6.945 Adventures in Advanced Symbolic Programming

Spring 2009

The Future Is Parallel: What's a Programmer to Do?

Breaking Sequential Habits of Thought

Guy Steele

Sun Fellow

Sun Microsystems Laboratories

April 2009

With Multicore, a Profound Shift

- Parallelism is here, now, and in our faces
  - Academics have been studying it for 50 years
  - Serious commercial offerings for 25 years
  - But now it's in desktops and laptops
- Specialized expertise for science codes and databases and networking
- But soon general practitioners must go parallel
- An opportunity to make parallelism easier for everyone

This Talk Is about Performance

The bag of programming tricks

that has served us so well

for the last 50 years

is

the wrong way to think

going forward and

must be thrown out.

Why?

- Good sequential code minimizes total number of operations.
  - Clever tricks to reuse previously computed results.
  - Good parallel code often performs redundant operations to reduce communication.
- Good sequential algorithms minimize space usage.
  - Clever tricks to reuse storage.
  - Good parallel code often requires extra space to permit temporal decoupling.
- Sequential idioms stress linear problem decomposition.
  - Process one thing at a time and accumulate results.
  - Good parallel code usually requires multiway problem decomposition and multiway aggregation of results.

Let's Add a Bunch of Numbers

```
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

Let's Add a Bunch of Numbers

```
SUM = 0                 // Oops!

DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

This is already bad!

Clever compilers have to undo this.

What Does a Mathematician Say?

$$\sum_{i=1}^{1000000} x_i \quad \text{or maybe just} \quad \sum x$$

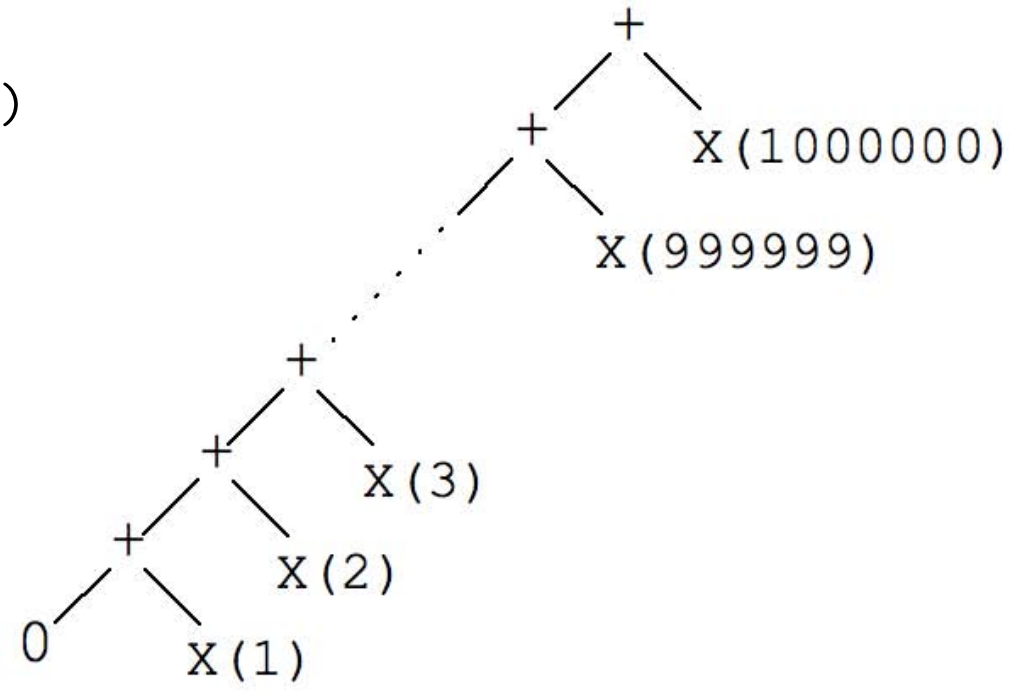
Compare Fortran 90 `SUM(X)`.

What, not how.

No commitment yet as to strategy. This is good.

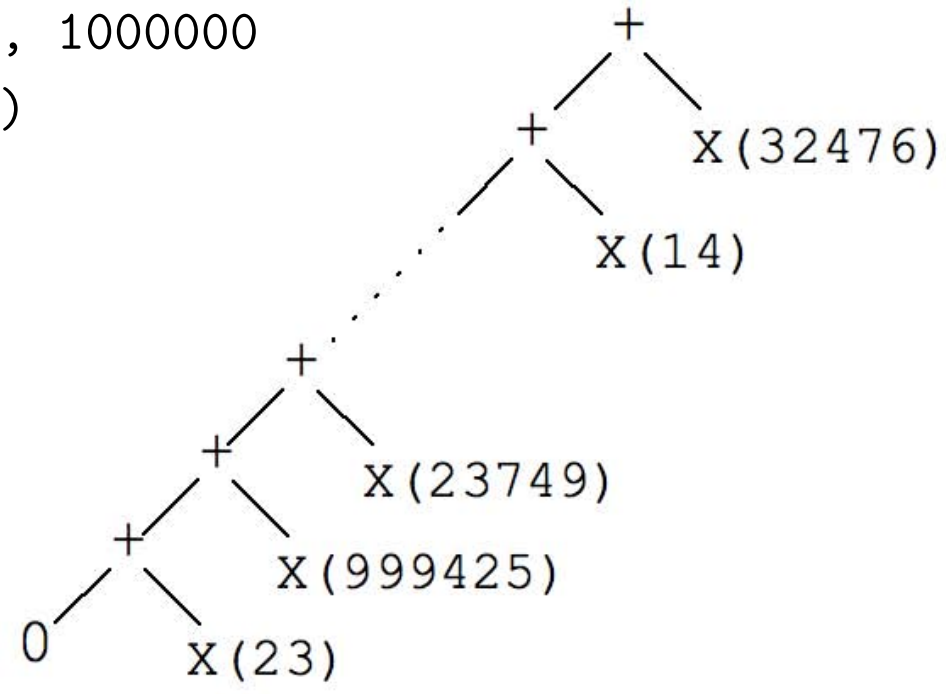
Sequential Computation Tree

```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```



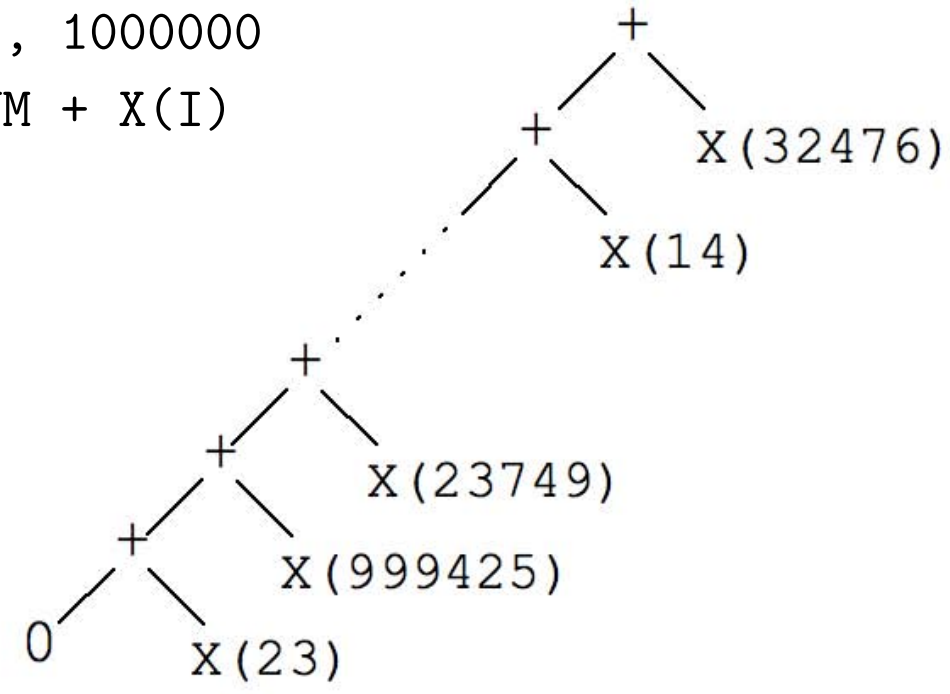
Atomic Update Computation Tree (a)

