

# A Growable Language

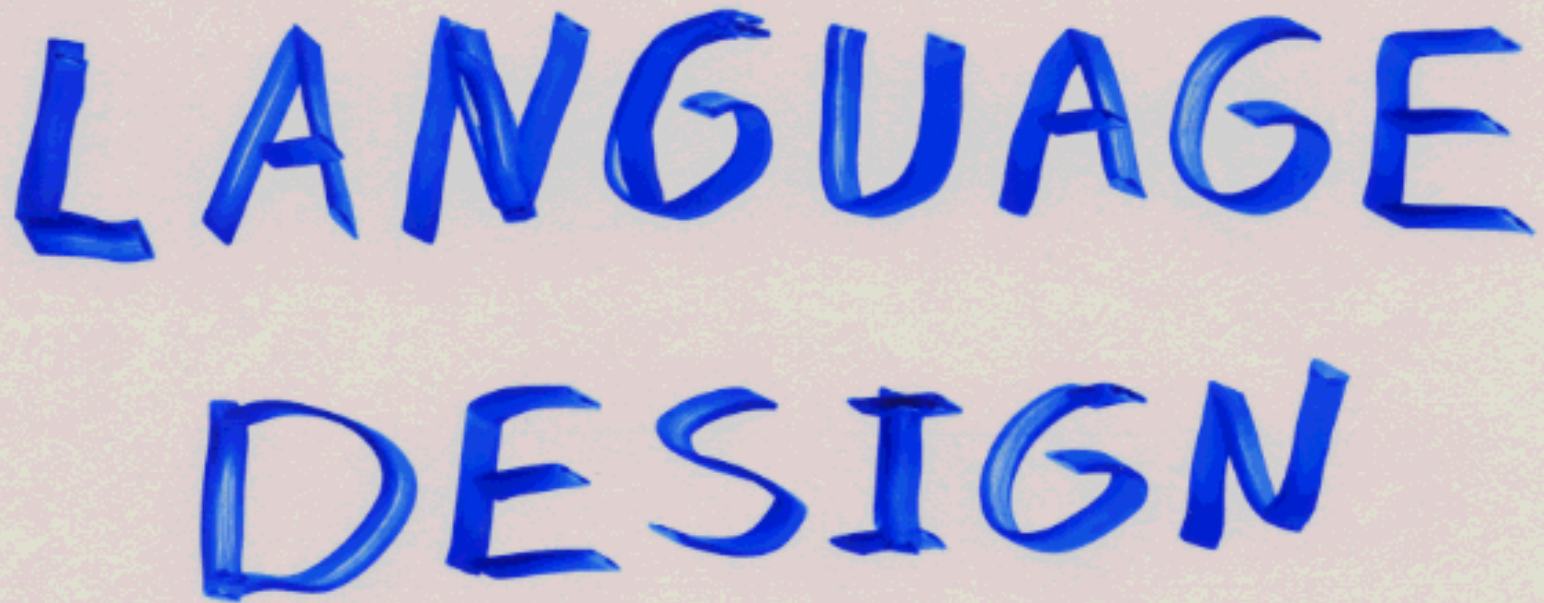
**Guy Steele**

Sun Microsystems Laboratories

October 2006

Copyright © 2006 Sun Microsystems, Inc. ("Sun"). All rights are reserved by Sun except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Sun.



A photograph of a whiteboard with the words "LANGUAGE DESIGN" written in blue marker. The letters are thick and have a 3D, extruded appearance, giving them a tactile, physical quality. The text is arranged in two lines: "LANGUAGE" on the top line and "DESIGN" on the bottom line. The background of the whiteboard is slightly textured and has some faint, illegible markings.

# Fortress: “To Do for Fortran What Java™ Did for C”

- Catch “stupid mistakes” (like array bounds errors)
- Extensive libraries (e.g., for network environment)
- Security model (including type safety)
- Dynamic compilation
- Platform independence
- Multithreading
  
- Make programmers more productive

# The Context of the Research

- **Improving programmer productivity** for scientific and engineering applications
- Research funded in part by the US DARPA IPTO (Defense Advanced Research Projects Agency Information Processing Technology Office) through their **High Productivity Computing Systems** program
- Goal is economically viable technologies for both government and industrial applications by the year **2010 and beyond**
- Targeted to a range of hardware, from petaflop supercomputers to single/multiple multicore chips

# Key Ideas

- Don't build the language—grow it
- Make programming notation closer to math
- Ease use of parallelism



# Growing a Language

- Languages have gotten much bigger
- You can't build one all at once
- Therefore it must grow over time
- What happens if you design it to grow?
- How does the need to grow affect the design?
- Need to grow a user community, too

See Steele, “Growing a Language” keynote talk, OOPSLA 1998;  
*Higher-Order and Symbolic Computation* **12**, 221–236 (1999)

# What Primitive Data Types to Include?

- Integers and floating-point (what sizes? bignums?)
- Complex numbers, rational numbers, intervals
- Conway-style “games”??
- Arrays, vectors, and matrices
- Rational intervals, complex intervals
- Complex vectors and matrices
- Quaternions and octonions?
- What about physical units (meters, kilograms)?

**“I might say ‘yes’ to *each* one of these, but it is clear that I *must* say ‘no’ to *all* of them!”**

# Interesting Language Design Strategy

Wherever possible,  
consider whether a proposed language feature  
can be provided by a library  
rather than having it built into the compiler.

To make this work, library designers need  
substantial control over syntax and semantics—  
not just the ability to code new functions or methods  
invoked by a single method call syntax `foo(a, b, c)`.

# Minimalist Approach

- As few primitive types as possible (cf. Bacon's Kava)
  - > Binary words of many different sizes
  - > Linear sequences (fixed length known at compile time)
  - > Heap sequences (fixed length known at allocation time)
- User-defined parameterized types
  - > Objects with methods and fields
  - > Value objects (like records—need not be heap-allocated)
- User-defined polymorphic (overloaded) operators
- Aggressive type inference to reduce clutter
  - > Many variables require no type declarations
- Aggressive static and dynamic optimization

# Types Defined by Libraries

- Lists, vectors, sets, multisets, and maps
  - > Like C Standard Template Library, but better notation

$$\langle 1, 2, 4, 3, 4 \rangle \quad A \cup \{1, 2, 3, 4\}$$

$$[3 \ 4 \ 5] \times [1 \ 0 \ 0]$$

- Matrices and multidimensional arrays
- Integers, floats, rationals, **with physical units**

$m$ :  $\mathbb{R}$  Mass = 3.7 kg

$\mathbf{v}$ :  $\mathbb{R}^3$  Velocity = [3.5 0 1] m/s

$\mathbf{p}$ :  $\mathbb{R}^3$  Momentum =  $m \mathbf{v}$

- Data structures may be local or distributed

# ASCII (“Wiki-like markup”) Notation

- Lists, vectors, sets, multisets, and maps
    - > Like C Standard Template Library, but better notation
- <|1, 2, 3, 4|>            A UNION {1, 2, 3, 4}**  
**[3 4 5] CROSS [1 0 0]**
- Matrices and multidimensional arrays
  - Integers, floats, rationals, with physical units
    - m: RR Mass = 3.7 kg\_**
    - \_v: RR^3 Velocity = [3.5 0 1] m\_/s\_**
    - \_p: RR^3 Momentum = m \_v**
  - Data structures may be local or distributed

# A Growable, Open Language

- Old language design model:
  - > Study applications
  - > Add language features to improve application coding
- Our new model:
  - > Study applications
  - > Study how a library can improve application coding
  - > Add language features to improve library coding
- Conjectures:
  - > Better leverage, leading to more rapid improvement
  - > Enables experimentation with open-source strategies



# Replaceable Components

- Avoid a monolithic “Standard Library”
- Replaceable components with version control
- Encourage alternate implementations
  - > Performance choices
  - > Test them against each other
- Encourage experimentation
  - > Framework for alternate language designs

# Making Abstraction Efficient

- We assume implementation technology that makes aggressive use of runtime performance measurement and optimization
  - > Frequently *more* efficient than static optimization
- Repeat the success of the Java™ Virtual Machine
- Goal: programmers (especially library writers) need never fear subroutines, functions, methods, and interfaces for performance reasons
- This may take years, but our target is the year 2010 and beyond

# Type System: Objects and Traits

- Traits: like interfaces, but may contain code
  - > Based on work by Schärli, Ducasse, Nierstrasz, Black, et al.
- Multiple inheritance of code (but not fields)
  - > Objects with fields are the leaves of the hierarchy
- Multiple inheritance of contracts and tests
  - > Automated unit testing
- Traits and methods may be parameterized
  - > Parameters may be types or compile-time constants
- Primitive types are first-class
  - > Booleans, integers, floats, characters are all objects

# Features of Trait Declarations

- **where** clauses introduce and constrain type variables

```
trait List[X] extends { List[Y] }  
  where { X extends Y, Y extends Object }
```

- **excludes** clauses declare traits to be disjoint

```
trait String excludes { Number, Thread } ...
```

- **comprises** clauses list all immediate subtypes

```
trait Boolean comprises { true, false } ...
```

These express interesting relationships among types that are useful for optimizing code.

