Organizing Functional Code for Parallel Execution

or, `foldl` and `foldr` Considered Slightly Harmful

Guy Steele

Sun Fellow

Sun Microsystems Laboratories

August 2009

The Big Messages

- Effective parallelism uses trees.
- Associative combining operators are good.
- MapReduce is good. Catamorphisms are good.
- There are systematic strategies for parallelizing superficially sequential code.
- We must lose the “accumulator” paradigm and emphasize “divide-and-conquer.”

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?

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

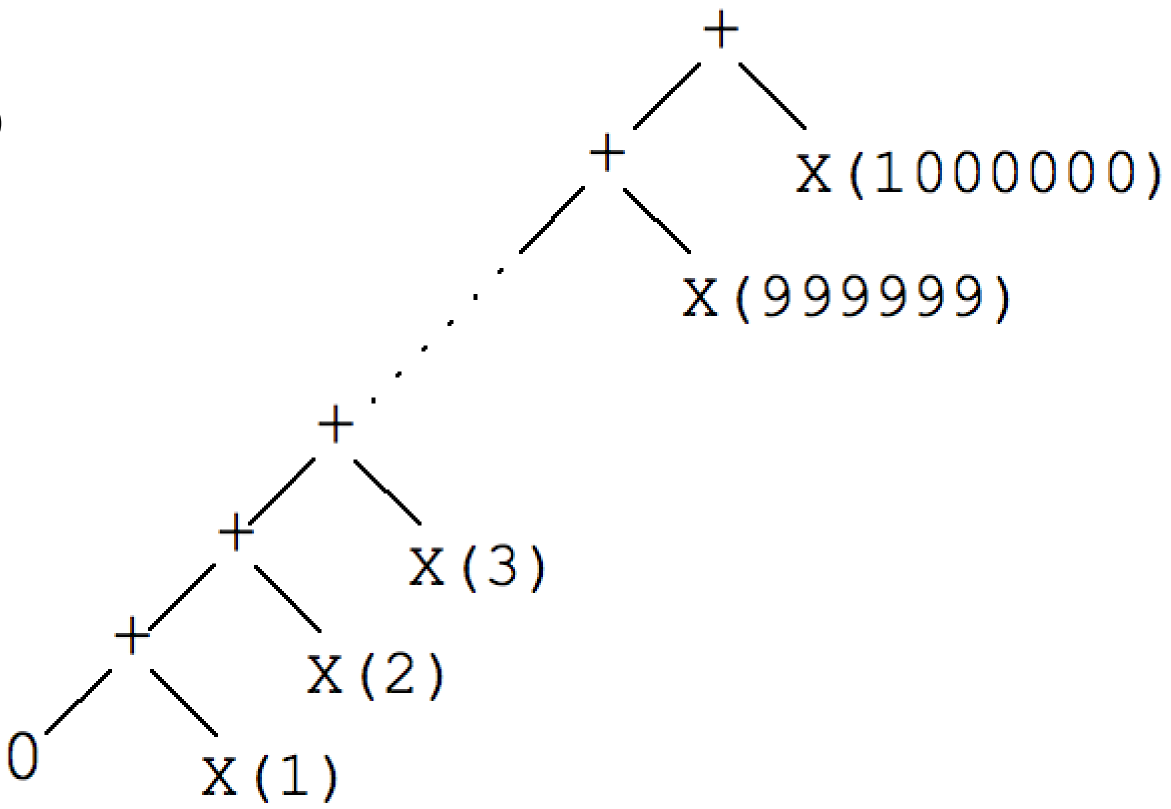
Compare Fortran 90 `SUM(X)`.

What, not how.

No commitment yet as to strategy. This is good.