```
SUM = 0
PARALLEL DO I = 1, 1000000
   SUM = SUM + X(I)
END DO
```



Atomic Update Computation Tree (b)

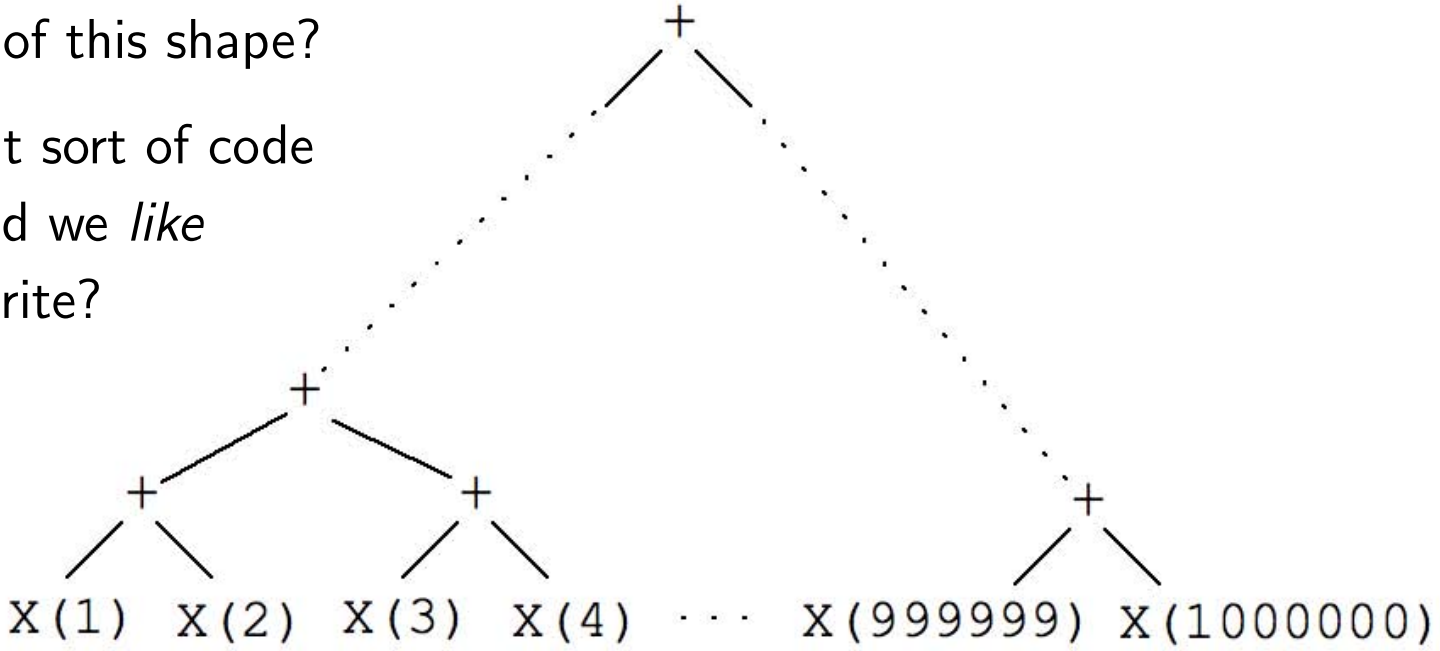
```
SUM = 0
PARALLEL DO I = 1, 1000000
  ATOMIC SUM = SUM + X(I)
END DO
```



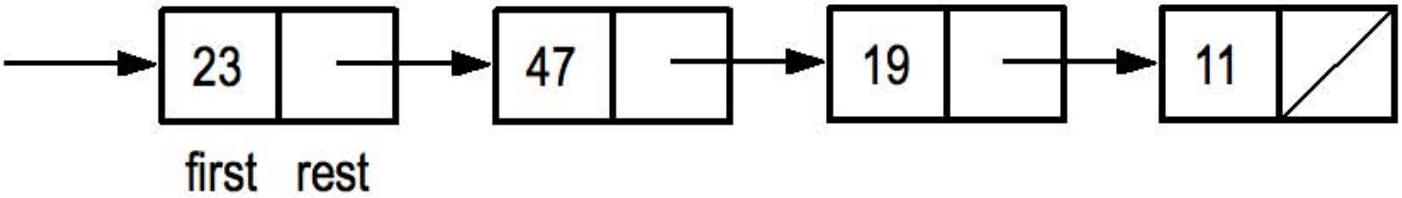
Parallel Computation Tree

What sort of code
should we write
to get a computation
tree of this shape?

What sort of code
would we *like*
to write?



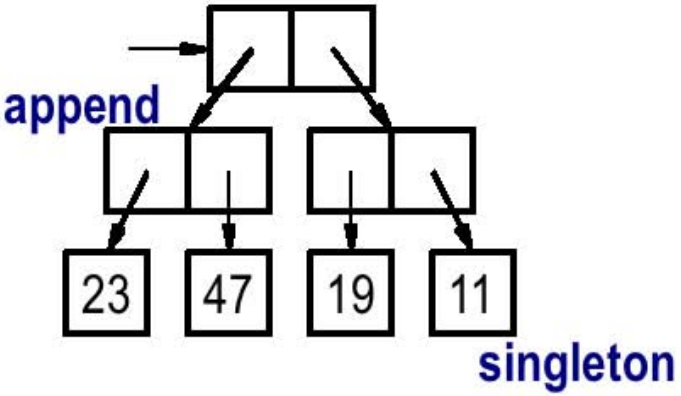
Finding the Length of a LISP List



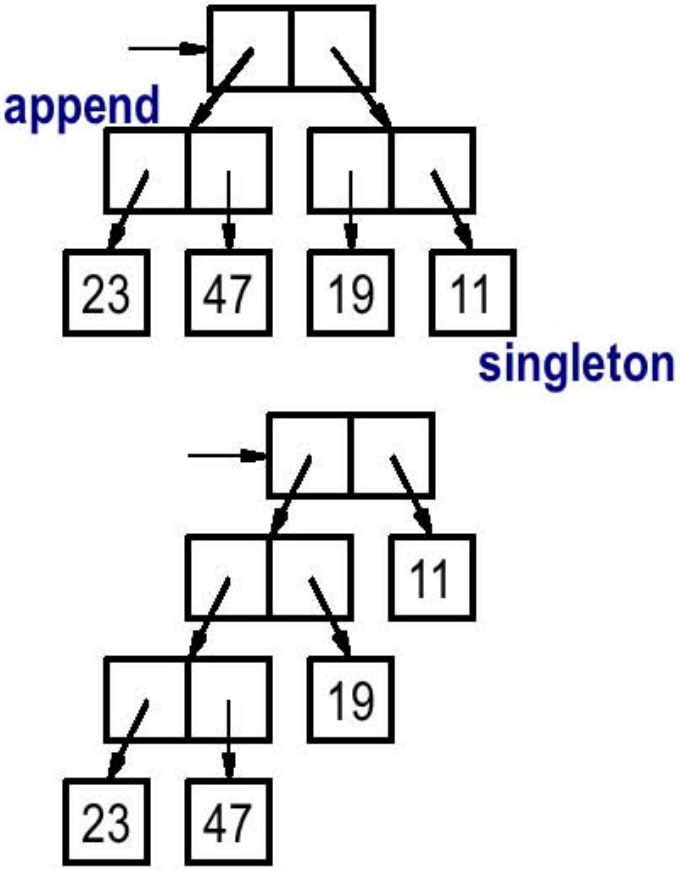
Recursive:

```
(define length (list)
  (cond ((null list) 0)
        (else (+ 1 (length (rest list))))))
```

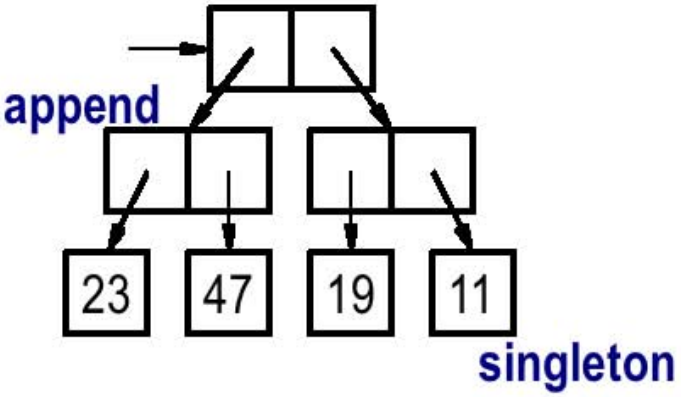
Append Lists (1)

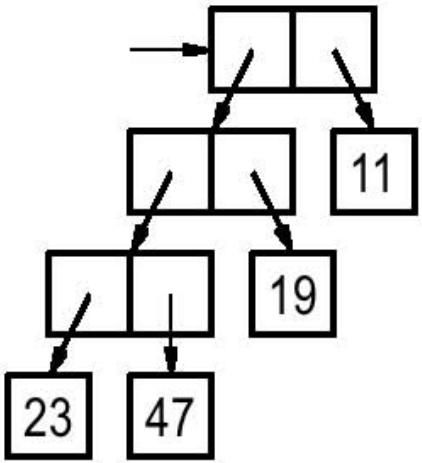


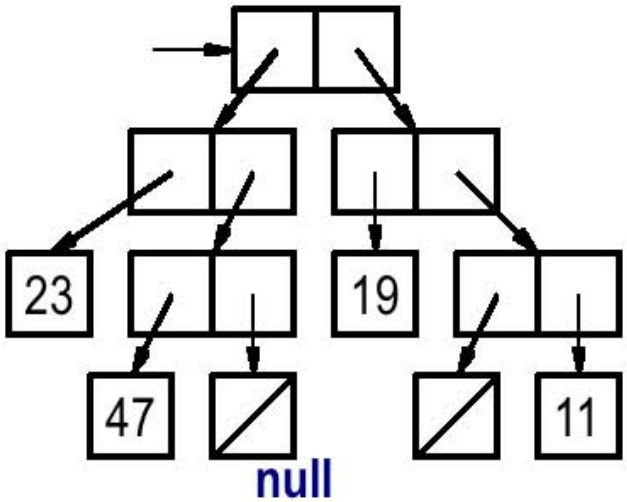
Append Lists (2)



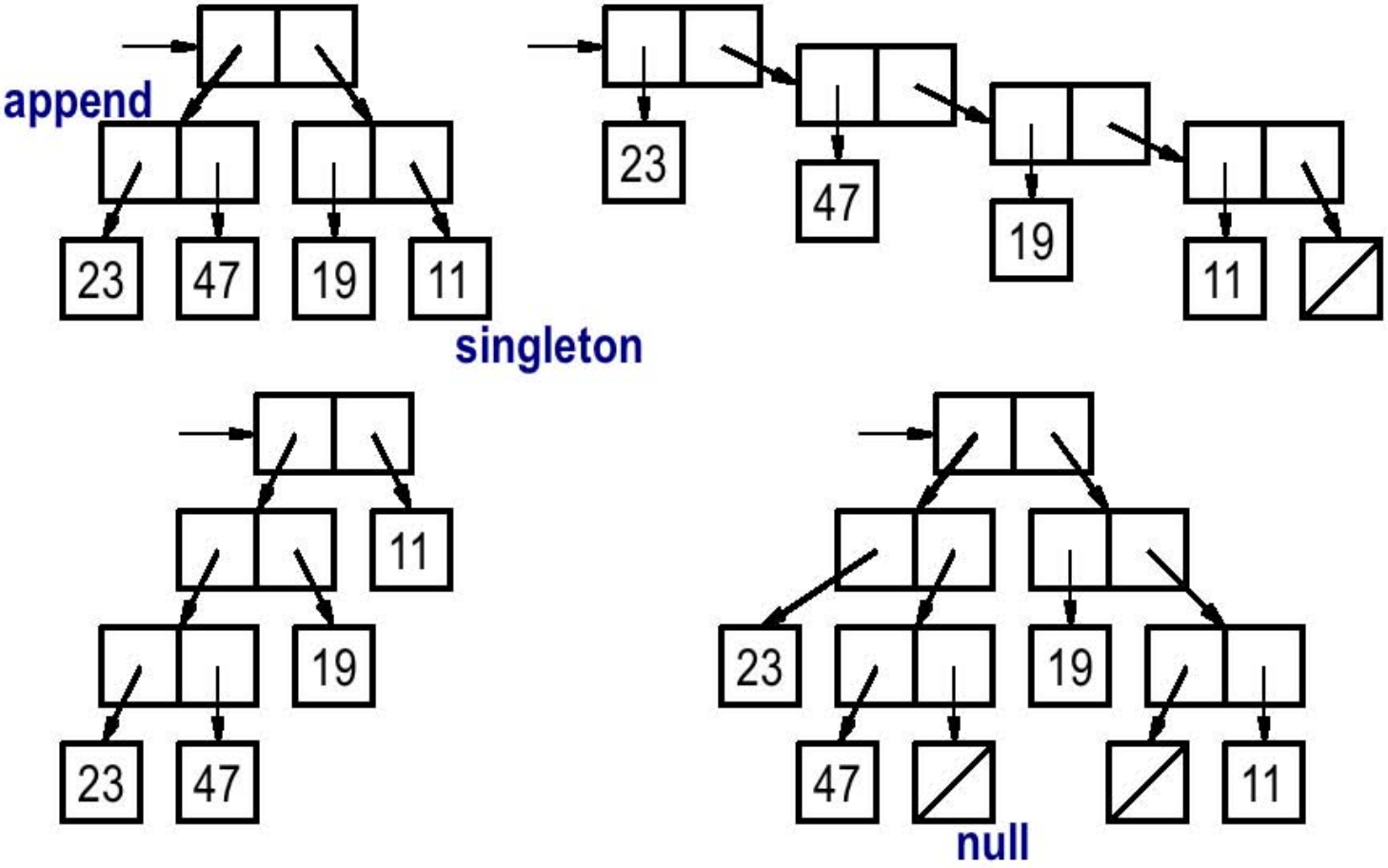
Append Lists (3)







Append Lists (4)



Defining Lists Using `car cdr cons`

```
(define (first x)
  (cond ((null? x) '())
        (else (car x))))

(define (rest x)
  (cond ((null? x) '())
        (else (cdr x))))

(define (append xs ys)
  (cond ((null? xs) ys)
        (else (cons (car xs) (append (cdr xs) ys)))))

(define (addleft a xs) (cons a xs))

(define (addright xs a)
  (cond ((null? xs) (list a))
        (else (cons (car xs) (addright (cdr xs) a)))))
```

Defining Lists Using `car cdr cons`

```
(define (first x)                          ;Constant time
  (cond ((null? x) '())
        (else (car x))))

(define (rest x)                           ;Constant time
  (cond ((null? x) '())
        (else (cdr x))))

(define (append xs ys)                     ;Linear in (length xs)
  (cond ((null? xs) ys)
        (else (cons (car xs) (append (cdr xs) ys)))))

(define (addleft a xs) (cons a xs))        ;Constant time

(define (addright xs a)                    ;Linear in (length xs)
  (cond ((null? xs) (list a))
        (else (cons (car xs) (addright (cdr xs) a)))))
```

map reduce mapreduce