# Sample Code: Algebraic Constraints

```
trait BinaryPredicate[[T extends BinaryPredicate[[T, ~]], opr ~]]
  opr ~(self, other: T): Boolean
end

trait Symmetric[[T extends Symmetric[[T, ~]], opr ~]]
  extends { BinaryPredicate[[T, ~]] }
  property  $\forall(a: T, b: T) (a \sim b) \leftrightarrow (b \sim a)$ 
end

trait EquivalenceRelation[[T extends EquivalenceRelation[[T, ~]], opr ~]]
  extends { Reflexive[[T, ~]], Symmetric[[T, ~]], Transitive[[T, ~]] }
end

trait Integer extends { CommutativeRing[[Integer, +, -, ·, zero, one]],
  TotalOrderOperators[[Integer, <, ≤, ≥, >, CMP]],
  ... }
  ...
end
```

(This is actual Fortress library code.)

# Data and Control Models

- Data model: shared global address space
- Control model: multithreaded
  - > Basic primitive is “spawn”
  - > We hope application code seldom uses it
- Declared distribution of data and threads
  - > Managing aggregates integrated into type system
  - > Policies programmed as libraries, not wired in
- Transactional access to shared variables
  - > Atomic blocks (implicit or explicit retry)
  - > Lock-free (no blocking, no deadlock)

# Conventional Mathematical Notation

- The language of mathematics is centuries old, concise, convenient, and widely taught
- Early programming languages were constrained by keyboards and printers
- Experiments in the 1960s were not portable
- Now we have bitmap displays and Unicode
- Still, parsing mathematical notation is a challenge
  - > Subtle reliance on whitespace:  $\{ |x| \mid x \leftarrow S, 3 \mid x \}$
  - > Semantic conventions:  $y = 3 x \sin x \cos 2 x \log \log x$
- Programming language tradition has contributions
  - > Type theory, block structure, variable scope

# What Syntax is Actually Wired In?

- Parentheses ( ) for grouping
  - Comma , to separate expressions in tuples
  - Semicolon ; to separate statements on a line
  - Dot . for field and method selection
  - Conservative, traditional rules of precedence
    - > A dag, not always transitive (examples:  $A+B>C$  is okay; so is  $B>C \vee D>E$ ; but  $A+B \vee C$  needs parentheses)
- 
- Juxtaposition is a binary operator
- 
- Any other operator can be infix, prefix, and/or postfix
  - Many sets of brackets

# Libraries Define . . .

- Which operators have infix, prefix, postfix definitions, and what types they apply to

**opr**  $-(m:\mathbb{Z}, n:\mathbb{Z}) = m.\text{subtract}(n)$

**opr**  $-(m:\mathbb{Z}) = m.\text{negate}()$

**opr**  $(n:\mathbb{N})! = \text{if } n=0 \text{ then } 1 \text{ else } n \cdot (n-1)! \text{ end}$

- Whether a juxtaposition is meaningful

**opr**  $\text{juxtaposition}(m:\mathbb{Z}, n:\mathbb{Z}) = m.\text{times}(n)$

- What bracketing operators actually mean

**opr**  $\lceil x:\mathbb{R} \rceil = \text{ceiling}(x)$

**opr**  $|x:\mathbb{R}| = \text{if } x < 0 \text{ then } -x \text{ else } x \text{ end}$

**opr**  $|s:\text{Set}| = s.\text{size}$

# But Wasn't Operator Overloading a Disaster in C++ ?

- Yes, it was
  - > Not enough operators to go around
  - > Failure to stick to traditional meanings
- We have also been tempted and had to resist
- We believe Unicode + discipline can avert disaster
- We see benefits in using notations for programming that are also used for specification

# Simple Example: NAS CG Kernel (ASCII)

```

conjGrad(A: Matrix[\Float\], x: Vector[\Float\]):
  (Vector[\Float\], Float)
  cgit_max = 25
  z: Vector[\Float\] := 0
  r: Vector[\Float\] := x
  p: Vector[\Float\] := r
  rho: Float := r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)

```

```
(z, norm) = conjGrad(A, x)
```

Matrix[\T\] and Vector[\T\] are parameterized interfaces, where T is the type of the elements.

The form  $x:T:=e$  declares a variable  $x$  of type  $T$  with initial value  $e$ , and that variable may be updated using the assignment operator  $:=$ .

# Simple Example: NAS CG Kernel (ASCII)

```

conjGrad[\Elt extends Number, nat N,
         Mat extends Matrix[\Elt,N BY N\],
         Vec extends Vector[\Elt,N\]
        \](A: Mat, x: Vec): (Vec, EIt)
cgit_max = 25
z: Vec := 0
r: Vec := x
p: Vec := r
rho: EIt := r^T r
for j <- seq(1:cgit_max) do
  q = A p
  alpha = rho / p^T q
  z := z + alpha p
  r := r - alpha q
  rho0 = rho
  rho := r^T r
  beta = rho / rho0
  p := r + beta p
end
(z, ||x - A z||)

```

Here we make conjGrad a generic procedure. The runtime compiler may produce multiple instantiations of the code for various types E.

The form  $x=e$  as a statement declares variable  $x$  to have an unchanging value. The type of  $x$  is exactly the type of the expression  $e$ .

```
(z,norm) = conjGrad(A,x)
```

# Simple Example: NAS CG Kernel (Unicode)

```

conjGrad[[Elt extends Number, nat N,
          Mat extends Matrix[[Elt, N×N]],
          Vec extends Vector
          ]](A: Mat, x: Vec): (Vec, Elt)
  cgit_max = 25
  z: Vec := 0
  r: Vec := x
  p: Vec := r
  ρ: Elt := r^T r
  for j ← seq(1:cgit_max) do
    q = A p
    α = ρ / p^T q
    z := z + α p
    r := r - α q
    ρ₀ = ρ
    ρ := r^T r
    β = ρ / ρ₀
    p := r + β p
  end
  (z, ||x - A z||)

```

This would be considered entirely equivalent to the previous version. You might think of this as an abbreviated form of the ASCII version, or you might think of the ASCII version as a way to conveniently enter this version on a standard keyboard.

# Simple Example: NAS CG Kernel

```

conjGrad || Elt extends Number, nat N,
             Mat extends Matrix || Elt,  $N \times N$  ||,
             Vec extends Vector || Elt, N ||
             || (A : Mat, x : Vec) : (Vec, Elt)

```

```

cgmax = 25

```

```

z : Vec := 0

```

```

r : Vec := x

```

```

p : Vec := r

```

```

ρ : Elt :=  $r^T r$ 

```

```

for j ← seq(1 : cgmax) do

```

```

  q = A p

```

$$\alpha = \frac{\rho}{p^T q}$$

```

  z := z + α p

```

```

  r := r - α q

```

```

  ρ0 = ρ

```

```

  ρ :=  $r^T r$ 

```

$$\beta = \frac{\rho}{\rho_0}$$

```

  p := r + β p

```

```

end

```

```

(z, ||x - A z||)

```

It's not new or surprising that code written in a programming language might be displayed in a conventional math-like format. The point of this example is how similar the code is to the math notation: the gap between the two syntaxes is relatively small. We want to see what will happen if a principal goal of a new language design is to minimize this gap.

# Comparison: NAS NPB 1 Specification

```

z = 0
r = x
ρ = rT r
p = r
DO i = 1, 25
    q = A p
    α = ρ / (pT q)
    z = z + α p
    ρ0 = ρ
    r = r - α q
    ρ = rT r
    β = ρ / ρ0
    p = r + β p
ENDDO
compute residual norm explicitly: ||r|| = ||x - A z||
  
```

```

z : Vec := 0
r : Vec := x
p : Vec := r
ρ : Elt := rT r
for j ← seq(1 : cgitmax) do
    q = A p
    α =  $\frac{\rho}{p^T q}$ 
    z := z + α p
    r := r - α q
    ρ0 = ρ
    ρ := rT r
    β =  $\frac{\rho}{\rho_0}$ 
    p := r + β p
end
(z, ||x - A z||)
  
```

# Comparison: NAS NPB 2.3 Serial Code

```

do j=1,naa+1
  q(j) = 0.0d0
  z(j) = 0.0d0
  r(j) = x(j)
  p(j) = r(j)
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
  do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
    enddo
    w(j) = sum
  enddo
  do j=1,lastcol-firstcol+1
    q(j) = w(j)
  enddo
enddo

```

```

do j=1,lastcol-firstcol+1
  w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + p(j)*q(j)
enddo
d = sum
alpha = rho / d
rho0 = rho
do j=1,lastcol-firstcol+1
  z(j) = z(j) + alpha*p(j)
  r(j) = r(j) - alpha*q(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  sum = sum + r(j)*r(j)
enddo
rho = sum
beta = rho / rho0
do j=1,lastcol-firstcol+1
  p(j) = r(j) + beta*p(j)
enddo
enddo

```

```

do j=1,lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j),rowstr(j+1)-1
    sum = sum + a(k)*z(colidx(k))
  enddo
  w(j) = sum
enddo
do j=1,lastcol-firstcol+1
  r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
  d = x(j) - r(j)
  sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )

```

# Parallelism Is Not a Feature!

- Parallel programming is not a goal, but a pragmatic compromise.
- It would be a lot easier to program a single processor chip running at 1 PHz than a million processors running at 20 GHz.
  - > We don't know how to build a 1 PHz processor.
  - > Even if we did, someone would still want to strap a bunch of them together!
- Parallel programming is difficult and error-prone.