Sequential Computation Tree

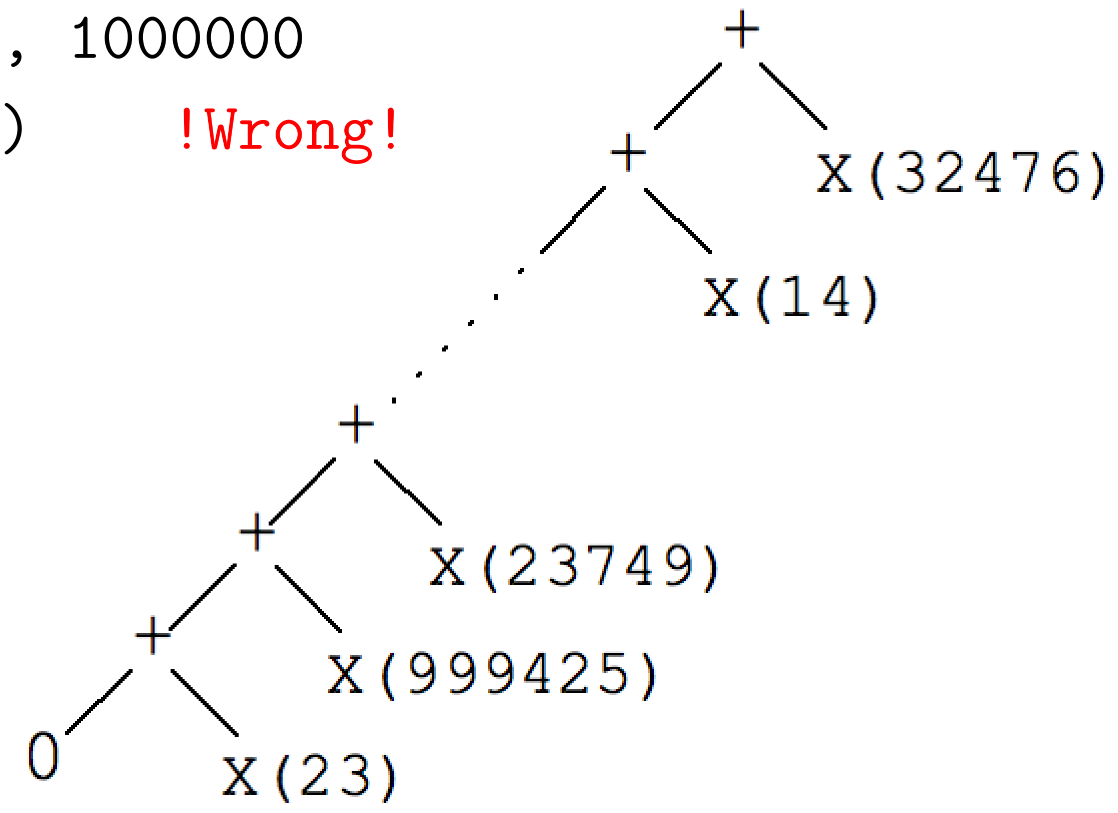
```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```



Atomic Update Computation Tree (a)

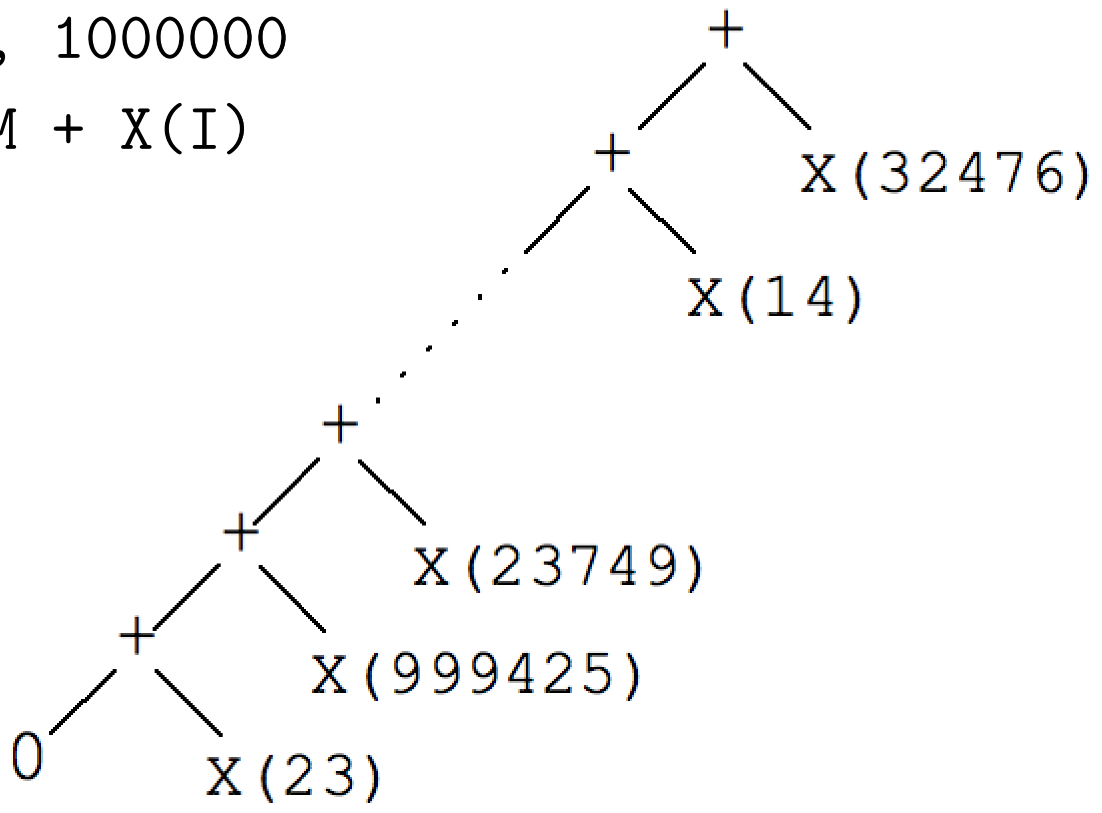
```
SUM = 0
PARALLEL DO I = 1, 1000000
  SUM = SUM + X(I)      !Wrong!
END DO
```

Race condition
can cause updates
to be lost.



Atomic Update Computation Tree (b)

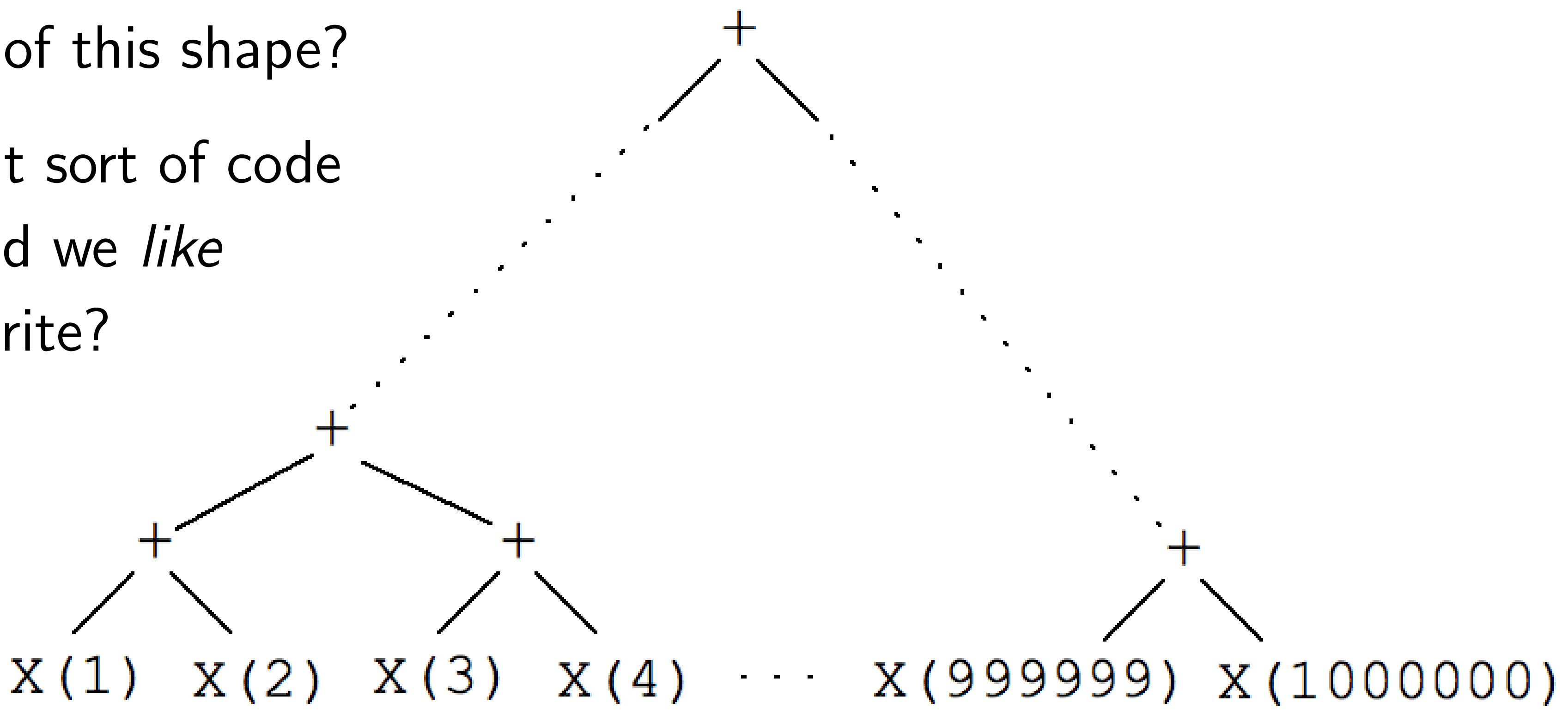
```
SUM = 0
PARALLEL DO I = 1, 1000000
  ATOMIC SUM = SUM + X(I)
END DO
```



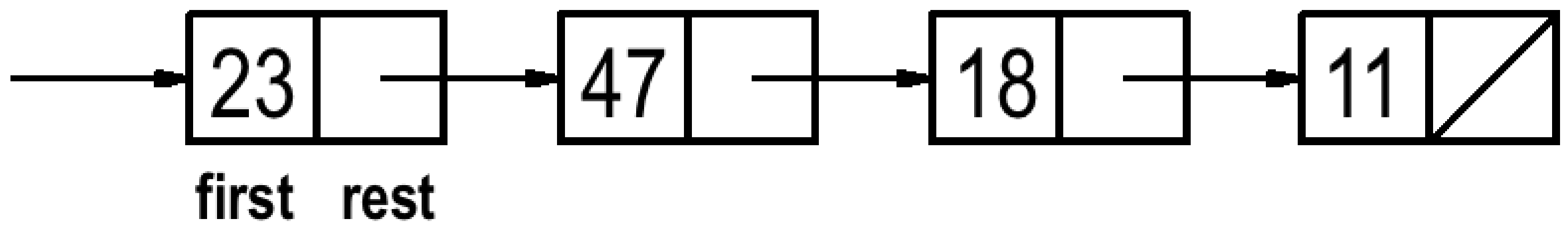
Parallel Computation Tree

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))))))
```

Total work: $\Theta(n)$

Delay: $\Omega(n)$

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:

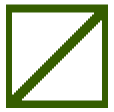
```
length [ ] = 0
length [a] = 1
length (a++b) = (length a) + (length b)
```

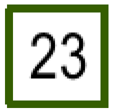
This is just a summation problem: adding up a bunch of 1's!