```
(map (λ (x) (* x x)) '(1 2 3))  =>  (1 4 9)

(reduce + 0 '(1 4 9))  =>  14

(mapreduce (λ (x) (* x x)) + 0 '(1 2 3))  =>  14
```

```
(define (map f xs)                          ;Linear in (length xs)
  (cond ((null? xs) '())
        (else (cons (f (car xs)) (map f (cdr xs))))))

(define (reduce g id xs)                    ;Linear in (length xs)
  (cond ((null? xs) id)
        (else (g (car xs) (reduce g id (cdr xs))))))

(define (mapreduce f g id xs)               ;Linear in (length xs)
  (cond ((null? xs) id)
        (else (g (f (car xs)) (mapreduce f g id (cdr xs))))))
```

length filter

```
(define (length xs)                          ;Linear in (length xs)
  (mapreduce (λ (q) 1) + 0 xs))

(define (filter p xs)                        ;Linear in (length xs)
  (cond ((null? xs) '())
        ((p (car xs)) (cons p (filter p (cdr xs))))
        (else (filter p (cdr x)))))

(define (filter p xs)                        ;Linear in (length xs)??
  (apply append
         (map (λ (x) (if (p x) (list x) '())) xs)))

(define (filter p xs)                        ;Linear in (length xs)!!
  (mapreduce (λ (x) (if (p x) (list x) '()))
             append '() xs))
```

reverse

```
(define (reverse xs)                    ;QUADRATIC in (length xs)
  (cond ((null? xs) '())
        (else (addright (reverse (cdr xs)) (car xs)))))

(define (revappend xs ys)               ;Linear in (length xs)
  (cond ((null? xs) ys)
        (else (revappend (cdr xs) (cons (car xs) ys)))))

(define (reverse xs)                    ;Linear in (length xs)
  (revappend xs '()))
```

Linear versus Multiway Decomposition

- Linearly linked lists are inherently sequential.
  - Compare Peano arithmetic: $5 = ((((0+1)+1)+1)+1)+1$
  - Binary arithmetic is much more efficient than unary!
- We need a *multiway decomposition* paradigm:

```
length [ ] = 0
length [a] = 1
length (a++b) = (length a) + (length b)
```

This is just a summation problem: adding up a bunch of 1's!
(More generally: a bunch of 0's and 1's.)

Defining Lists Using item list split conc (1)

```
(define (first xs)                          ;Depth of left path
  (cond ((null? xs) '())
        ((singleton? xs) (item xs))
        (else (split xs (λ (ys zs) (first ys))))))

(define (rest xs)                           ;Depth of left path
  (cond ((null? xs) '())
        ((singleton? xs) '())
        (else (split xs (λ (ys zs) (append (rest ys) zs))))))

(define (append xs ys)                      ;Constant time
  (cond ((null? xs) ys)
        ((null? ys) xs)
        (else (conc xs ys))))
```

Defining Lists Using item list split conc (2)

```
(define (first xs)                          ;Depth of left path
  (cond ((null? xs) '())
        ((singleton? xs) (item xs))
        (else (split xs (λ (ys zs) (first ys))))))

(define (rest xs)                           ;Depth of left path
  (cond ((null? xs) '())
        ((singleton? xs) '())
        (else (split xs (λ (ys zs) (append (rest ys) zs))))))

(define (append xs ys)                      ;???
  (cond ((null? xs) ys)
        ((null? ys) xs)
        (else (rebalance (conc xs ys)))))
```

Defining Lists Using item list split conc (3)

```
(define (addleft a xs)
  (cond ((null? xs) (list a))
        ((singleton? xs) (append (list a) xs))
        (else (split xs (λ (ys zs) (append (addleft a ys) zs))))))

(define (addright xs a)
  (cond ((null? xs) (list a))
        ((singleton? xs) (append xs (list a)))
        (else (split xs (λ (ys zs) (append ys (addright a zs)))))))

(define (addleft a xs) (append (list a) xs))

(define (addright xs a) (append xs (list a)))
```

Parallel map reduce mapreduce

```
(define (mapreduce f g id xs)          ;Logarithmic in (length xs)??
  (cond ((null? xs) id)
        ((singleton? xs) (f (item xs)))
        (else (split xs (λ (ys zs)
                (g (mapreduce f g id ys)
                   (mapreduce f g id zs)))))))

(define (map f xs)
  (mapreduce (λ (x) (list (f x))) append '() xs))

(define (reduce g id xs)
  (mapreduce (λ (x) x) g id xs))
```

length filter reverse

```
(define (length xs)                          ;Logarithmic in (length xs)??
  (mapreduce (λ (q) 1) + 0 xs))

(define (filter p xs)                        ;Logarithmic in (length xs)??
  (mapreduce (λ (x) (if (p x) (list x) '()))
             append '() xs))

(define (reverse xs)                         ;Logarithmic in (length xs)??
  (mapreduce list (λ (yx zs) (append zs ys)) '() xs))
```

Exercise: Write Mergesort and Quicksort in This Binary-split Style

- Mergesort: structural induction on input
  - Cheaply split input in half
  - Recursively sort the two halves
  - Carefully merge the two sorted sublists (tricky)
- Quicksort: structural induction on output
  - Carefully split input into lower and upper halves (tricky)
  - Recursively sort the two halves
  - Cheaply append the two sorted sublists

A Modest Example: Filter (1)

$$
\begin{array}{l}
\mathit{sequentialFilter}[\![E]\!](a\colon \mathrm{List}[\![E]\!], p\colon E \to \mathrm{Boolean})\colon \mathrm{List}[\![E]\!] = \texttt{do} \\
\quad \mathit{result}\colon \mathrm{List}[\![E]\!] := \langle\,\rangle \\
\quad \texttt{for}\ \ k \leftarrow \mathit{seq}(0\,\texttt{\#}\,a.\mathit{size}())\ \texttt{do} \\
\quad\quad \texttt{if}\ \ p(a_k)\ \texttt{then}\ \ \mathit{result} := \mathit{result}.\mathit{addRight}(a_k)\ \texttt{end} \\
\quad \texttt{end} \\
\quad \mathit{result} \\
\texttt{end}
\end{array}
$$

Example of use:

$$\mathit{odd}(x\colon \mathbb{Z}) = ((x\ \texttt{MOD}\ 2) \neq 0)$$

$$\mathit{sequentialFilter}(\langle\, 1, 4, 7, 2, 5 \,\rangle, \mathit{odd})$$

So what language is this? **Fortress**.

A Modest Example: Filter (2)

$$
\begin{array}{l}
\mathit{recursiveFilter}[\![E]\!](a\colon \mathrm{List}[\![E]\!], p\colon E \to \mathrm{Boolean})\colon \mathrm{List}[\![E]\!] = \\
\quad \texttt{if } a.\mathit{isEmpty}() \texttt{ then } \langle\,\rangle \\
\quad \texttt{else} \\
\qquad (\mathit{first}, \mathit{rest}) = a.\mathit{extractLeft}().\mathit{get}() \\
\qquad \mathit{rest}' = \mathit{recursiveFilter}(\mathit{rest}, p) \\
\qquad \texttt{if } p(\mathit{first}) \texttt{ then } \mathit{rest}'.\mathit{addLeft}(\mathit{first}) \texttt{ else } \mathit{rest}' \texttt{ end} \\
\quad \texttt{end}
\end{array}
$$

A Modest Example: Filter (3)

$$
\begin{array}{l}
parallelFilter[\![E]\!](a\colon \mathrm{List}[\![E]\!], p\colon E \to \mathrm{Boolean})\colon \mathrm{List}[\![E]\!] = \\
\quad \texttt{if } |a| = 0 \texttt{ then } \langle\,\rangle \\
\quad \texttt{elif } |a| = 1 \texttt{ then} \\
\qquad (first, \_) = a.extractLeft().get() \\
\qquad \texttt{if } p(first) \texttt{ then } a \texttt{ else } \langle\,\rangle \texttt{ end} \\
\quad \texttt{else} \\
\qquad (x, y) = a.split() \\
\qquad parallelFilter(x, p) \parallel parallelFilter(y, p) \\
\quad \texttt{end}
\end{array}
$$

A Modest Example: Filter (4)

$$reductionFilter[\![E]\!](a\colon \mathrm{List}[\![E]\!], p\colon E \to \mathrm{Boolean})\colon \mathrm{List}[\![E]\!] = \Vert_{x \leftarrow a}\ (\texttt{if}\ \ p(x)\ \texttt{then}\ \langle\, x \,\rangle\ \texttt{else}\ \langle\,\rangle\ \texttt{end})$$

$$\langle\, x \mid x \leftarrow a, p(x) \,\rangle$$

Oh, yes: $\sum_{i \leftarrow 1:1000000} x_{\mathrm{i}}$

Splitting a String into Words (1)

- Given: a string
- Result: List of strings, the words separated by spaces
  - Words must be nonempty
  - Words may be separated by more than one space
  - String may or may not begin (or end) with spaces

Splitting a String into Words (2)

- Tests:

$$\mathit{println}\ \mathit{words}(\text{“}\texttt{This is a sample}\text{”})$$
$$\mathit{println}\ \mathit{words}(\text{“}\ \ \texttt{Here is another sample}\ \ \text{”})$$
$$\mathit{println}\ \mathit{words}(\text{“}\texttt{JustOneWord}\text{”})$$
$$\mathit{println}\ \mathit{words}(\text{“}\ \ \text{”})$$
$$\mathit{println}\ \mathit{words}(\text{“”})$$

- Expected output:

$\langle\, \texttt{This}, \texttt{is}, \texttt{a}, \texttt{sample} \,\rangle$

$\langle\, \texttt{Here}, \texttt{is}, \texttt{another}, \texttt{sample} \,\rangle$

$\langle\, \texttt{JustOneWord} \,\rangle$

$\langle\, \rangle$

$\langle\, \rangle$

Splitting a String into Words (3)

