
```

- Variable *y* serves as place-holder to contain result
- Also possible to define using only one function and APPEND

```
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

```
REVERSE[x]=REVERSE1[x,nil]

REVERSE1[x,y]=if nx then y
              else REVERSE1[dx,<ax>.y]
```

- Variable *y* serves as place-holder to contain result
- Also possible to define using only one function and APPEND

```
REVERSE[x]=if nx then nil
           else 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:

```
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 FLAT[*x*,*y*] where *y* accumulates the atoms
- Result list will contain atoms encountered from left to right
- Whenever an atom is encountered we add it to *y*

base case: `if atx then x.y`

induction:

```
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 FLAT[*x*,*y*] where *y* accumulates the atoms
- Result list will contain atoms encountered from left to right
- Whenever an atom is encountered we add it to *y*

base case: `if atx then x.y`

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

```
FLATTEN[x]=

FLAT[x,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 FLAT[*x*,*y*] where *y* accumulates the atoms
- Result list will contain atoms encountered from left to right
- Whenever an atom is encountered we add it to *y*

base case: `if atx then x.y`

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

`flat[ax,flat[dx,y]]`

```
FLATTEN[x]=FLAT[x,nil]

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

- This technique is useful for applying an arbitrary function to both the head and tail of a given s-expression
- Could also be constructed without using auxiliary function:

`FLATTEN[x]=`

FLATTEN FUNCTION

- Make flat list of all atoms in a given s-expression
- Use auxiliary function FLAT[*x*,*y*] where *y* accumulates the atoms
- Result list will contain atoms encountered from left to right
- Whenever an atom is encountered we add it to *y*

  base case: `if atx then x.y`

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

```
FLATTEN[x]=FLAT[x,nil]

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

- This technique is useful for applying an arbitrary function to both the head and tail of a given s-expression
- Could also be constructed without using auxiliary function:

```
FLATTEN[x]=if atx then x
           else FLATTEN[ax]*FLATTEN[dx]
```

TRADEOFF: EXTRA ARG VS EFFICIENCY

- Common when creating LISP functions
  Ex: factorial:
  `FACT[x]=`

  with the addition of a second argument:
  `FACT[x]=`

  `FACT1[x,y]=`

- The general case
  1. note similarity of transformation in REVERSE and FACT
  2. consider these schemas
     f(x) = if p(x) then a else $b \oplus f(g(x))$ $\Rightarrow$ h(x,y) = if p(x) then $y \oplus a$ else $(h(g(x), y \oplus b)$
     f(x) = if p(x) then a else $f(g(x)) \oplus b$ $\Rightarrow$ h(x,y) = if p(x) then $a \oplus y$ else $(h(g(x), b \oplus y)$
     with $f(x) \equiv h(x, id\oplus)$ and $id\oplus$ 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 <CAR x>
     g is CDR
     $\oplus$ is APPEND
     $b \oplus y \equiv$ <CAR x> APPEND y $\equiv$ <u>a</u> x CONS y (since <u>a</u> x 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:

```
FACT[x]= if x eq 1 then 1
         else x•FACT[x–1]
```

  with the addition of a second argument:

```
FACT[x]=


FACT1[x,y]=
```

- The general case
  1. note similarity of transformation in REVERSE and FACT
  2. consider these schemas

     f(x) = if p(x) then a ⇒ h(x,y) = if p(x) then y⊕a
     else b⊕f(g(x)) else (h(g(x),y⊕b)

     f(x) = if p(x) then a ⇒ h(x,y) = if p(x) then a⊕y
     else f(g(x))⊕b else (h(g(x),b⊕y)

     with f(x) ≡ h(x,id⊕) and id⊕ is identity element of ⊕ op
  3. when are these transformations valid?
     - ⊕ must be

  4. Ex: REVERSE: p is null
     a is NIL
     b is <CAR x>
     g is CDR
     ⊕ is APPEND

     b⊕y ≡ <CAR x> APPEND y ≡ <u>a</u> x CONS y (since <u>a</u> x is atom)
  5. Ex: FACTORIAL: p is x eq 1
     a is 1
     b is x
     g is x–1
     ⊕ is multiplication

TRADEOFF: EXTRA ARG VS EFFICIENCY

- Common when creating LISP functions
  Ex: factorial:

```
FACT[x]=if x eq 1 then 1
        else x•FACT[x−1]
```

  with the addition of a second argument:

```
FACT[x]=FACT1[x,1]

FACT1[x,y]=if x eq 1 then y
           else FACT1[x−1,x•y]
```