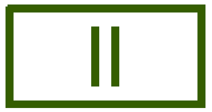
Total work: $\Theta(n)$

Delay: $\Omega(\log n), O(n)$ depending on how a++b is split;
even worse if splitting has worse than constant cost

Conc Lists



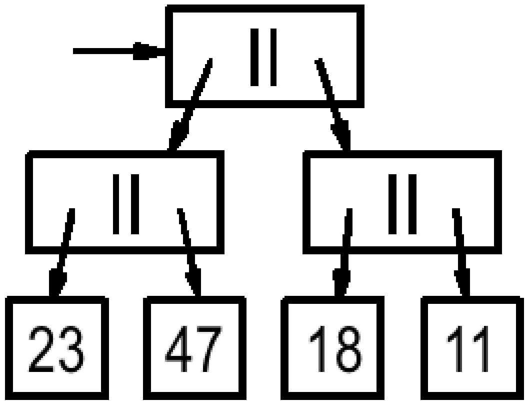




empty list — singleton — concatenation

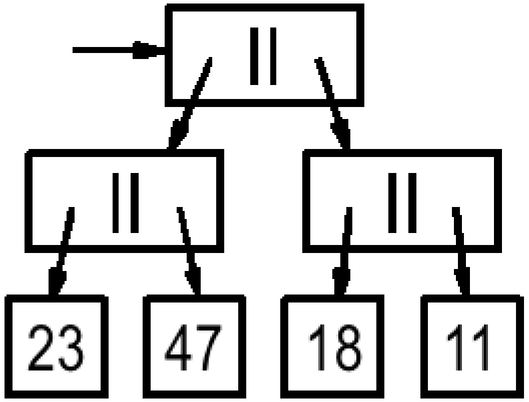
$\langle\,\rangle$ $\qquad$ $\langle\,23\,\rangle$ $\qquad$ $a \,||\, b$

Conc List for $\langle 23, 47, 18, 11 \rangle$ (1)

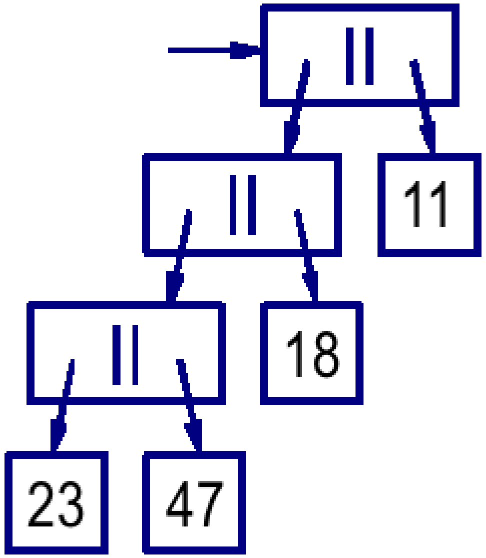


$$\Big(\langle\, 23 \,\rangle \parallel \langle\, 47 \,\rangle\Big) \parallel \Big(\langle\, 18 \,\rangle \parallel \langle\, 11 \,\rangle\Big)$$

Conc Lists for $\langle 23, 47, 18, 11 \rangle$ (2)

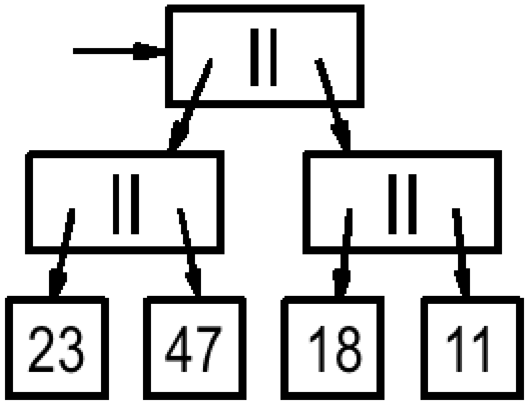


$$\Big(\langle\, 23 \,\rangle \parallel \langle\, 47 \,\rangle\Big) \parallel \Big(\langle\, 18 \,\rangle \parallel \langle\, 11 \,\rangle\Big)$$

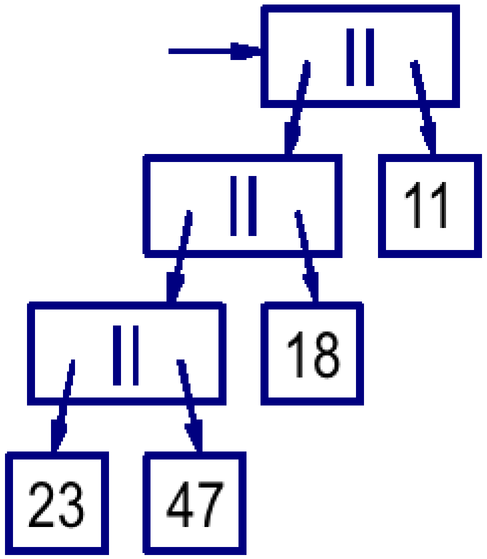


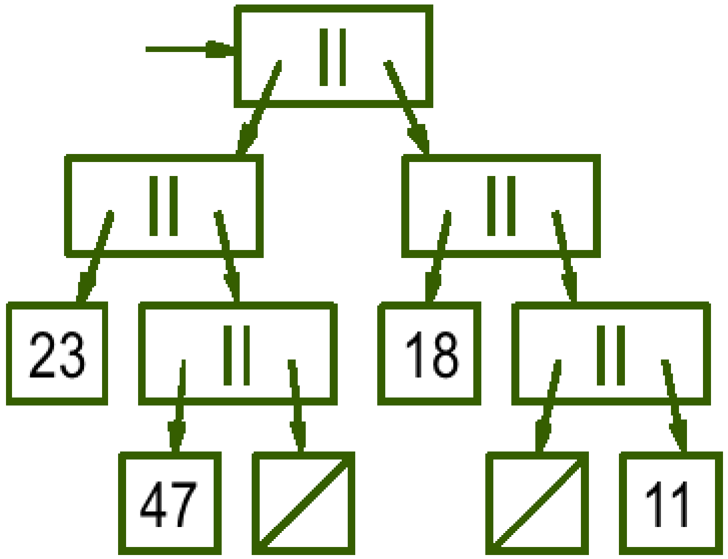
$$\Big(\Big(\langle\, 23 \,\rangle \parallel \langle\, 47 \,\rangle\Big) \parallel \langle\, 18 \,\rangle\Big) \parallel \langle\, 11 \,\rangle$$

Conc Lists for $\langle 23, 47, 18, 11 \rangle$ (3)