# Questions

Can we encapsulate parallelism in libraries?

Will this separation be effective?

# Should Parallelism Be the Default?

- “Loop” can be a misleading term
  - > A set of executions of a parameterized block of code
  - > Whether to order or parallelize those executions should be a separate question
- Fortress “loops” are parallel by default
  - > This is actually a library convention about generators
  - > You get sequential execution by asking for it specifically

# In Fortress, Parallelism Is the Default

```
for i←1:m, j←1:n do
  a[i,j] := b[i] c[j]
end
```

**1:n** is a generator

```
for i←seq(1:m) do
  for j←seq(1:n) do
    print a[i,j]
  end
end
```

**seq(1:n)** is a sequential generator

```
for i←1:m, j←i:n do
  a[i,j] := b[i] c[j]
end
```

**a.indices** is a generator for the indices of the array **a**

**a.indices.rowMajor** is a sequential generator of indices

```
for (i,j)←a.indices do a[i,j] := b[i] c[j] end
```

```
for (i,j)←a.indices.rowMajor do print a[i,j] end
```

- Generators (defined by libraries) manage parallelism and the assignment of threads to processors

# Generators and Reducers

$$y = \Sigma[k \leftarrow 1:n] a[k] x^k$$

$$z = \text{MAX}[(j,k) \leftarrow a.\text{indices}] |a[j,k] - b[j,k]|$$

$$y = \sum_{k \leftarrow 1:n} a_k x^k$$

$$z = \text{MAX}_{(j,k) \leftarrow a.\text{indices}} |a_{j,k} - b_{j,k}|$$

- Reducers (also defined by libraries) such as  $\Sigma$  (or **SUM**) and **MAX** may have serial/parallel implementations
- Reducers are driven by generators
- Distribution of generator guides parallelism of reducer

# Kinds of Generators

- Aggregates
  - > Lists `<1,2,4,3,4>` and vectors `[1 2 4 3 4]`
  - > Sets `{1,2,3,4}` and multisets `{|1,2,3,4,4|}`
  - > Arrays (including multidimensional)
- Ranges `1:10` and `1:99:2` and `5#20`
- Index sets `a.indices` and `a.indices.rowMajor`
- Index-value sets of maps `ht.keyValuePairs`
- Functions compute new generators from old ones
  - > `seq` computes sequential version of a generator
  - > `zip` takes two generators, returns a generator of pairs

# Loops, Reducers, Comprehensions

**for**  $k \leftarrow 1:n$  **do** *print k* **end**

$$y = \sum_{k \leftarrow 1:n} a_k x^k$$

$$w = \sum S \quad (* \text{ same as } \sum_{x \leftarrow S} x *)$$

$$v = \bigcap_{\substack{k \leftarrow S \\ \text{prime } k}} \text{arrayOfSets}_k$$

$$z = \text{MAX}_{(j,k) \leftarrow a.\text{indices}} |a_{j,k} - b_{j,k}|$$

$$B = \{ f(x, y) \mid x \leftarrow S, y \leftarrow A, x \neq y \}$$

$$l_{\text{triangle}} = \left\langle \frac{x(x+1)}{2} \mid x \leftarrow 1:100 \right\rangle$$

# Loops, Reducers, Comprehensions

for  $k \leftarrow 1:n$  do print i end

$y = \Sigma[\underline{k \leftarrow 1:n}] a[k] x^k$

$w = \Sigma S$  (\* same as  $\Sigma[\underline{x \leftarrow S}] x$  \*)

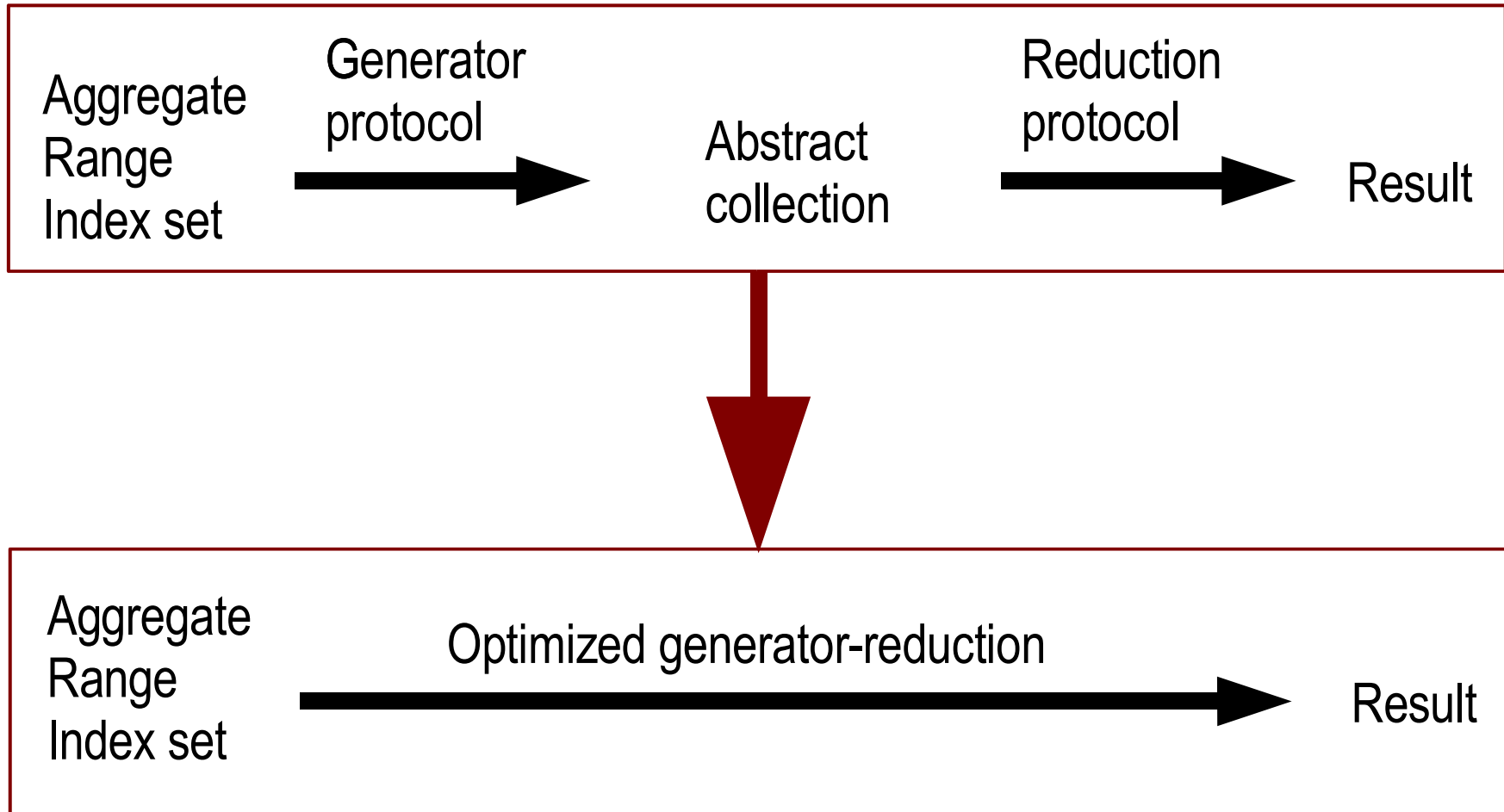
$v = \cap[\underline{k \leftarrow S, \text{prime } k}] \text{arrayOfSets}[k]$

$z = \text{MAX}[\underline{(j,k) \leftarrow a.\text{indices}}] |a[j,k] - b[j,k]|$