```
words(s: String) = do
  result: List⟦String⟧ := ⟨ ⟩
  word: String := ""
  for k ← seq(0 # length(s)) do
    char = substring(s, k, k + 1)
    if (char = " ") then
      if (word ≠ "") then result := result || ⟨ word ⟩ end
      word := ""
    else
      word := word || char
    end
  end
  if (word ≠ "") then result := result || ⟨ word ⟩ end
  result
end
```

Splitting a String into Words (4a)

```
Here is a sesquipedalian string of words

Here is a sesquipeda|lian string of words

Here is a |sesquipeda|lian strin|g of words
```

Splitting a String into Words (4b)

Chunk("sesquipeda")

Splitting a String into Words (4c)

Segment(“g”, ⟨“of”⟩, “words”)

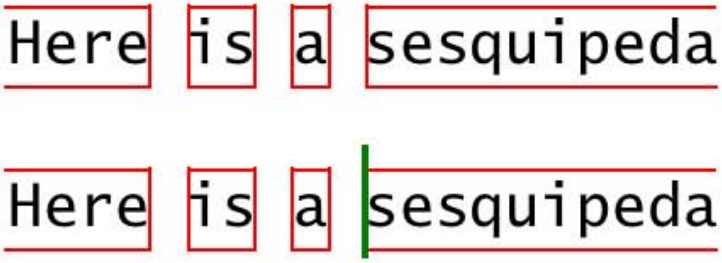
Splitting a String into Words (4d)

Segment(“Here”, ⟨“is”, “a”⟩, “”)

Splitting a String into Words (4e)

Segment("lian", ⟨ ⟩, "strin")

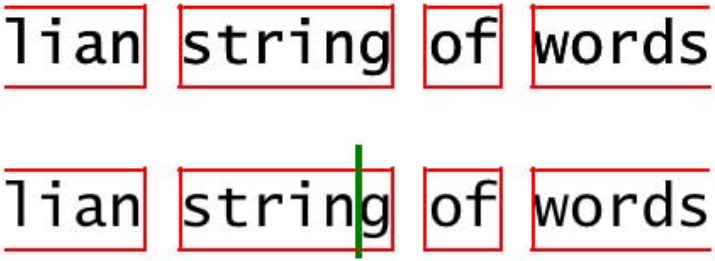
Splitting a String into Words (4f)



Segment("Here", ⟨"is", "a"⟩, "") Chunk("sesquipeda")

Segment("Here", ⟨"is", "a"⟩, "sesquipeda")

Splitting a String into Words (4g)



Segment("lian", ⟨ ⟩, "strin")

Segment("g", ⟨"of"⟩, "words")

Segment("lian", ⟨"string", "of"⟩, "words")

Splitting a String into Words (4h)

Segment(
“Here”,
⟨“is”, “a”, “sesquipedialian”, “string”, “of”⟩,
“words”)

Splitting a String into Words (5)

$$maybeWord(s\colon \mathrm{String})\colon \mathrm{List}[\![\mathrm{String}]\!] =$$
$$\texttt{if } s = \text{“”} \texttt{ then } \langle\,\rangle \texttt{ else } \langle\, s\,\rangle \texttt{ end}$$

```
trait WordState
        extends { Associative[[WordState, ⊕]] }
        comprises { Chunk, Segment }
    opr ⊕(self, other: WordState): WordState
end
```

Splitting a String into Words (6)

```
object Chunk(s: String) extends WordState
  opr ⊕(self, other: Chunk): WordState =
    Chunk(s ‖ other.s)
  opr ⊕(self, other: Segment): WordState =
    Segment(s ‖ other.l, other.A, other.r)
end
```

Splitting a String into Words (7)

```
object Segment(l: String, A: List⟦String⟧, r: String)
    extends WordState
  opr ⊕(self, other: Chunk): WordState =
    Segment(l, A, r ‖ other.s)
  opr ⊕(self, other: Segment): WordState =
    Segment(l, A ‖ maybeWord(r ‖ other.l) ‖ other.A, other.r)
end
```

Splitting a String into Words (8)

$$
\begin{array}{l}
\mathit{processChar}(c\colon \mathrm{String})\colon \mathrm{WordState} = \\
\quad \texttt{if } (c = \text{“ ”}) \texttt{ then } \mathrm{Segment}(\text{“”}, \langle\,\rangle, \text{“”}) \\
\quad \texttt{else } \mathrm{Chunk}(c) \\
\quad \texttt{end} \\
\\
\mathit{words}(s\colon \mathrm{String}) = \texttt{do} \\
\quad g = \bigoplus_{k \leftarrow 0 \# \mathit{length}(s)} \mathit{processChar}(\mathit{substring}(s, k, k+1)) \\
\quad \texttt{typecase } g \texttt{ of} \\
\qquad \mathrm{Chunk} \Rightarrow \mathit{maybeWord}(g.s) \\
\qquad \mathrm{Segment} \Rightarrow \mathit{maybeWord}(g.l) \parallel g.A \parallel \mathit{maybeWord}(g.r) \\
\quad \texttt{end} \\
\texttt{end}
\end{array}
$$

Splitting a String into Words (9)