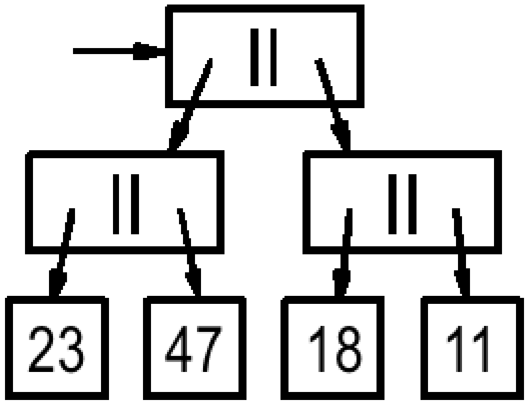


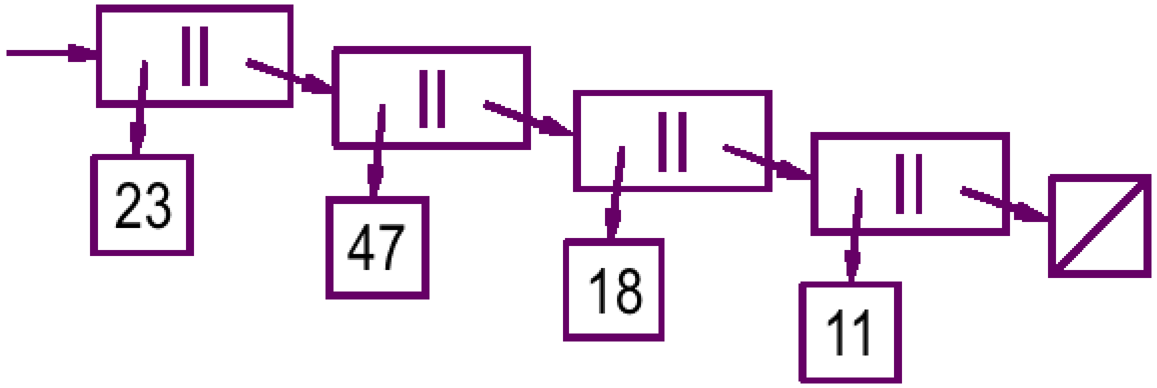
$$\Big(\langle\, 23 \,\rangle \parallel \Big(\langle\, 47 \,\rangle \parallel \langle\,\rangle\Big)\Big)$$
$$\parallel \Big(\langle\, 18 \,\rangle \parallel \Big(\langle\,\rangle \parallel \langle\, 11 \,\rangle\Big)\Big)$$

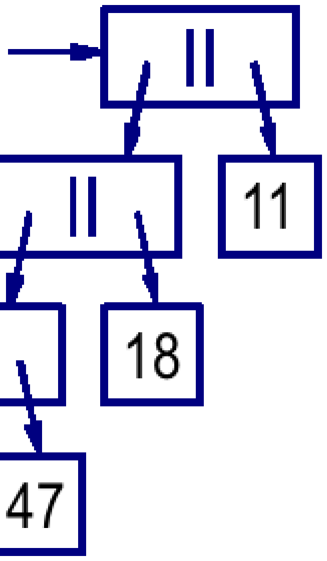


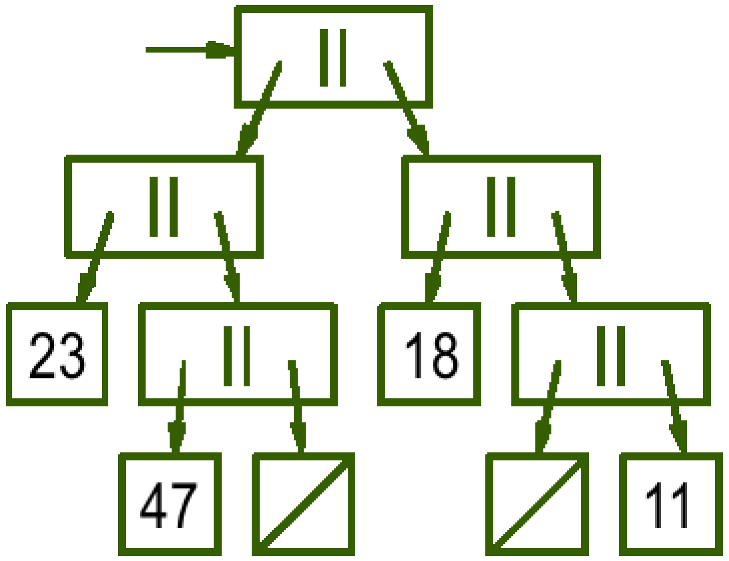


Conc Lists for $\langle 23, 47, 18, 11 \rangle$ (4)









Primitives on Lists (1)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs))` $=$ `xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`left`, `right`
	`(list (item s))` $=$ `s`		
	`(conc (left xs) (right xs))` $=$ `xs`		

Primitives on Lists (2)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs))` $=$ `xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`split`
	`(list (item s))` $=$ `s`		
	`(split xs (`λ` (ys zs) (conc ys zs)))` $=$ `xs`		

Primitives on Lists (3)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs))` = `xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`split`
	`(list (item s))` = `s`		
	`(split xs (λ (ys zs) (conc ys zs)))` = `xs`		
	`(split xs conc)` = `xs`		

Defining Lists Using `car cdr cons` (1)

```
(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` (2)

```
(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)))))
```

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)))))))
```

Defining Lists Using `item list split conc` (4)

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

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

map reduce mapreduce **Using** car cdr cons

```
(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 **Using** car cdr cons

```
(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))
```

The latter analysis depends on a crucial fact: in this situation, each call to append will require constant, not linear, time!

reverse **Using** car cdr cons

```
(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 '()))
```

Structural recursion on cons lists produces poor performance for `reverse`. An accumulation trick gets it down to linear time.

Parallel map reduce mapreduce Using item list split conc

```
(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)              ;Opportunity for
                   (mapreduce f g id zs)))))))        ; parallelism

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

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

Parallel `length filter reverse` Using `item list split conc`

```
(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 (ys zs) (append zs ys)) '() xs))
```

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

- 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
- Mergesort: structural induction on input
  - Cheaply split input in half
  - Recursively sort the two halves
  - Carefully merge the two sorted sublists (tricky)

Filters in Fortress (1)

```
$sequentialFilter[\![E]\!](p\colon E \rightarrow \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!] =$ do
    $result\colon \mathrm{List}[\![E]\!] := \langle\,\rangle$
    for $a \leftarrow seq(xs)$ do
        if $p(a)$ then $result := result.addRight(a)$ end
    end
    $result$
end
```

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

Example of use:

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

$sequentialFilter(odd, \langle\, 1, 4, 7, 2, 5, 3 \,\rangle)$ produces $\langle\, 1, 7, 5, 3 \,\rangle$

Filters in Fortress (2)

```
$recursiveFilter[\![E]\!](p\colon E \rightarrow \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!] =$
  if $xs.isEmpty()$ then $\langle\,\rangle$
  else
    $(first, rest) = xs.extractLeft().get()$
    $rest' = recursiveFilter(rest, p)$
    if $p(first)$ then $rest'.addLeft(first)$ else $rest'$ end
  end
```

Still linear-time delay.