$B = \{ f(x,y) \mid \underline{x \leftarrow S, y \leftarrow A, x \neq y} \}$

$1\_triangle = \langle x(x+1)/2 \mid \underline{x \leftarrow 1:100} \rangle$

# Abstract Collections

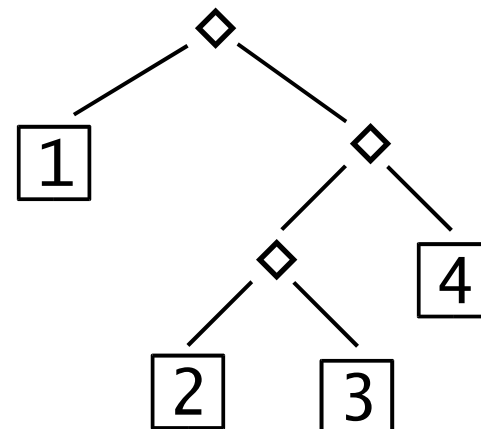
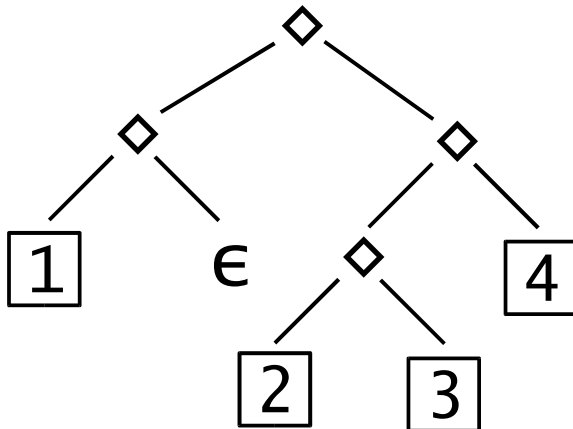


# Representation of Abstract Collections

Binary operator  $\diamond$

Leaf operator (“unit”)  $\square$

Optional empty collection (“zero”)  $\epsilon$   
that is the identity for  $\diamond$

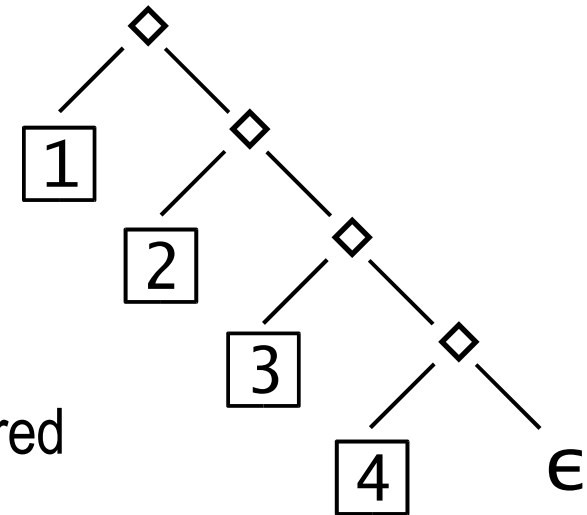
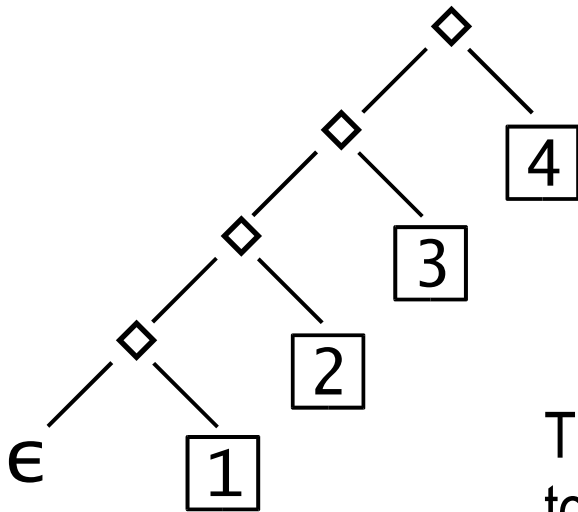
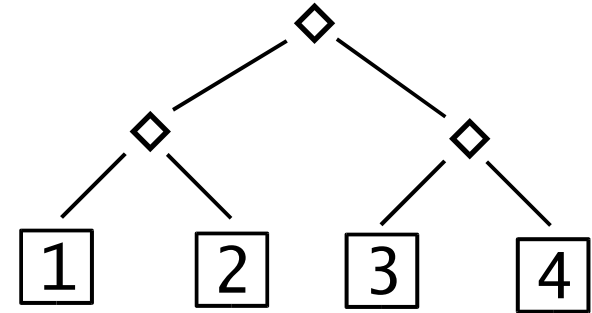
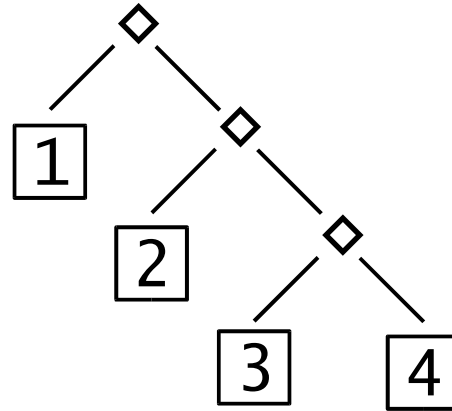
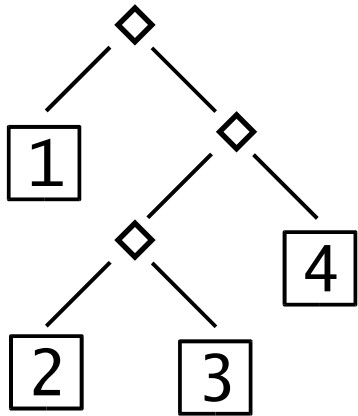


# Algebraic Properties of $\diamond$

Associative	Commutative	Idempotent	
no	no	no	binary trees
no	no	yes	weird
no	yes	no	mobiles
no	yes	yes	weird
yes	no	no	lists
yes	no	yes	weird
yes	yes	no	multisets
yes	yes	yes	sets

The “Boom hierarchy”

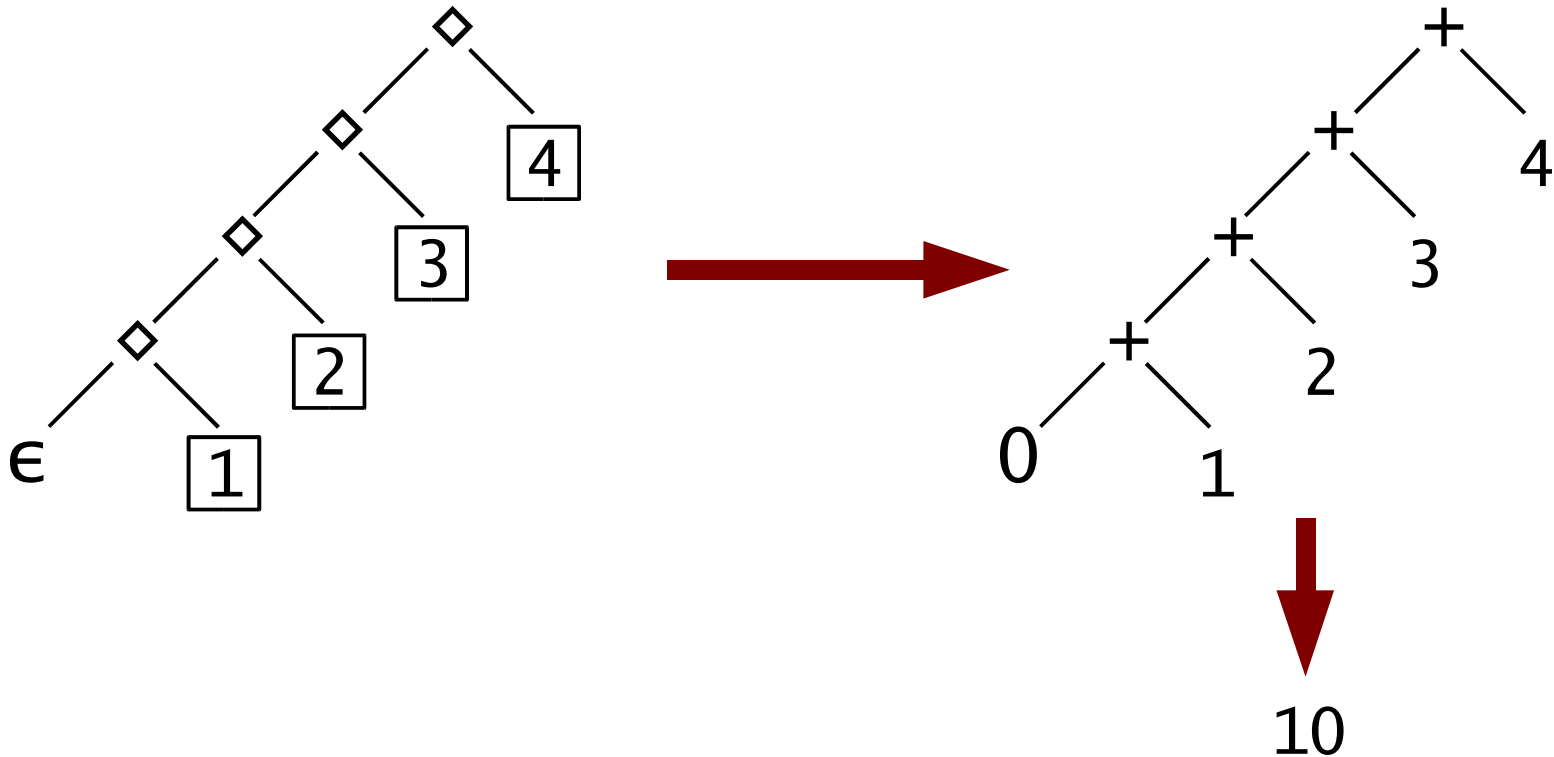
# Associativity



These are all considered to be equivalent.