```
(* The mechanics of BIG OPLUS *)
opr BIG ⊕ [[T]](g : (Reduction[[WordState]],
                     T → WordState)
                  → WordState): WordState =
  g(GlomReduction, identity[[WordState]])
object GlomReduction extends Reduction[[WordState]]
  getter toString() = "GlomReduction"
  empty(): WordState = Chunk("")
  join(a: WordState, b: WordState): WordState = a ⊕ b
end
```

What's Going On Here?

Instead of linear induction
with one base case (empty),
we have multiway induction
with two base cases (empty and unit).

Why are these two base cases important?

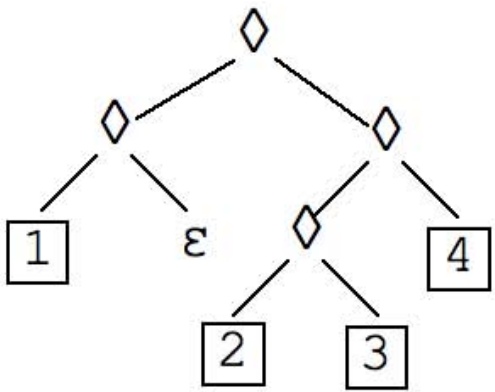
Representation of Abstract Collections

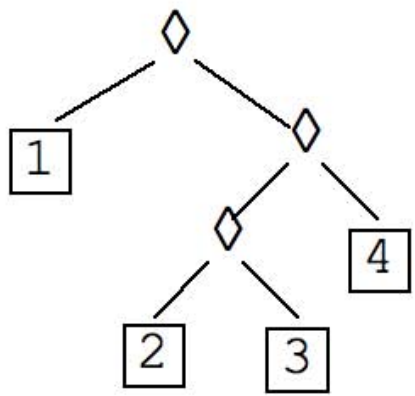
Binary operator ◊

Leaf operator ("unit") □

Optional empty collection ("zero") ε

that is the identity for ◊



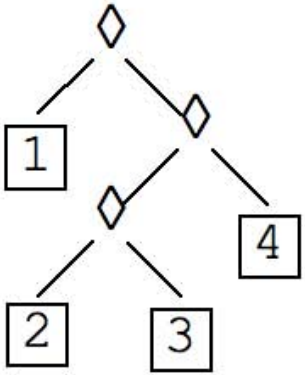


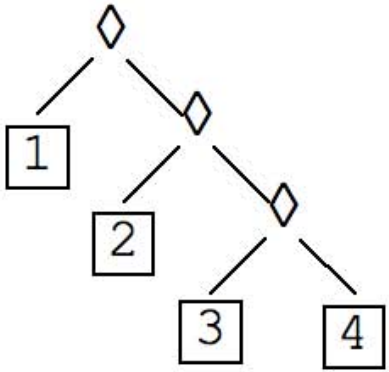
Algebraic Properties of $\diamond$

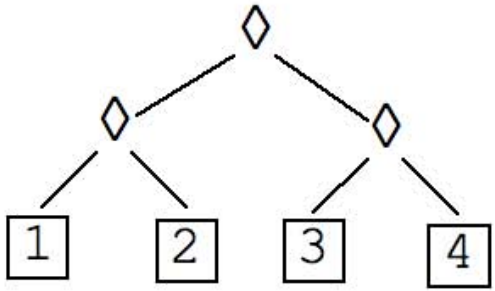
Associative	Commutative	Idempotent	
no	no	no	binary trees
no	no	yes	weird
no	yes	no	"mobiles"
no	yes	yes	weird
yes	no	no	lists (arrays)
yes	no	yes	weird
yes	yes	no	multisets (bags)
yes	yes	yes	sets

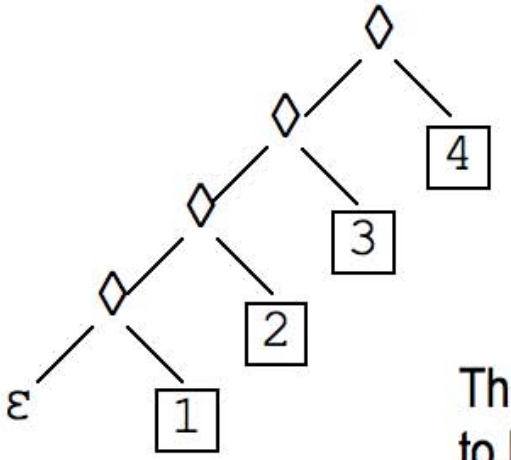
The "Boom hierarchy"

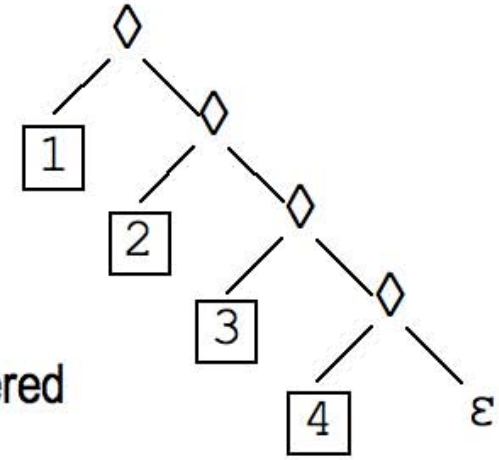
Associativity







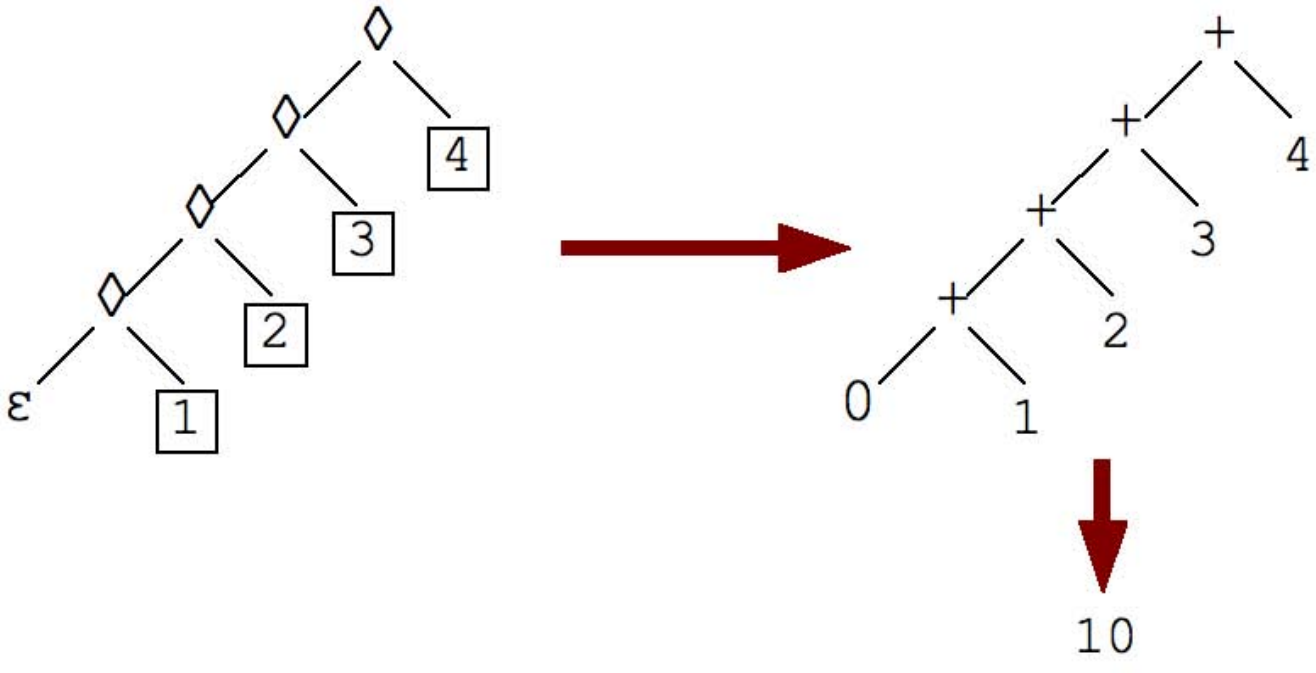




These are all considered to be equivalent.

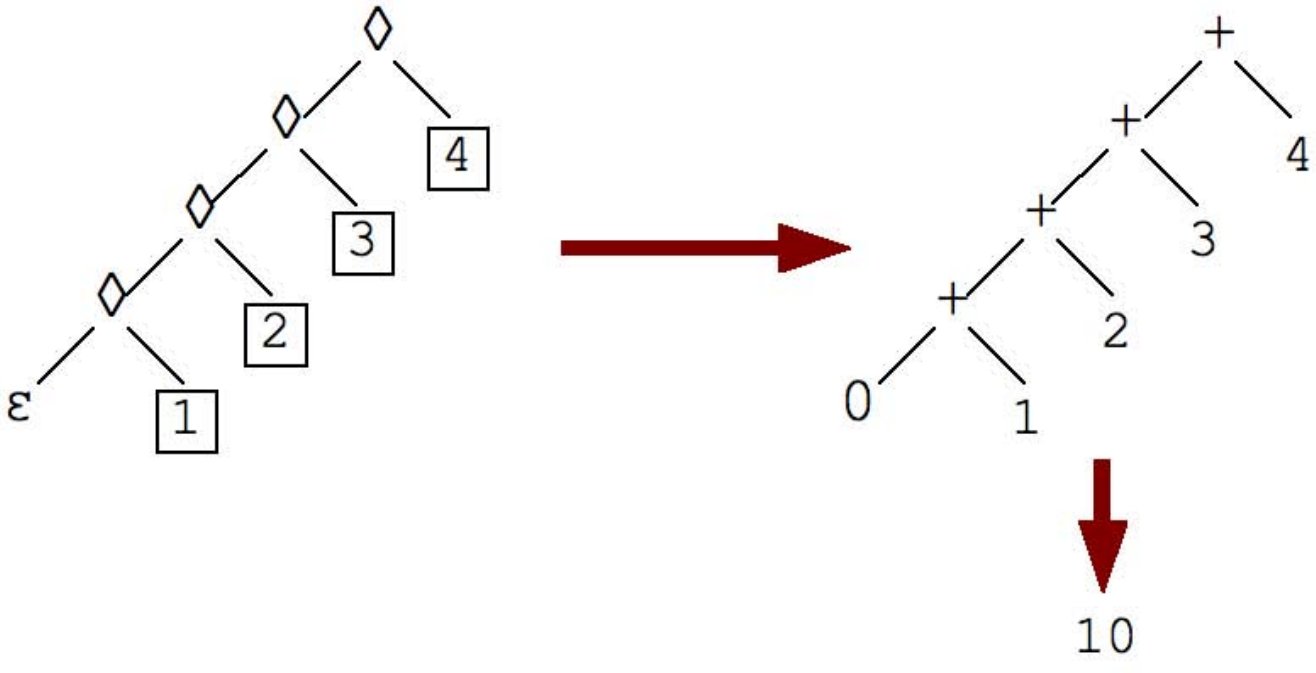
Catamorphism: Summation

Replace ◊ □ ε with + identity 0



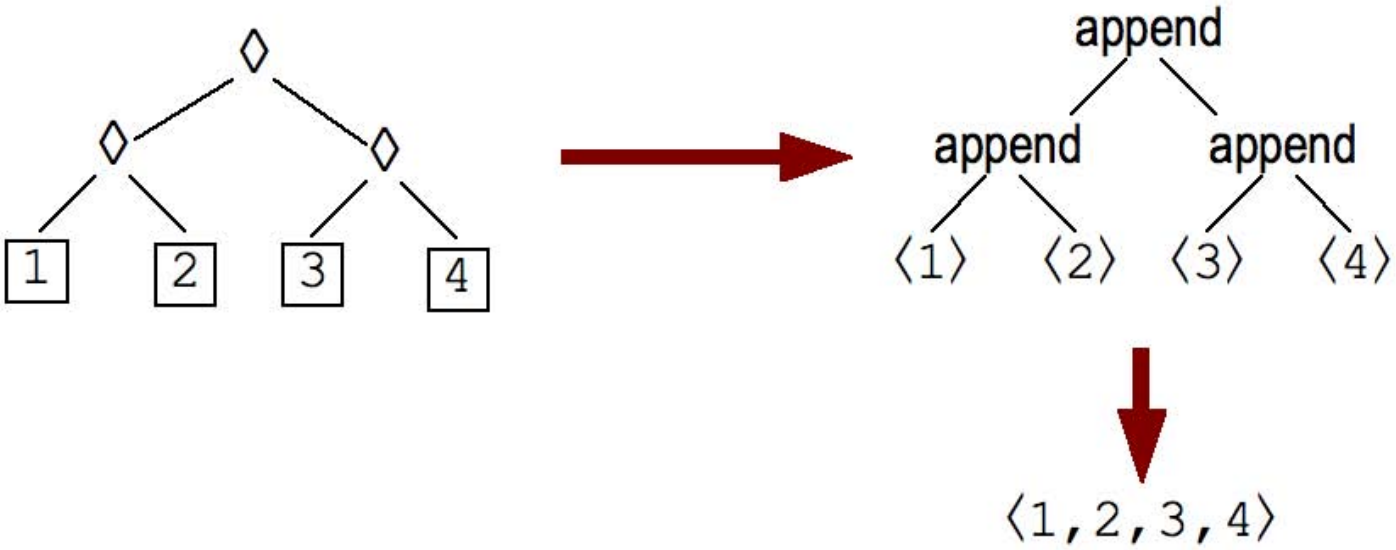
Computation: Summation

Replace ◊ □ ε with + identity 0



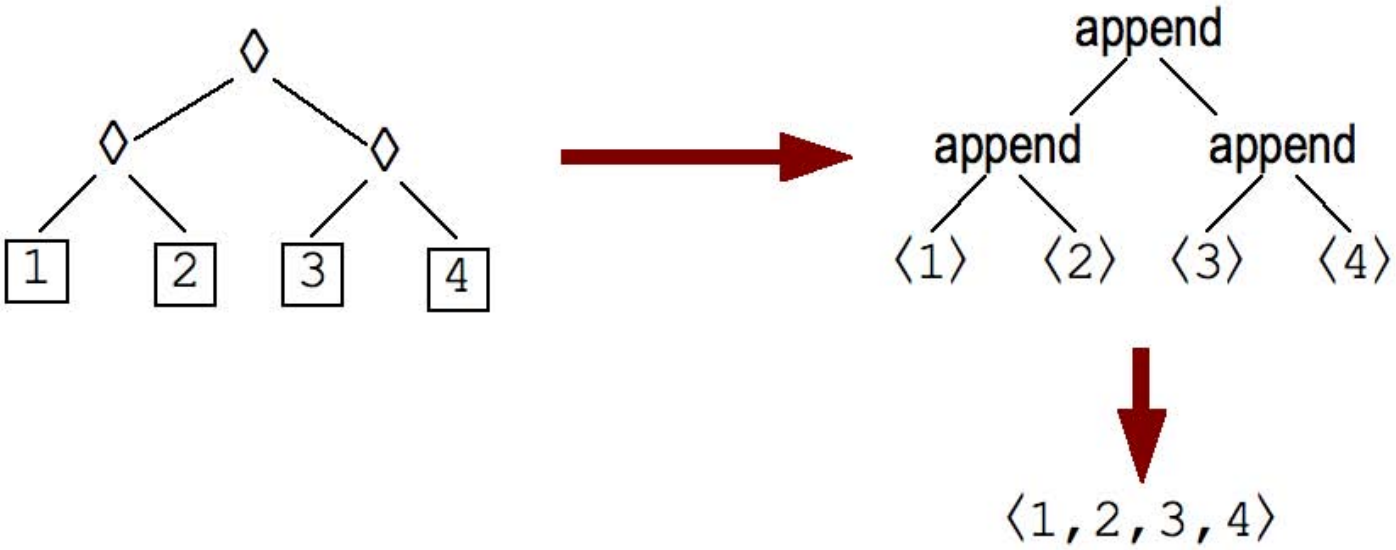
Catamorphism: Lists

Replace ◊ □ ε with append ⟨–⟩ ⟨⟩



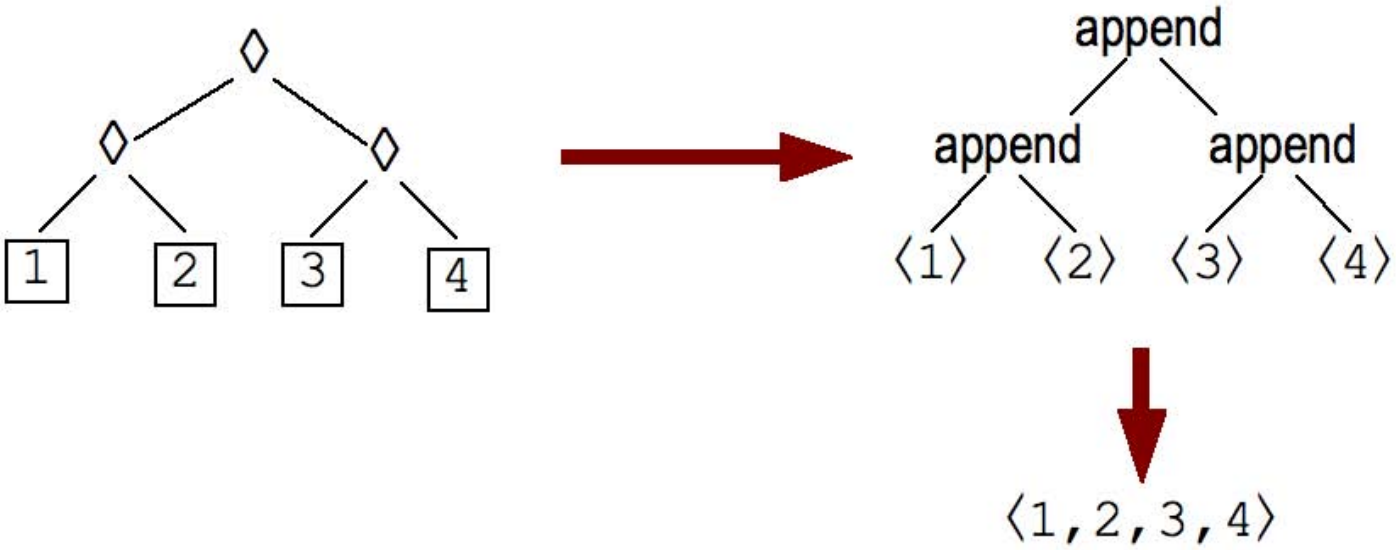
Computation: Lists