Filters in Fortress (3a)

```
$parallelFilter[\![E]\!](p\colon E \rightarrow \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$
  if $|xs| = 0$ then $\langle\,\rangle$
  elif $|xs| = 1$ then
    $(x, \_) = xs.extractLeft().get()$
    if $p(x)$ then $\langle x \rangle$ else $\langle\,\rangle$ end
  else
    $(x, y) = xs.split()$
    $parallelFilter(x, p) \parallel parallelFilter(y, p)$
  end
```

Filters in Fortress (3b)

```
$parallelFilter[\![E]\!](p\colon E \rightarrow \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!] =$
  if $|xs| = 0$ then $\langle\,\rangle$
  elif $|xs| = 1$ then
    $(x, \_) = xs.extractLeft().get()$
    if $p(x)$ then $\langle x \rangle$ else $\langle\,\rangle$ end
  else
    $(x, y) = xs.split()$
    $parallelFilter(x, p) \parallel parallelFilter(y, p)$
  end
```

Filters in Fortress (3c)

```
$parallelFilter[\![E]\!](p\colon E \rightarrow \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$
  if $|xs| = 0$ then $\langle\,\rangle$
  elif $|xs| = 1$ then
    $(x, \_) = xs.extractLeft().get()$
    if $p(x)$ then $\langle x \rangle$ else $\langle\,\rangle$ end
  else
    $(x, y) = xs.split()$
    $parallelFilter(x, p) \mathbin{\|} parallelFilter(y, p)$
  end
```

```
$reductionFilter[\![E]\!](p\colon E \rightarrow \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$
  $\underset{x \leftarrow xs}{\|}$ (if $p(x)$ then $\langle\, x \,\rangle$ else $\langle\,\rangle$ end)
```

Filters in Fortress (4)

Actually, filters are so useful that they are built into the Fortress comprehension notation in the usual way:

$$\mathit{comprehensionFilter}[\![E]\!](p\colon E \to \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!] = \langle\, x \mid x \leftarrow xs, p(x)\,\rangle$$

Oh, yes: $\displaystyle\sum_{i \leftarrow 1:1000000} x_i$ and $\displaystyle\mathop{\mathtt{MAX}}_{i \leftarrow 1:1000000} x_i$

or maybe: $\displaystyle\sum_{a \leftarrow x} a$ and $\displaystyle\mathop{\mathtt{MAX}}_{a \leftarrow x} a$

or maybe just: $\sum x$ and $\mathtt{MAX}\, x$

Point of Order

For `filter`, unlike summation, we rely on maintaining the original order of the elements in the input list. (Both $\|$ and $+$ are associative, but only $+$ is commutative.)

Do not confuse the ordering of elements in the result list (which is a spatial order) with the order in which they are computed (which is a temporal order).

Sequential programming often ties the one to the other. Good parallel programming decouples this unnecessary dependency.

This strategy for parallelism relies only on associativity, *not* commutativity.

Conjugate Transforms

A very simple but very powerful idea.

Instead of mapping input items directly to output data type T:

- Map inputs (maybe by way of T) to a *richer* data type U.
- Perform computations in this richer space U
  (chosen to make computation simpler or faster).
- Finally, project the result from U back into T.

The Three-way Unshuffle Problem (1)

- Goal: deal a deck of cards into three piles.
- Example: from $\langle a, b, c, d, e, f, g, h, i, j, k \rangle$,
  produce $(\langle a, d, g, j \rangle, \langle b, e, h, k \rangle, \langle c, f, i \rangle)$.
- Base cases:
  - > $\langle\,\rangle$ yields $(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle)$
  - > $\langle a \rangle$ yields $(\langle a \rangle, \langle\,\rangle, \langle\,\rangle)$
- Combining: let's consider input $\langle a, b, c, d \rangle$
  - > $(\langle a \rangle, \langle\,\rangle, \langle\,\rangle)$ plus $(\langle b \rangle, \langle\,\rangle, \langle\,\rangle)$ yields $(\langle a \rangle, \langle b \rangle, \langle\,\rangle)$
  - > $(\langle c \rangle, \langle\,\rangle, \langle\,\rangle)$ plus $(\langle d \rangle, \langle\,\rangle, \langle\,\rangle)$ yields $(\langle c \rangle, \langle d \rangle, \langle\,\rangle)$
  - > $(\langle a \rangle, \langle b \rangle, \langle\,\rangle)$ plus $(\langle c \rangle, \langle d \rangle, \langle\,\rangle)$ yields $(\langle a, d \rangle, \langle b \rangle, \langle c \rangle)$

  We always perform three concatenations;
  we just need to pair them up correctly. How?

The Three-way Unshuffle Problem (2)

```
$unshuffle(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!]) =$
  if $|xs| = 0$ then $(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle)$
  elif $|xs| = 1$ then $(xs, \langle\,\rangle, \langle\,\rangle)$
  else $(ys, zs) = xs.split()$
       $((a, b, c), (d, e, f)) = (unshuffle\ ys, unshuffle\ zs)$
       if $|c| = |a|$ then $(a \parallel d, b \parallel e, c \parallel f)$
       elif $|b| = |a|$ then $(a \parallel e, b \parallel f, c \parallel d)$
       else $(a \parallel f, b \parallel d, c \parallel e)$
       end
  end
```

Unfortunately, the tests $|c| = |a|$ and $|b| = |a|$ are slow.

The Three-way Unshuffle Problem (3)

```
$unshuffle'(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!], \mathbb{Z}) =$
  if $|xs| = 0$ then $(\langle\,\rangle, \langle\,\rangle, \langle\,\rangle, 0)$          ⊛ Solution: project inputs into
  elif $|xs| = 1$ then $(xs, \langle\,\rangle, \langle\,\rangle, 1)$       ⊛ a space of 4-tuples, not 3-tuples.
  else $(ys, zs) = xs.split()$
       $((a, b, c, j), (d, e, f, k)) = (unshuffle'\ ys, unshuffle'\ zs)$
       case $j$ of
         $0 \Rightarrow (a \parallel d, b \parallel e, c \parallel f, (j + k)$ MOD $3)$
         $1 \Rightarrow (a \parallel f, b \parallel d, c \parallel e, (j + k)$ MOD $3)$
         $2 \Rightarrow (a \parallel e, b \parallel f, c \parallel d, (j + k)$ MOD $3)$
       end
  end

$unshuffle(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!], \mathrm{List}[\![\mathbb{Z}]\!]) =$ do
  $(as, bs, cs, m) = unshuffle'\ xs$      ⊛ Now project the result 4-tuple
  $(as, bs, cs)$                          ⊛ to the desired 3-tuple.
end
```

The Three-way Unshuffle Problem (4)

Abstract the combining operator as $\boxplus$ and then use "big $\boxplus$":

```
opr $\boxplus$($p$: (List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧, $\mathbb{Z}$), $q$: (List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧, $\mathbb{Z}$) ) = do
  $((a, b, c, j), (d, e, f, k)) = (p, q)$
  case $j$ of
    $0 \Rightarrow (a \parallel d, b \parallel e, c \parallel f, (j + k)$ MOD $3)$
    $1 \Rightarrow (a \parallel f, b \parallel d, c \parallel e, (j + k)$ MOD $3)$
    $2 \Rightarrow (a \parallel e, b \parallel f, c \parallel d, (j + k)$ MOD $3)$
  end
end

$unshuffle(xs\colon$ List⟦$\mathbb{Z}$⟧$)\colon$ (List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧, List⟦$\mathbb{Z}$⟧) = do
  $(as, bs, cs, m) = \underset{x \leftarrow xs}{\boxplus} (\langle x \rangle, \langle\,\rangle, \langle\,\rangle, 1)$
  $(as, bs, cs)$
end
```