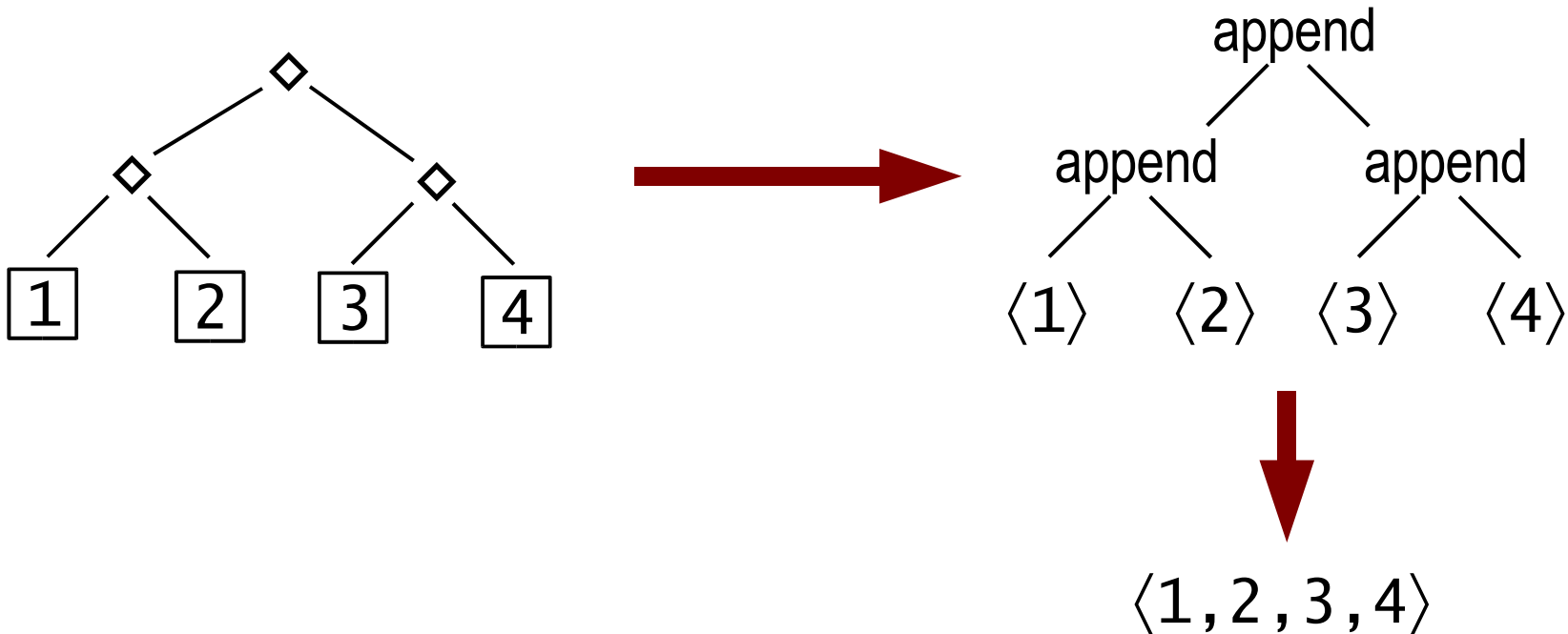
# Catamorphism: Summation

Replace  $\diamond$   $\square$   $\epsilon$  with  $+$  identity  $0$



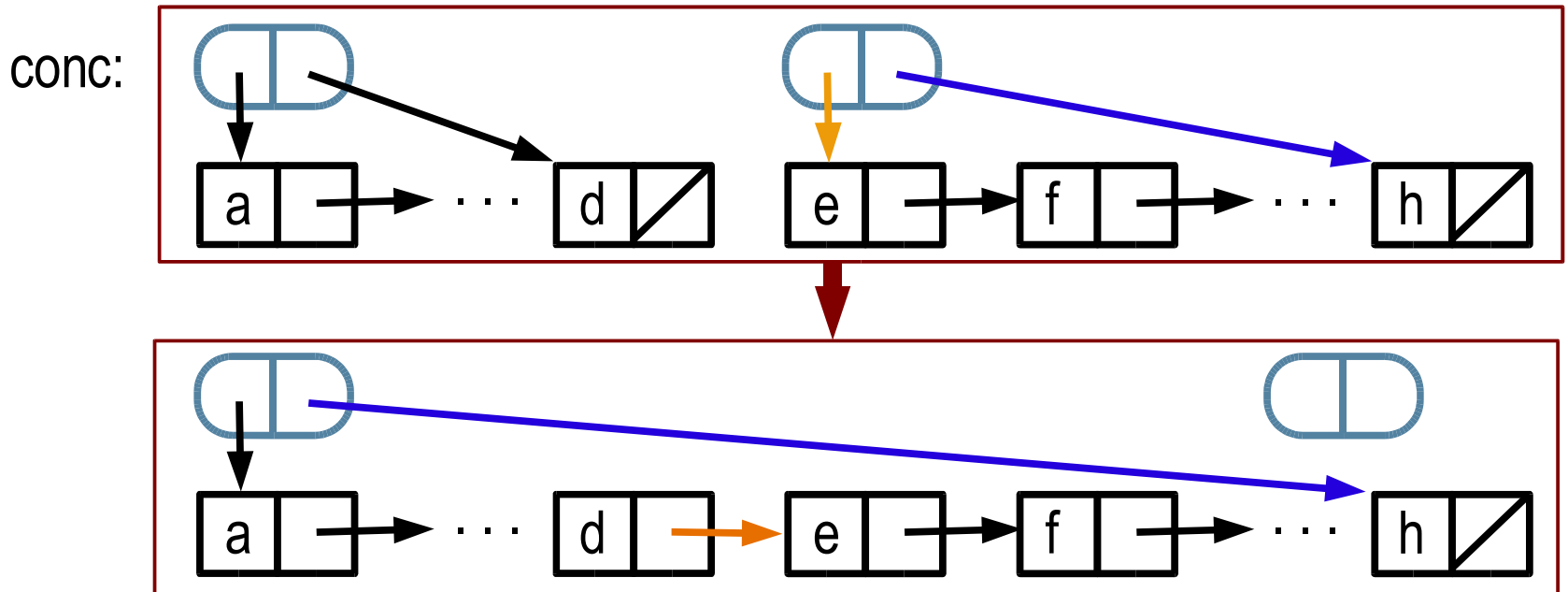
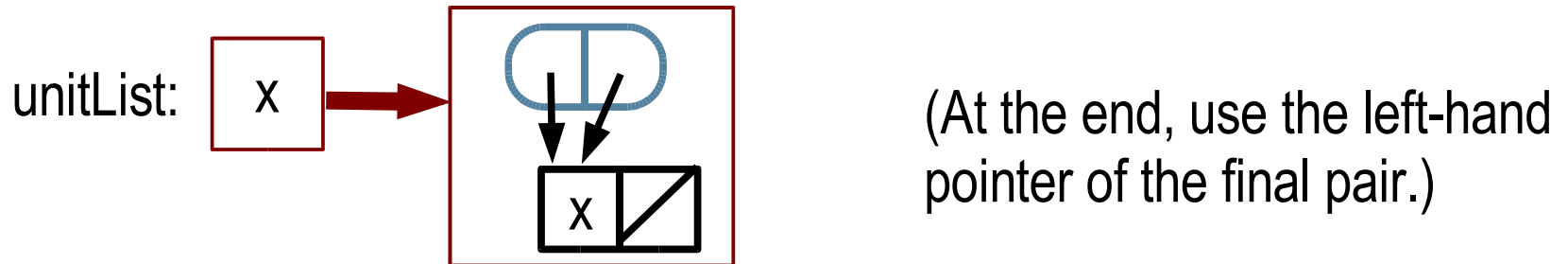
# Catamorphism: Lists

Replace  $\diamond$   $\square$   $\epsilon$  with `append`  $\langle - \rangle$   $\langle \rangle$