Replace ◊ □ ε with append ⟨–⟩ ⟨⟩



Representation: Lists

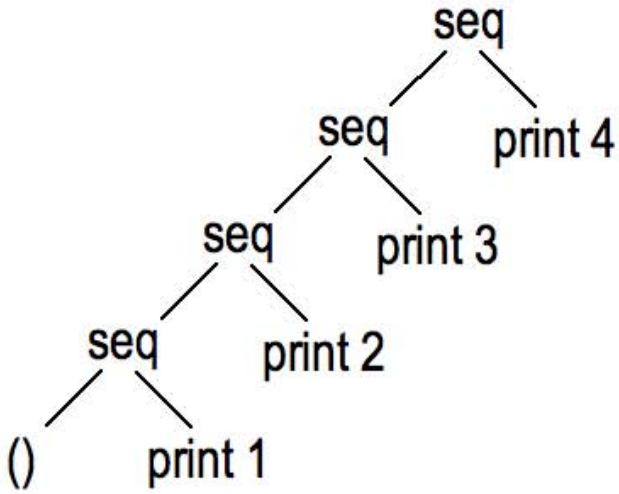
Replace ◊ □ ε with append ⟨–⟩ ⟨⟩

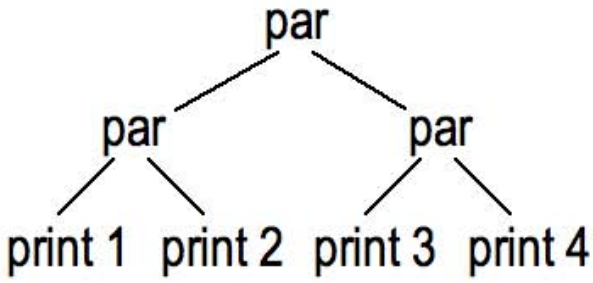


Catamorphism: Loops

Replace ◊ □ ε with seq identity () or par identity ()
where seq: (),() → () and par: (),() → ()

`for` $i \leftarrow seq(1:4)$ `do` $print\ i$ `end`





`for` $i \leftarrow 1:4$ `do` $print\ i$ `end`

Generators can modify the catamorphism and so control the parallelism.

To Summarize: A Big Idea

- Loops and summations and list constructors are alike!

$$\texttt{for } i \leftarrow 1:1000000 \texttt{ do } x_{\mathrm{i}} := x_{\mathrm{i}}^2 \texttt{ end}$$

$$\sum_{i \leftarrow 1:1000000} x_{\mathrm{i}}^2$$

$$\langle\, x_{\mathrm{i}}^2 \mid i \leftarrow 1:1000000 \,\rangle$$

  - Generate an abstract collection
  - The *body* computes a function of each item
  - Combine the results (or just synchronize)

- Whether to be sequential or parallel is a separable question
  - That's why they are especially good abstractions!
  - Make the decision on the fly, to use available resources

Another Big Idea

- Formulate a sequential loop as successive applications of state transformation functions f_i
- Find an *efficient* way to compute and represent compositions of such functions (this step requires ingenuity)
- Instead of computing
  $s := s_0; \texttt{for}\ i \leftarrow seq(1:1000000)\ \texttt{do}\ s := f_i(s)\ \texttt{end}$,
  compute $s := (\underset{i \leftarrow 1:1000000}{\circ} f_i)\ s_0$
- Because function composition is associative, the latter has a parallel strategy
- In the "words in a string" problem, each character can be regarded as defining a state transformation function

Splitting a String into Words (3, again)