Mergesort Is a Catamorphism

Abstract merging as operator `MERGE` and then use "big `MERGE`":

```
opr MERGE($p$: List⟦$\mathbb{Z}$⟧, $q$: List⟦$\mathbb{Z}$⟧) = do
  ⊛ Merge two sorted lists into a new sorted list
  . . .
end
```

(The `MERGE` operator is associative and has identity $\langle\,\rangle$;
thus lists under the `MERGE` operator form a *monoid*.)

$$mergesort(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!] = \underset{x \leftarrow xs}{\mathtt{MERGE}} \langle x \rangle$$

The Parallel Prefix Problem (1)

- Goal: compute the running totals and final sum of a sequence.
- Example: from $\langle a, b, c, d, e \rangle$, produce $(\langle 0, a, a+b, a+b+c, a+b+c+d \rangle, a+b+c+d+e)$.
- Example: from $\langle 1, 2, 3, 6, 3, 4, -5, 2, 9 \rangle$, produce $(\langle 0, 1, 3, 6, 9, 13, 17, 12, 14 \rangle, 23)$.
- This is not the only possible formulation, but it is convenient for our purposes here.
- Parallel prefix is useful with any monoid; for simplicity we will restrict our attention to addition on integers.

The Parallel Prefix Problem (2)

$$ppsum(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathbb{Z}) =$$
$$\left(\left\langle \sum xs[0:k-1] \,\middle|\, k \leftarrow 0:|xs|-1 \right\rangle, \sum xs \right)$$

Unfortunately, the total work here is quadratic in $|xs|$.

The Parallel Prefix Problem (3)

```
$ppsum(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathbb{Z}) =$
  if $|xs| = 0$ then $(\langle\,\rangle, 0)$
  elif $|xs| = 1$ then $(\langle 0\rangle, xs_0)$
  else $(ys, zs) = xs.split()$
       $((ps, a), (qs, b)) = (ppsum\; ys, ppsum\; zs)$
       $ps \parallel \langle\, a + q \mid q \leftarrow qs \,\rangle$
  end
```

Now the total work is linear in $|xs|$.

Unfortunately, the dependency on a results in delay $\Omega((\log n)^2)$.

The Parallel Prefix Problem (4)

```
$ppsum(xs\colon \text{List}[\![\mathbb{Z}]\!])\colon (\text{List}[\![\mathbb{Z}]\!], \mathbb{Z}) = ppsum'(0, xs)$

$ppsum'(k\colon \mathbb{Z}, xs\colon \text{List}[\![\mathbb{Z}]\!])\colon (\text{List}[\![\mathbb{Z}]\!], \mathbb{Z}) =$
  if $|xs| = 0$ then $(\langle\,\rangle, k)$
  elif $|xs| = 1$ then $(\langle k \rangle, k + xs_0)$
  else $(ys, zs) = xs.split()$
       $(ps, a) = ppsum'(k, ys)$
       $(qs, b) = ppsum'(a, zs)$
       $(ps \parallel qs, b)$
  end
```

One pass, and again the total work is linear in $|xs|$.

Unfortunately, the dependency on a results in delay $\Omega(n)$.

The Parallel Prefix Problem (5)

A solution in NESL from Guy Blelloch's 2009 PPoPP talk:

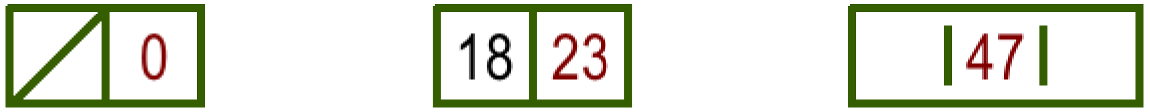
```
function scan(A, op) =
if (#A <= 1) then [0]
else let
  sums = {op(A[2*i], A[2*i+1]) : i in [0:#A/2]};
  evens = scan(sums, op);
  odds = {op(evens[i], A[2*i]) : i in [0:#A/2]};
in interleave(evens,odds);
```

See slide 11 of http://www.cs.cmu.edu/~blelloch/papers/PPoPP09.pdf

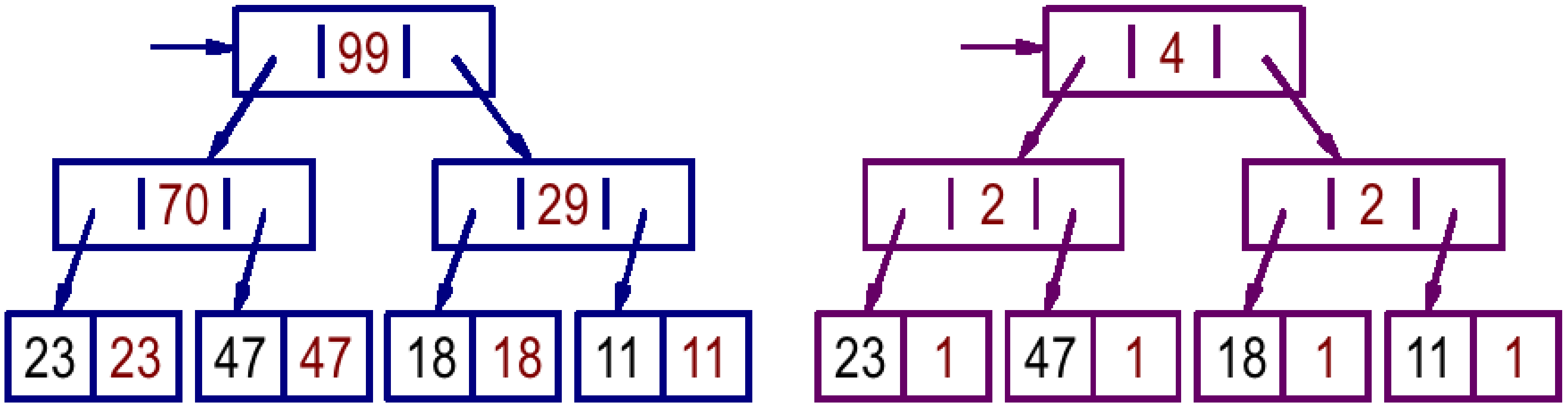
Using tree representation: total work $\Theta(n)$, delay $\Omega((\log n)^2)$

Can we do better?

Monoid-cached Trees



Empty Leaf Node



tree with cached sums tree with cached lengths

See Hinze, Ralf, and Paterson, Ross. “Finger Trees: A Simple General-purpose Data Structure.” Journal of Functional Programming 16 (2): 2006, 197–217.