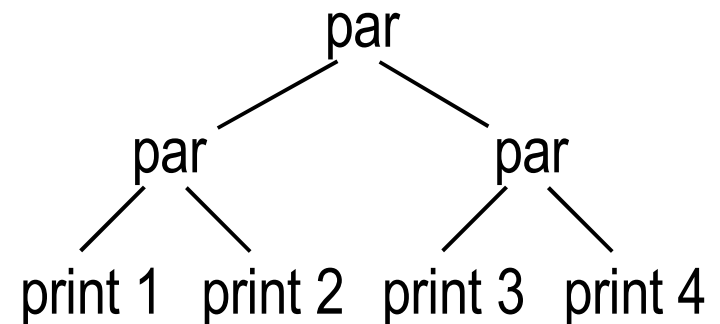
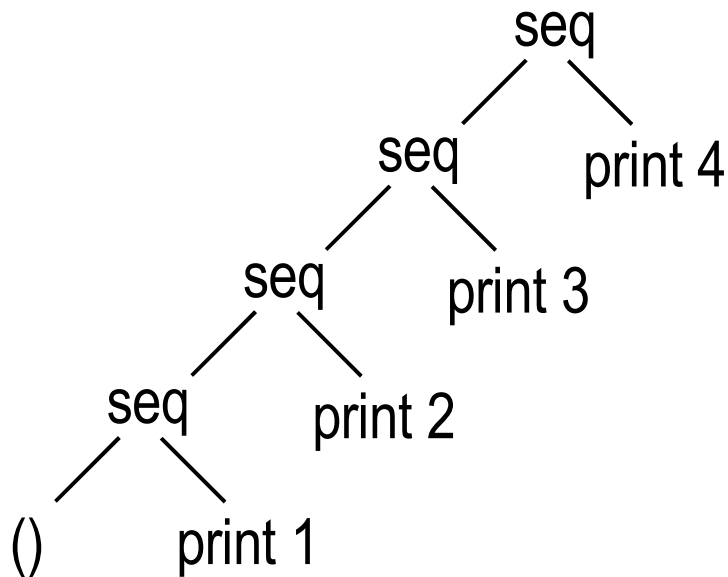
# Catamorphism: Splicing Linked Lists

Replace  $\diamond$   $\square$   $\epsilon$  with `conc` `unitList` `nil`



# Catamorphism: Loops

Replace  $\diamond$   $\square$   $\in$  with seq identity  $()$  or par identity  $()$   
 where  $\text{seq}: (), () \rightarrow ()$  and  $\text{par}: (), () \rightarrow ()$



# Desugaring

$\Sigma[i \leftarrow a, j \leftarrow b, p, k \leftarrow c] e$	becomes	$\Sigma(f)$
$\langle e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c \rangle$	becomes	$\text{List}(f)$
<b>for</b> $i \leftarrow a, j \leftarrow b, p, k \leftarrow c$ <b>do</b> $e$ <b>end</b>	becomes	<b>For</b> ( $f$ )

where  $f =$

```
(fn (r)=>
  (a).generate(r, fn (i)=>
    (b).generate(r, fn (j)=>
      (p).generate(r, fn ()=>
        (c).generate(r, fn (k)=>
          r.unit(e))))))
```

Note: **generate** method can be overloaded!

# Implementation

```
opr  $\Sigma$ [[T]](f: Catamorphism[[T,T]] $\rightarrow$ T): T
  where { T extends Monoid[[T,+]] } =
  f(Catamorphism(fn(x,y) $\Rightarrow$  x+y, identity, 0))
```

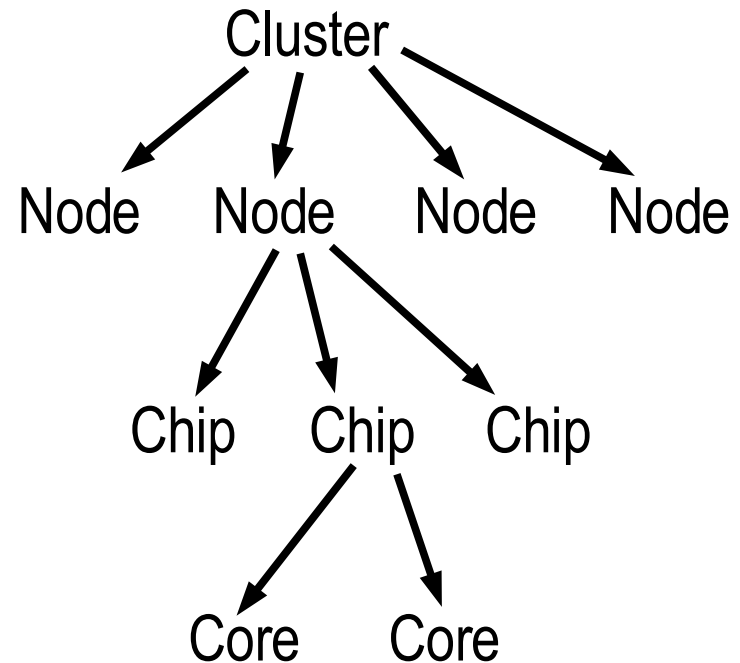
```
List[[T]](f: Catamorphism[[T,List[[T]]]]
   $\rightarrow$ List[[T]]): List[[T]] =
  f(Catamorphism(append, fn(x) $\Rightarrow$  <x>, <>))
```

```
List[[T]](f: Catamorphism[[T,List[[T]]]]
   $\rightarrow$ List[[T]]): List[[T]] =
  f(Catamorphism(conc, unitList, nil)).first
```

```
For(f: Catamorphism[[(),()]] $\rightarrow$ []): () =
  f(Catamorphism(par, identity, ()))
```

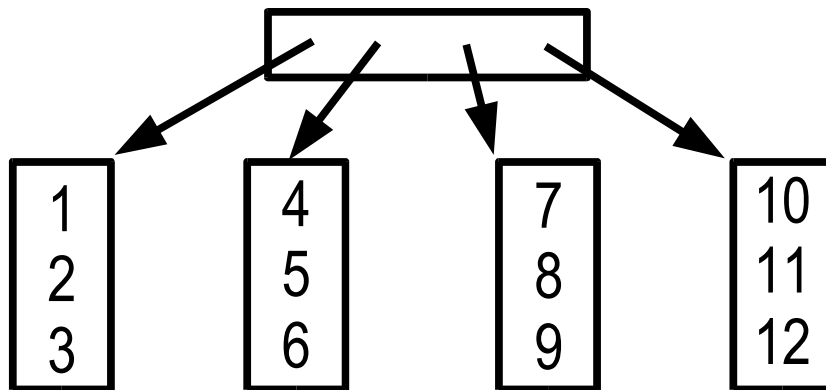
# Regions

- Hierarchical data structure describes CPU and memory resources and their properties
  - > Allocation heaps
  - > Parallelism
  - > Memory coherence
- A running thread can find out its resources
- Threads may be explicitly spawned in specified regions



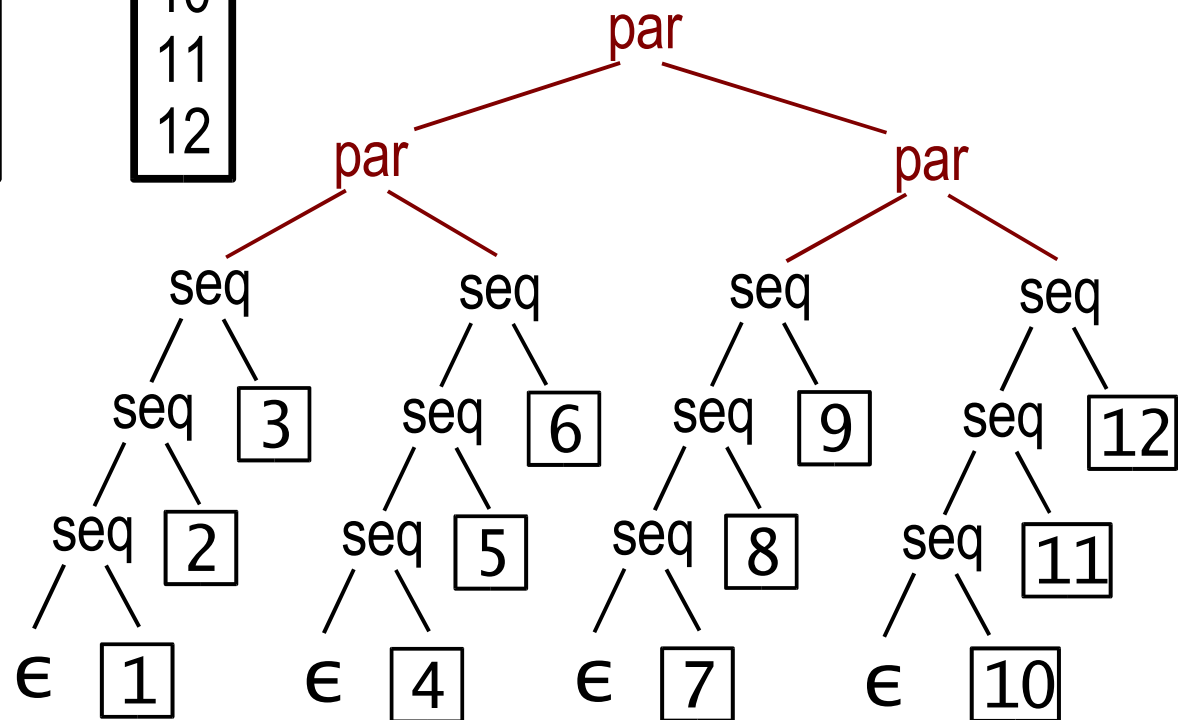
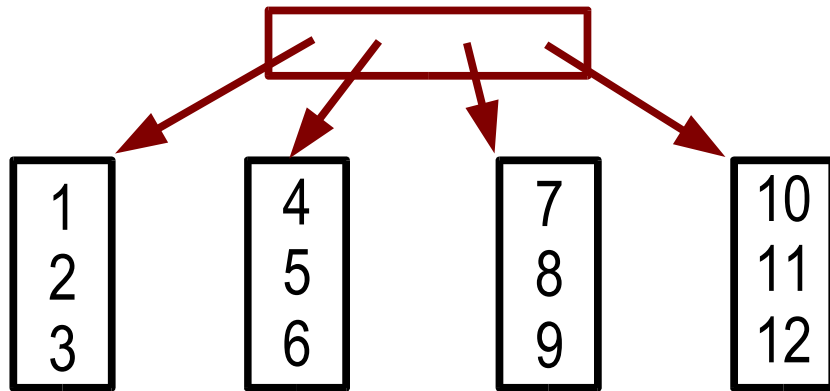
# Distributions