```
words(s: String) = do
  result: List⟦String⟧ := ⟨ ⟩
  word: String := ""
  for k ← seq(0 # length(s)) do
    char = substring(s, k, k + 1)
    if (char = " ") then
      if (word ≠ "") then result := result || ⟨ word ⟩ end
      word := ""
    else
      word := word || char
    end
  end
  if (word ≠ "") then result := result || ⟨ word ⟩ end
  result
end
```

MapReduce Is a Big Deal!

- Associative combining operators are a VERY BIG DEAL!
  - Google MapReduce requires that combining operators also be commutative.
  - There are ways around that.
- Inventing new combining operators is a very, very big deal.
  - Creative catamorphisms!
  - We need programming languages that encourage this.
  - We need assistance in proving them associative.

We Need a New Mindset

- DO loops are so 1950s! (Literally: Fortran is now 50 years old.)
- So are linear linked lists! (Literally: Lisp is now 50 years old.)
- Java™-style iterators are **so** last millennium!
- Even arrays are suspect!
- As soon as you say "first, `SUM = 0`" you are hosed. Accumulators are BAD.
- If you say, "process subproblems in order," you lose.
- The great tricks of the sequential past DON'T WORK.
- The programming idioms that have become second nature to us as everyday tools DON'T WORK.

The Parallel Future

- We need new strategies for problem decomposition.
  - Data structure design/object relationships
  - Algorithmic organization
  - Don't split a problem into "the first" and "the rest."
  - Do split a problem into roughly equal pieces.
    Then figure out how to combine general subsolutions.
  - Often this makes combining the results a bit harder.
- We need programming languages and runtime implementations that support parallel strategies *and* hybrid sequential/parallel strategies.
- We must learn to manage new space-time tradeoffs.

Conclusion

- A program organized according to linear problem decomposition principles can be really hard to parallelize.
- A program organized according to parallel problem decomposition principles is easily run either in parallel or sequentially, according to available resources.
- The new strategy has costs and overheads. They will be reduced over time but will not disappear.
- In a world of parallel computers of wildly varying sizes, this is our only hope for program portability in the future.
- Better language design can encourage better parallel programming.