Declaring Monoid-cached Trees in Fortress

```
trait ValueTree⟦T, V⟧ comprises { Empty⟦T, V⟧, Leaf⟦T, V⟧, Node⟦T, V⟧ }
  val: V
end

object Empty⟦T, V⟧(val: V) extends ValueTree⟦T, V⟧ end

object Leaf⟦T, V⟧(item: T, val: V) extends ValueTree⟦T, V⟧ end

object Node⟦T, V⟧(left: ValueTree⟦T, V⟧, val: V, right: ValueTree⟦T, V⟧)
      extends ValueTree⟦T, V⟧
end
```

The Parallel Prefix Problem (6)

```
$ppsum(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon (\mathrm{List}[\![\mathbb{Z}]\!], \mathbb{Z}) =$ do
  $c = sumcache\ xs$
  $(ppfinish(0, c), c.val)$
end
```

Total work $\Theta(n)$

Delay $\Omega(\log n)$

```
$sumcache(xs\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{ValueTree}[\![\mathbb{Z}, \mathbb{Z}]\!] =$
  if $|xs| = 0$ then $\mathrm{Empty}[\![\mathbb{Z}, \mathbb{Z}]\!](0)$
  elif $|xs| = 1$ then $\mathrm{Leaf}[\![\mathbb{Z}, \mathbb{Z}]\!](xs_0, xs_0)$
  else $(ys, zs) = xs.split()$
       $(p, q) = (sumcache\ ys, sumcache\ zs)$
       $\mathrm{Node}[\![\mathbb{Z}, \mathbb{Z}]\!](p, p.val + q.val, q)$
  end
```

```
$ppfinish(k\colon \mathbb{Z}, \_\colon \mathrm{Empty}[\![\mathbb{Z}, \mathbb{Z}]\!]) = \langle\,\rangle$
$ppfinish(k\colon \mathbb{Z}, \_\colon \mathrm{Leaf}[\![\mathbb{Z}, \mathbb{Z}]\!]) = \langle k \rangle$
$ppfinish(k\colon \mathbb{Z}, n\colon \mathrm{Node}[\![\mathbb{Z}, \mathbb{Z}]\!]) = ppfinish(k, n.left) \parallel ppfinish(k + n.left.val, n.right)$
```

It would be nice to have a simple facility to cache any monoid in a tree. It's straightforward to cache more than one monoid, because the cross-product of two monoids is a monoid. Deforestation of monoid-cached trees may turn out to be an important optimization.

MapScanZip (cf. MapReduce)

```
MapScanZip op id f xs =
      zip xs (scan op id (map f xs))
```

where `scan` is the parallel prefix operation, parameterized by an associative operation `op` and its identity `id`.

Monoid-cached trees provide a fast implementation.

Note that `zip` is difficult in the general case because the shapes of the trees might not match, but this case is easy.

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:

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

  $println\ words(\text{“}\texttt{~~Here~~is~~another~~sample~~}\text{”})$

  $println\ words(\text{“}\texttt{JustOneWord}\text{”})$

  $println\ words(\text{“}\texttt{~~~}\text{”})$

  $println\ words(\text{“”})$

- Expected output:

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

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

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

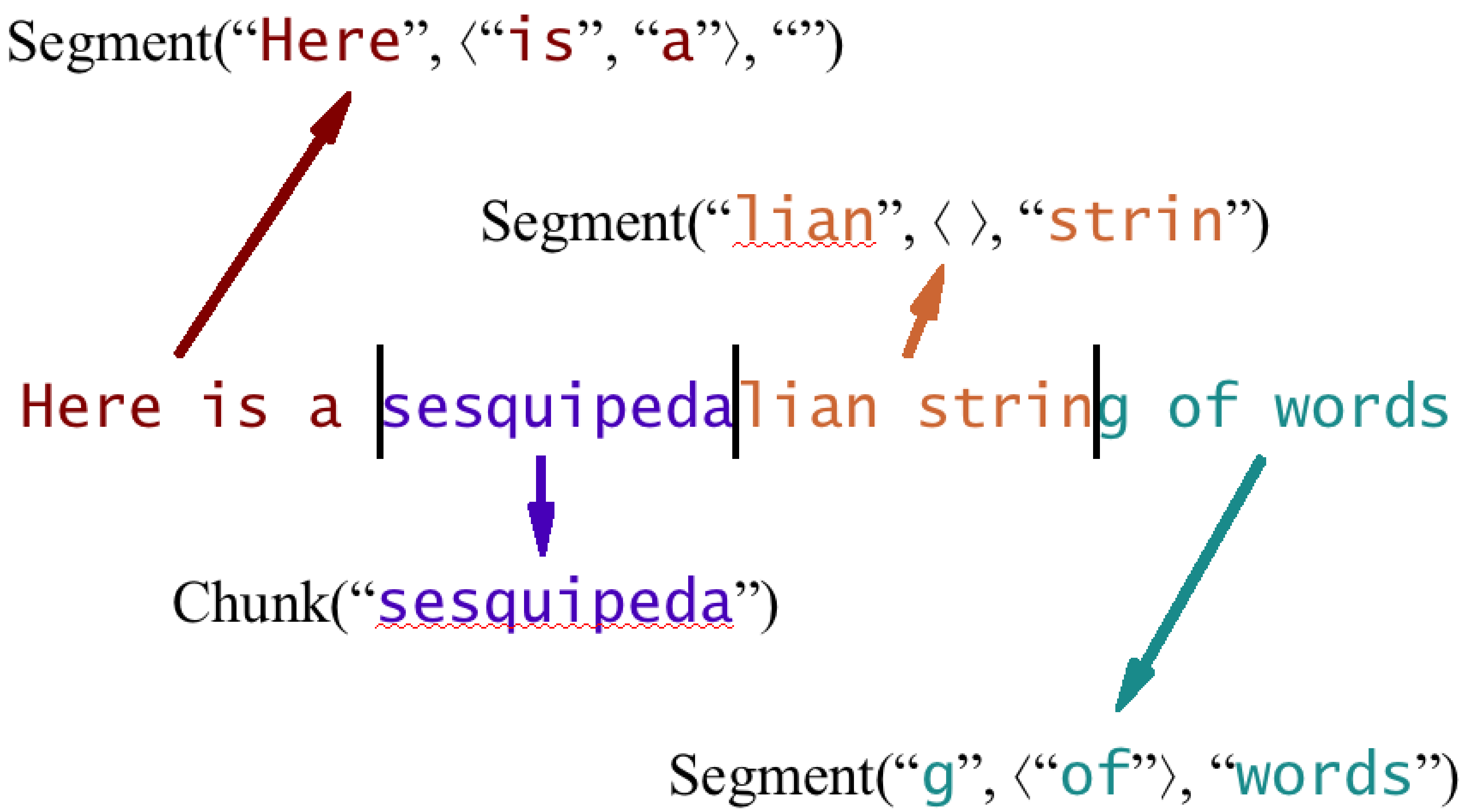
  $\langle\, \rangle$

  $\langle\, \rangle$

Splitting a String into Words (3)

```
$words(s\colon \mathrm{String}) =$ do
  $result\colon \mathrm{List}[\![\mathrm{String}]\!] := \langle\,\rangle$
  $word\colon \mathrm{String} :=$ “”
  for $c \leftarrow seq(s)$ do
    if $(c =$ ‘ ’$)$ then
      if $(word \neq$ “”$)$ then $result := result \parallel \langle\, word \,\rangle$ end
      $word :=$ “”
    else
      $word := word \parallel c$
    end
  end
  if $(word \neq$ “”$)$ then $result := result \parallel \langle\, word \,\rangle$ end
  $result$
end
```

Splitting a String into Words (4a)



Segment(“Here”, ⟨“is”, “a”⟩, “”)