- Describe how to map a data structure onto a region
  - > Block, cyclic, block-cyclic, Morton order ...
  - > Map an array into a chip? Use a local heap.
  - > Map an array onto a cluster? Break it up.



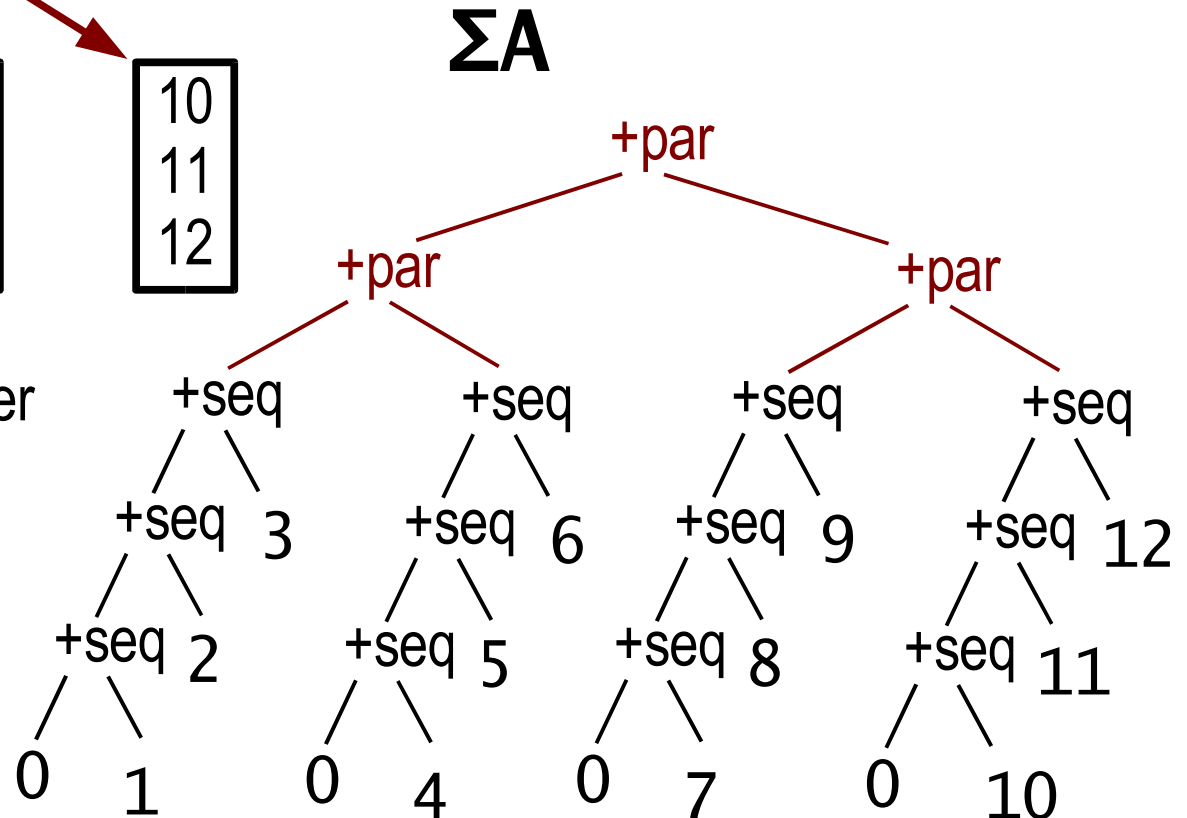
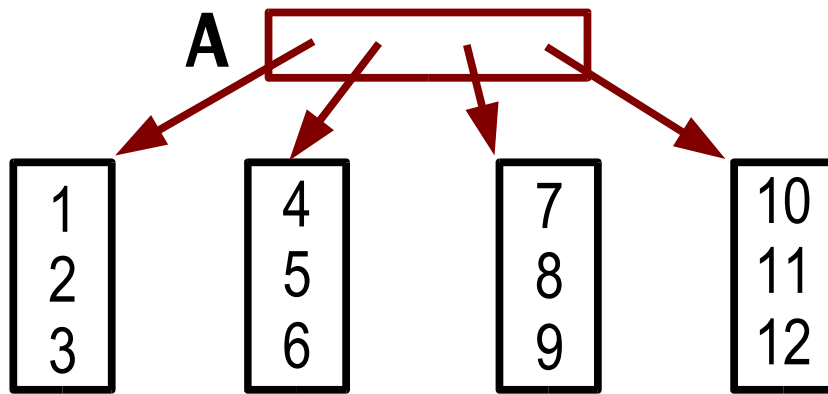
- Defined entirely by libraries!
  - > User-extensible

# Generators Drive Parallelism



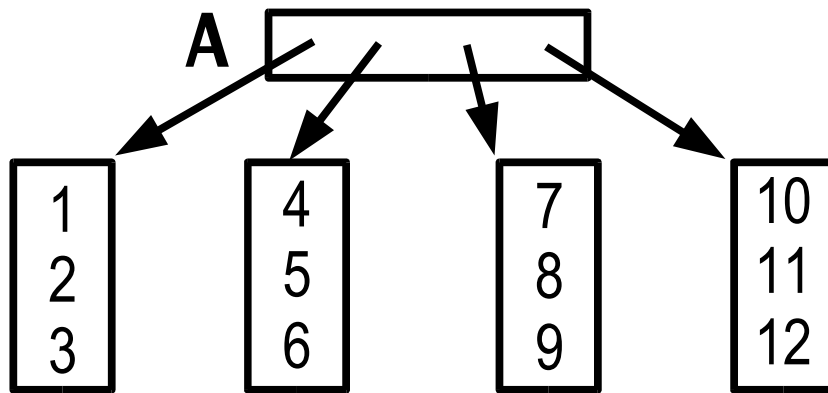
When a data structure (or its index set) is used as a generator, the parallelism of the generator reflects the distribution of the data structure.

# Generators Modify Reducers: Parallelism



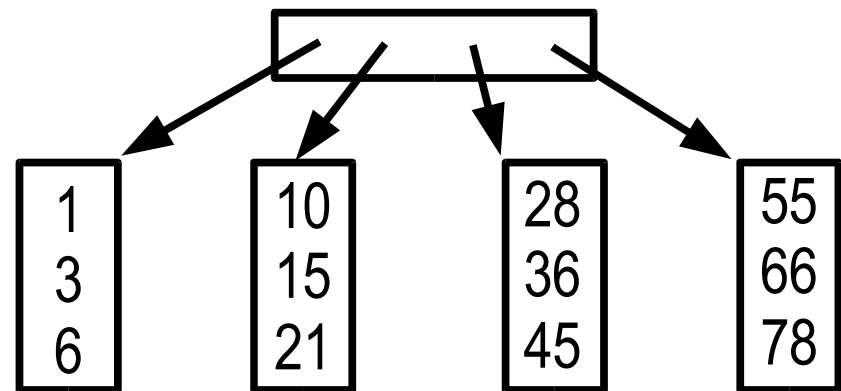
This works because integer addition is associative. The generator knows this because the trait  $\mathbb{Z}$  extends the trait `Associative[ $\mathbb{Z}$ , +]`.