- The general case
  1. note similarity of transformation in REVERSE and FACT
  2. consider these schemas

     f(x) = if p(x) then a else b⊕f(g(x)) ⇒ h(x,y) = if p(x) then y⊕a else (h(g(x),y⊕b)

     f(x) = if p(x) then a else f(g(x))⊕b ⇒ h(x,y) = if p(x) then a⊕y else (h(g(x),b⊕y)

     with f(x) ≡ h(x,id⊕) and id⊕ is identity element of ⊕ op
  3. when are these transformations valid?
     - ⊕ must be
  4. Ex: REVERSE: p is null
     a is NIL
     b is <CAR x>
     g is CDR
     ⊕ is APPEND

     b⊕y ≡ <CAR x> APPEND y ≡ <u>a</u> x CONS y (since <u>a</u> x is atom)
  5. Ex: FACTORIAL: p is x eq 1
     a is 1
     b is x
     g is x–1
     ⊕ is multiplication

TRADEOFF: EXTRA ARG VS EFFICIENCY

- Common when creating LISP functions

  Ex: factorial:

```
FACT[x]=if x eq 1 then 1
        else x•FACT[x-1]
```

  with the addition of a second argument:

```
FACT[x]=FACT1[x,1]


FACT1[x,y]=if x eq 1 then y
           else FACT1[x-1,x•y]
```

- The general case

  1. note similarity of transformation in REVERSE and FACT
  2. consider these schemas

     f(x) = if p(x) then a else b⊕f(g(x)) ⇒ h(x,y) = if p(x) then y⊕a else (h(g(x),y⊕b)

     f(x) = if p(x) then a else f(g(x))⊕b ⇒ h(x,y) = if p(x) then a⊕y else (h(g(x),b⊕y)

     with f(x) ≡ h(x,id⊕) and id⊕ is identity element of ⊕ op
  3. when are these transformations valid?
     - ⊕ must be associative
  4. Ex: REVERSE: p is null
     a is NIL
     b is <CAR x>
     g is CDR
     ⊕ is APPEND

     b⊕y ≡ <CAR x> APPEND y ≡ <u>a</u> x CONS y (since <u>a</u> x is atom)
  5. Ex: FACTORIAL: p is x eq 1
     a is 1
     b is x
     g is x–1
     ⊕ is multiplication

GREATEST COMMON DENOMINATOR

- highest number that divides both m and n
- recursively:

$$
\text{GCD}(m,n) = \begin{array}{l} \text{if } m>n \text{ then GCD}(n,m) \\ \text{else if } m=0 \text{ then } n \\ \text{else GCD}(n \text{ MOD } m, m) \end{array}
$$

where:

$$
n \text{ MOD } m = \begin{array}{l} \text{if } n<m \text{ then } n \\ \text{else } (n-m) \text{ MOD } m \end{array}
$$

i.e., subtract until number between 0 and $\text{MIN}(m,n)-1$

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 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:

  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 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,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]=`

- 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•*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

```
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•*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

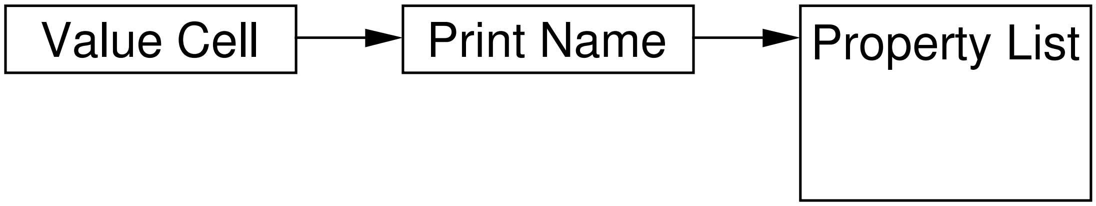
```
SUBST2[x,y,z]=
   if atz then
       if z eq y then x
       else z
   else LAMBDA (head,tail);
       if head eq az and tail eq dz then z
       else head.tail;
       (SUBST2[x,y,az],SUBST2[x,y,dz])
```

EXAMPLE OF THE USE OF INTERNAL LAMBDA

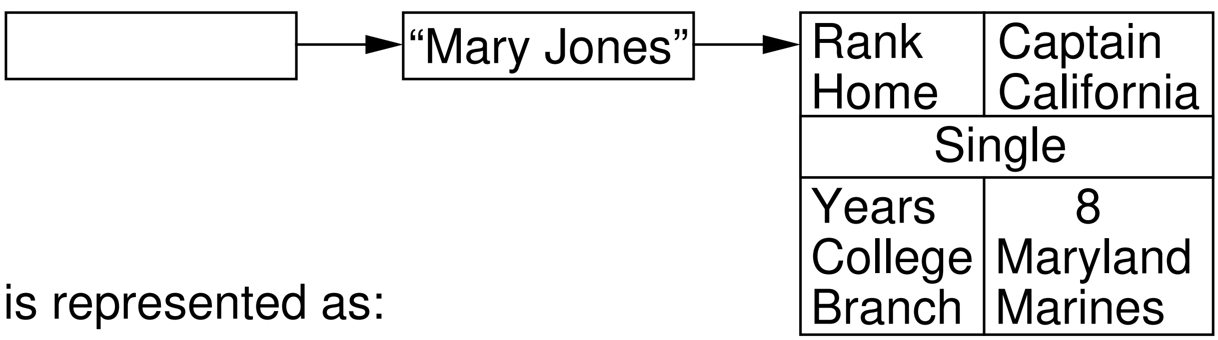
```
sexpr procedure substz(x,y,z);   // Subst x for y in z
begin                            // Copy made only if a
   if atom(z) then               // subst instance found
      if eq(y,z) then return(x)
      else return(z)
   else
    begin
      head ← substz(x,y,car(z));
      tail ← substz(x,y,cdr(z));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