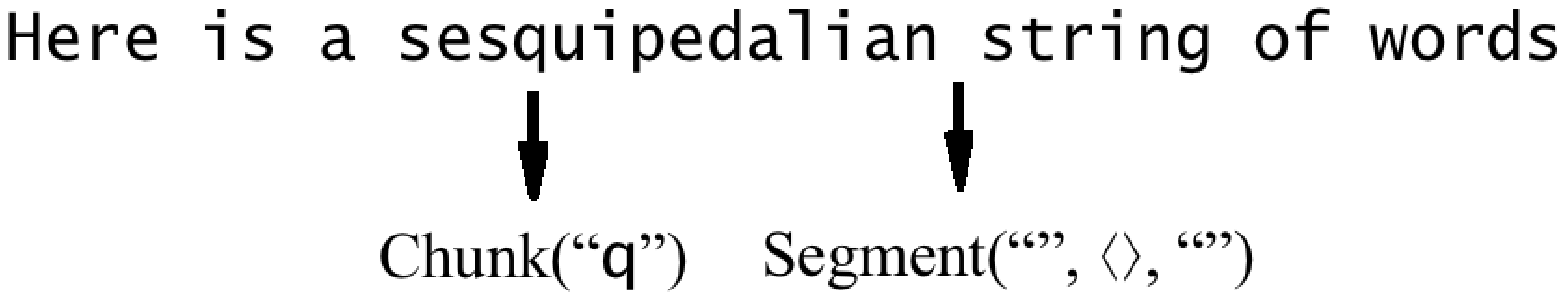
Segment(“lian”, ⟨ ⟩, “strin”)

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

Chunk(“sesquipeda”)

Segment(“g”, ⟨“of”⟩, “words”)

Splitting a String into Words (4b)



Splitting a String into Words (5)

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

trait $\mathrm{WordState}$
      extends $\{\, \mathrm{Associative}[\![\mathrm{WordState}, \oplus]\!] \,\}$
      comprises $\{\, \mathrm{Chunk}, \mathrm{Segment} \,\}$
  opr $\oplus$(self, $other\colon \mathrm{WordState})\colon \mathrm{WordState}$
end
```

Splitting a String into Words (6)

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

Splitting a String into Words (7)

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

Splitting a String into Words (8)

```
$processChar(c\colon \text{Character})\colon \text{WordState} =$
  if $(c = \text{` '})$ then $\text{Segment}(\text{``''}, \langle\,\rangle, \text{``''})$ else $\text{Chunk}(c)$ end

$words(s\colon \text{String}) =$ do
  $g = \bigoplus_{c \leftarrow s} processChar(c)$      ⊛ All the parallelism happens here
  typecase $g$ of
    $\text{Chunk} \Rightarrow maybeWord(g.s)$
    $\text{Segment} \Rightarrow maybeWord(g.l) \parallel g.A \parallel maybeWord(g.r)$
  end
end
```

Splitting a String into Words (9)

```
(* The mechanics of BIG OPLUS *)

opr BIG $\oplus[\![T]\!](g\colon(\mathrm{Reduction}[\![\mathrm{WordState}]\!],$
                        $T \rightarrow \mathrm{WordState})$
                      $\rightarrow \mathrm{WordState})\colon \mathrm{WordState} =$
  $g(\mathrm{GlomReduction}, \mathit{identity}[\![\mathrm{WordState}]\!])$

object GlomReduction extends $\mathrm{Reduction}[\![\mathrm{WordState}]\!]$
  getter $\mathit{toString}() =$ “GlomReduction”
  $\mathit{empty}()\colon \mathrm{WordState} = \mathrm{Chunk}(“”)$     ⊛ Identity value
  $\mathit{join}(a\colon \mathrm{WordState}, b\colon \mathrm{WordState})\colon \mathrm{WordState} = a \oplus b$
end
```

To Summarize: A Big Idea

- Summations and list constructors and loops are alike!

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

$$\langle\, x_i^2 \mid i \leftarrow 1:1000000 \,\rangle$$

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

  - > Generate an abstract collection
  - > The *body* computes a function of each item
  - > Combine the results (or just synchronize)
  - > In other words: generate-and-reduce
- 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 (or finite-state machine) 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
- If you need intermediate results, use parallel prefix function composition; then map down the result, applying each to s_0

See, for example, Hillis, W. D., and Steele, G. L. "Data parallel algorithms." Comm. ACM 29, 12 (Dec. 1986), 1170–1183.

Splitting a String into Words (3, again)

```
$words(s\colon \text{String}) =$ do
  $result\colon \text{List}[\![\text{String}]\!] := \langle\,\rangle$
  $word\colon \text{String} :=$ “”
  for $k \leftarrow seq(0\#length(s))$ do
    $char = substring(s, k, k+1)$
    if $(char =$ “ ”$)$ then
      if $(word \neq$ “”$)$ then $result := result \parallel \langle\, word\,\rangle$ end
      $word :=$ “”
    else
      $word := word \parallel char$
    end
  end
  if $(word \neq$ “”$)$ then $result := result \parallel \langle\, word\,\rangle$ end
  $result$
end
```

Automatic Construction of Parallel Code

If you can construct *two* sequential versions of a function that is a homomorphism on lists, one that operates left-to-right and one right-to-left, then there is a technique for constructing a parallel version automatically.

> Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. "Automatic inversion generates divide-and-conquer parallel programs." Proc. 2007 ACM SIGPLAN PLDI, 146-155.

Just derive a weak right inverse function and then apply the Third Homomorphism Theorem. See—it's easy!

There is an analogous result for tree homomorphisms. Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. "The third homomorphism theorem on trees: Downward and upward lead to divide-and-conquer." Proc. 2009 ACM SIGPLAN-SIGACT POPL, 177–185.

Full disclosure: the authors of these papers were members of a research group at the University of Tokyo that has had a collaborative research agreement with the Programming Language Research group at Sun Microsystems Laboratories.

We Need a New Mindset for Multicores

- 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! Ultimately, it's all trees.
- As soon as you say "first, `SUM = 0`" you are hosed. Accumulators are BAD for parallelism. Note that `foldl` and `foldr`, though functional, are fundamentally accumulative.
- 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 parallel strategies for problem decomposition, data structure design, and algorithmic organization:
  - The top-down view:
    Don't split a problem into "the first" and "the rest."
    Instead, split a problem into roughly equal pieces; recursively solve subproblems, then combine subsolutions.
  - The bottom-up view:
    Don't create a null solution, then successively update it;
    Instead, map inputs independently to singleton solutions, then merge the subsolutions treewise.
  - Combining subsolutions is usually trickier than incremental update of a single solution.

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.

The Fully Engineered Story

In practice, there are many optimizations:

- Optimized representations of singleton lists.
- Use tree branching factors larger than 2. (Example: Rich Hickey's Clojure is a JVM^{TM}-based Lisp that represents lists as 64-ary trees.)
- Use self-balancing trees (2-3, red-black, finger trees, . . .).
- Use sequential techniques near the leaves.
- Have arrays at the leaves. Decide dynamically whether to process them sequentially or by parallel recursive subdivision.
- When iterating over an integer range, decide dynamically whether to process it sequentially or by parallel recursive subdivision.

Linear lists must be processed sequentially. A tree can be processed breadth-first (parallel) or depth-first (sequential), or both. Therefore we should use both parallel and sequential strategies, and perhaps derive one from the other.

Conclusion

- Programs and data structures organized according to linear problem decomposition principles can be hard to parallelize.
- Programs and data structures organized according to parallel problem decomposition principles are easily processed either in parallel or sequentially, according to available resources.
- This parallel 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.

Get rid of `cons`!

guy.steele@sun.com
http://research.sun.com/projects/plrg
http://projectfortress.sun.com