# Generators Modify Reducers: Distribution



$$\left[ \frac{x(x+1)}{2} \mid x \leftarrow A \right]$$

Generators and reducers may agree to use a specialized protocol that, for example, communicates array shapes and distribution information.



# More Desugaring

$[ e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c ]_{dist}$  becomes  $Array(f, dist)$   
 $[ e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c ]^{Constr}$  becomes  $Constr(f)$   
 $[ e \mid i \leftarrow a, j \leftarrow b, p, k \leftarrow c ]_{dist}^{Constr}$  becomes  $Constr(f, dist)$

This lets us specify a distribution explicitly as a subscript, and/or a type constructor/catamorphism as a superscript.

$$[ x(x+1)/2 \mid x \leftarrow 1:n ]_{blockCyclic(4)}$$

$$[ x \mid x > 0 ]^{Maybe}$$

# Example: Lexicographic Comparison

- Assume a binary CMP operator that returns one of Less, Equal, or Greater
- Now consider the binary operator LEXICO:

LEXICO	Less	Equal	Greater
Less	Less	Less	Less
Equal	Less	Equal	Greater
Greater	Greater	Greater	Greater

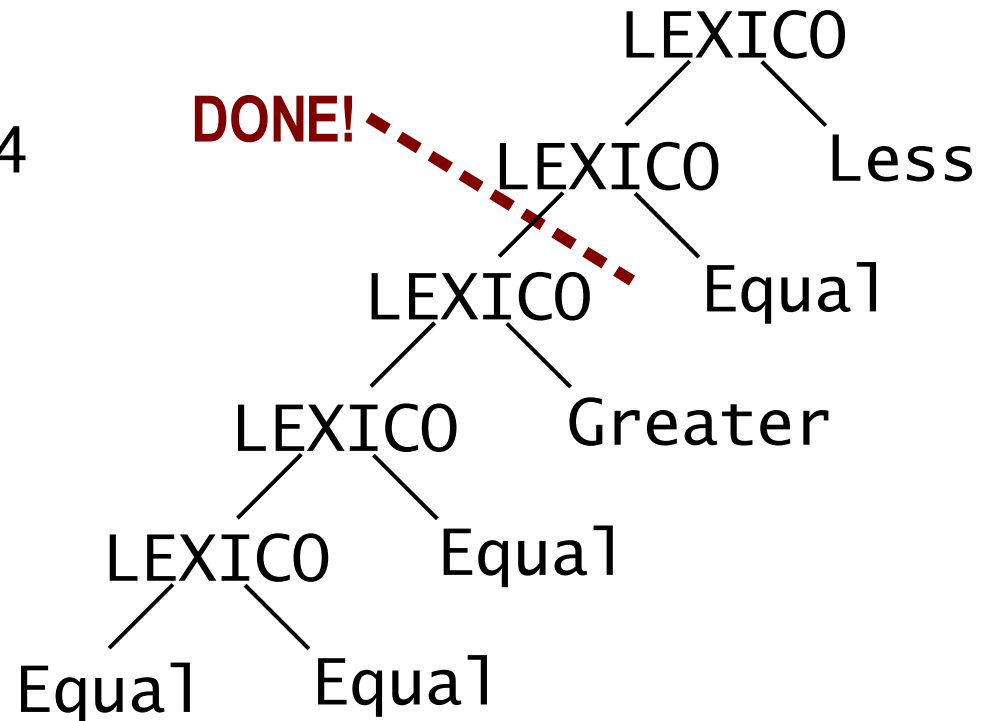
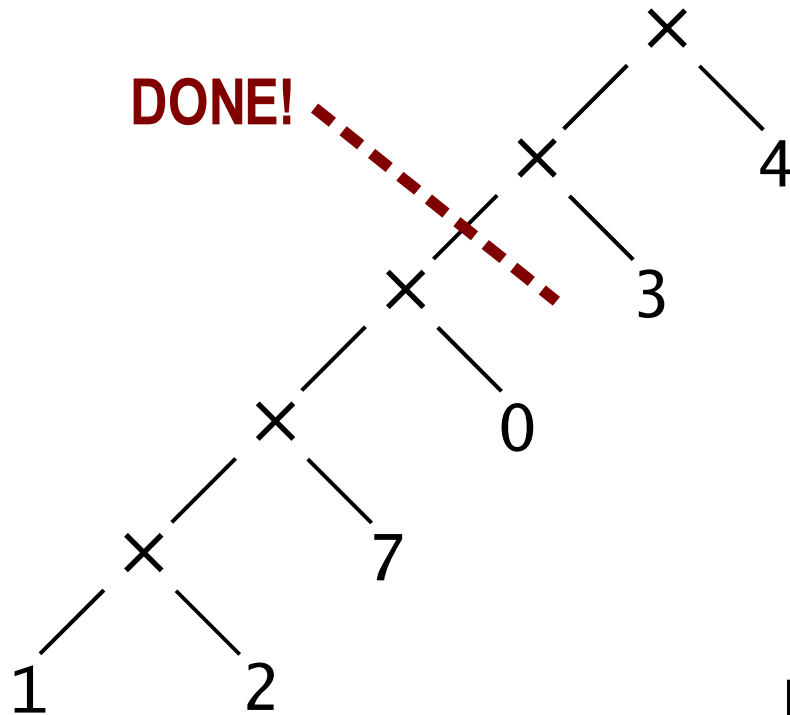
- > Associative (but *not* commutative)
- > Equal is the identity
- > Less and Greater are left zeroes

# Algebraic Properties of LEXICO

```
trait Comparison extends {  
  IdentityEquality[[Comparison]],  
  Associative[[Comparison, LEXICO]],  
  HasRightIdentity[[Comparison, LEXICO, Equal]],  
  HasLeftZeroes[[Comparison, LEXICO]]  
} comprises { Less, Equal, Greater }  
  
...  
test { Less, Equal, Greater }  
end
```

A generator that detects the LEXICO catamorphism (rather, the fact that it has left zeros) can choose to generate special code.

# Zeros Can Stop Iteration Early



# Code for Lexicographic Comparison

```

trait LexOrder[[T,E]]
  extends { TotalOrder[[T,≤,CMP]],
            Indexable[[LexOrder[[T,E]],E]] }
  where { T extends LexOrder[[T,E]],
          E extends TotalOrder[[T,≤,CMP]] }

  opr =(self,other:T):Boolean =
    |self| = |other| AND:
      AND[i←self.indices] self[i]=other[i]

  opr CMP(self,other:T):Comparison = do
    prefix = self.indices ∩ other.indices
    (LEXICO[i←prefix] self[i] CMP other[i]) &
    LEXICO (|self| CMP |other|)
  end

  opr ≤(self,other:T):Boolean =
    (self CMP other) ≠ Greater

end

```

# String Comparison

```
trait String
  extends { LexOrder[[String,Character]], ... }
  ...
  test { "foo", "foobar", "quux", "" }
  property "" < "a" < "foo" < "foobar" < "z"
end
```

# Summary: Parallelism in Fortress

- Regions describe machine resources.
- Distributions map aggregates onto regions.
- Aggregates used as generators drive parallelism.
- Algebraic properties drive implementation strategies.
- Algebraic properties are described by traits.
- Properties are verified by automated unit testing.
- Traits allow sharing of code, properties, and test data.
- Reducers and generators negotiate through overloaded method dispatch keyed by traits to achieve mix-and-match code selection.

# Our Vision

- Powerful libraries make applications simpler
- Big idea is exposing algorithmic and design decisions in libraries rather than burying them in compilers
- Other big idea is burying algorithmic decisions in libraries rather than exposing them in applications
- With key algorithms properly factored and encapsulated in tuned libraries (cf. MATLAB), application code can be concise, therefore easier to check against design specifications

# Our Key Design Themes

- Make stupid mistakes impossible
  - And make clever mistakes relatively unlikely
- Design the language to be grown by (expert) users
  - Rich library language enables simple application languages
- Make abstraction efficient
  - Aggressive static and dynamic optimization
- Make parallelism tractable
  - Appropriate abstractions for managing thread and data distribution
- Emulate standard mathematical notation
  - Reduce the effort of translating from science to computation

# Features for Growth

- Traits with static parameters and where-clauses
- Overloaded functions, methods, and operators (also with static parameters and where-clauses)
- Type inference to reduce type clutter
- Syntactic abstraction with a rich character set
- Reduce syntactic sugar to method and function calls
- Multimethod dynamic dispatch
- Components with API and version management

Building a simple, **array-oriented** scientific language on the foundation of a rich, **object-oriented** framework!

[guy.steele@sun.com](mailto:guy.steele@sun.com)



<http://research.sun.com/projects/plrg>



Carl Eastlund, Guy Steele, Jan-Willem Maessen, Yossi Lev, Eric Allen, Joe Hallett, Sukyoung Ryu, Sam Tobin-Hochstadt, David Chase, João Dias