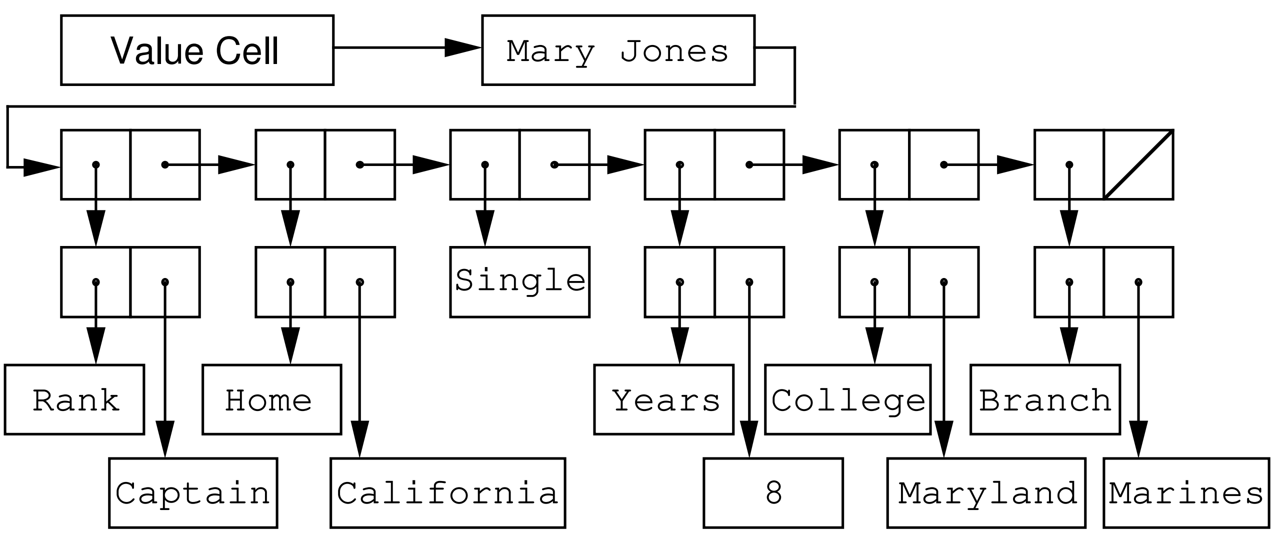
- Wisconsin LISP represents an atom:



- *Value cell* contains value bound to the atom
  i.e. `(SETQ A (QUOTE (JOHN MARY)))`
  means that the value of `A` is `(JOHN MARY)`
- *Print name* is atom's name as a sequence of characters
- *Property list*
  1. data structure storing two levels of information on atom
  2. like association list with addition of *flag atoms*
  3. Ex:



is represented as:



PROPERTY LIST FUNCTIONS

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

1. `(PUT x y z)` put property `y` on atom `x`'s property list with property value `z`, e.g.,
   `(PUT (QUOTE AL)(QUOTE HAIR)(QUOTE RED))`
2. `(GET x y)` fetch property value associated with property `y` on atom `x`'s property list, e.g.,
   `(GET (QUOTE CHARLES)(QUOTE ADDRESS))`
   - just like ASSOC
3. `(REMPROP x y)` removes property `y` and its associated property value from atom `x`'s list, e.g.,
   `(REMPROP (QUOTE ANGOLA)(QUOTE COLONY))`
4. `(FLAG x y)` places flag `y` on atom `x`'s prop list, e.g.,
   `(FLAG (QUOTE MARY)(QUOTE MARRIED))`
5. `(IFFLAG x y)` returns TRUE if atom `x` has flag `y`, e.g.,
   `(IFFLAG (QUOTE JOE)(QUOTE CITIZEN))`
6. `(UNFLAG x y)` removes flag `y` from atom `x`'s list, e.g.,
   `(UNFLAG (QUOTE CASE)(QUOTE RECESSED))`