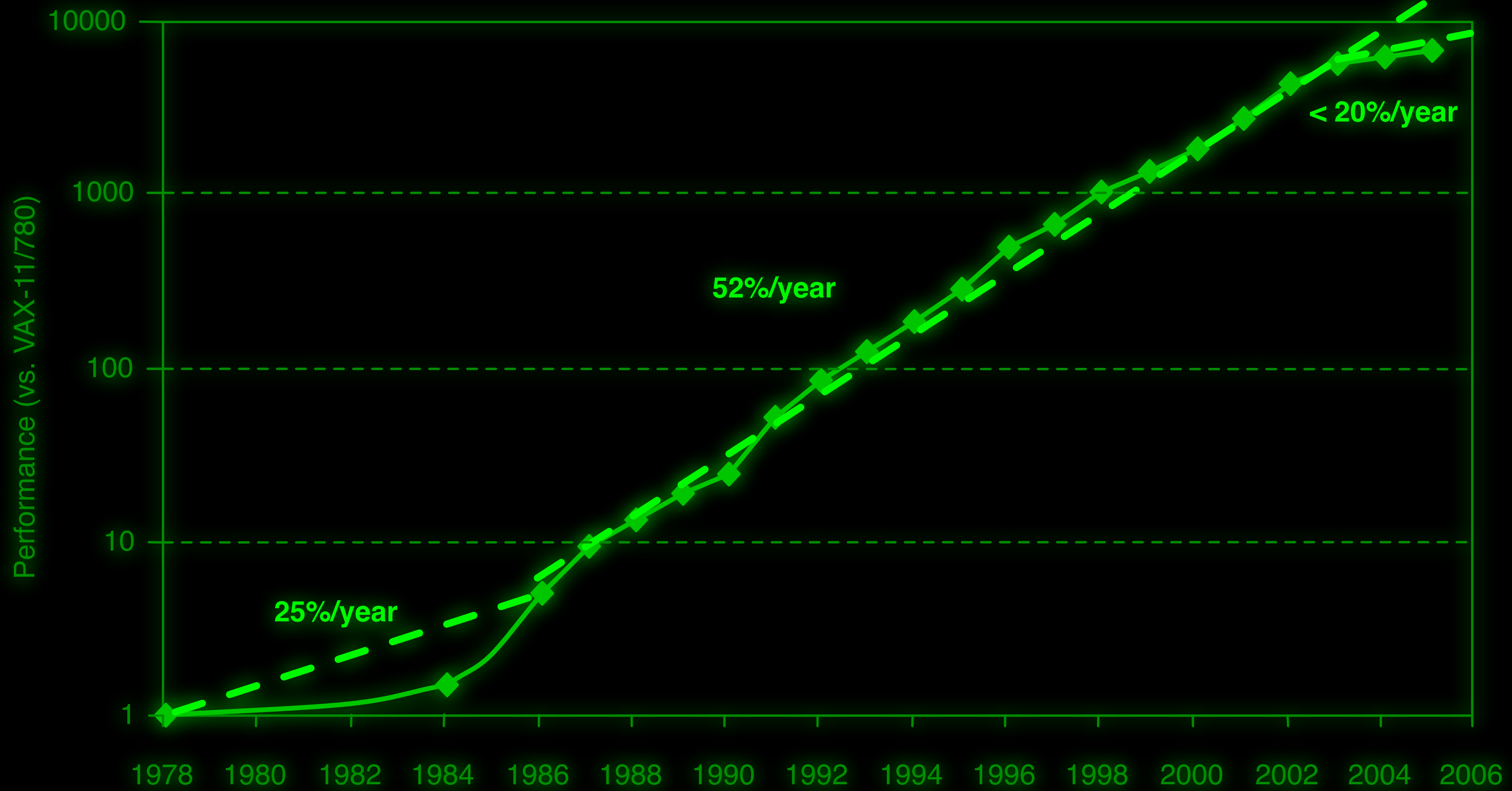
The Great Depression

Thorbiörn Fritzon

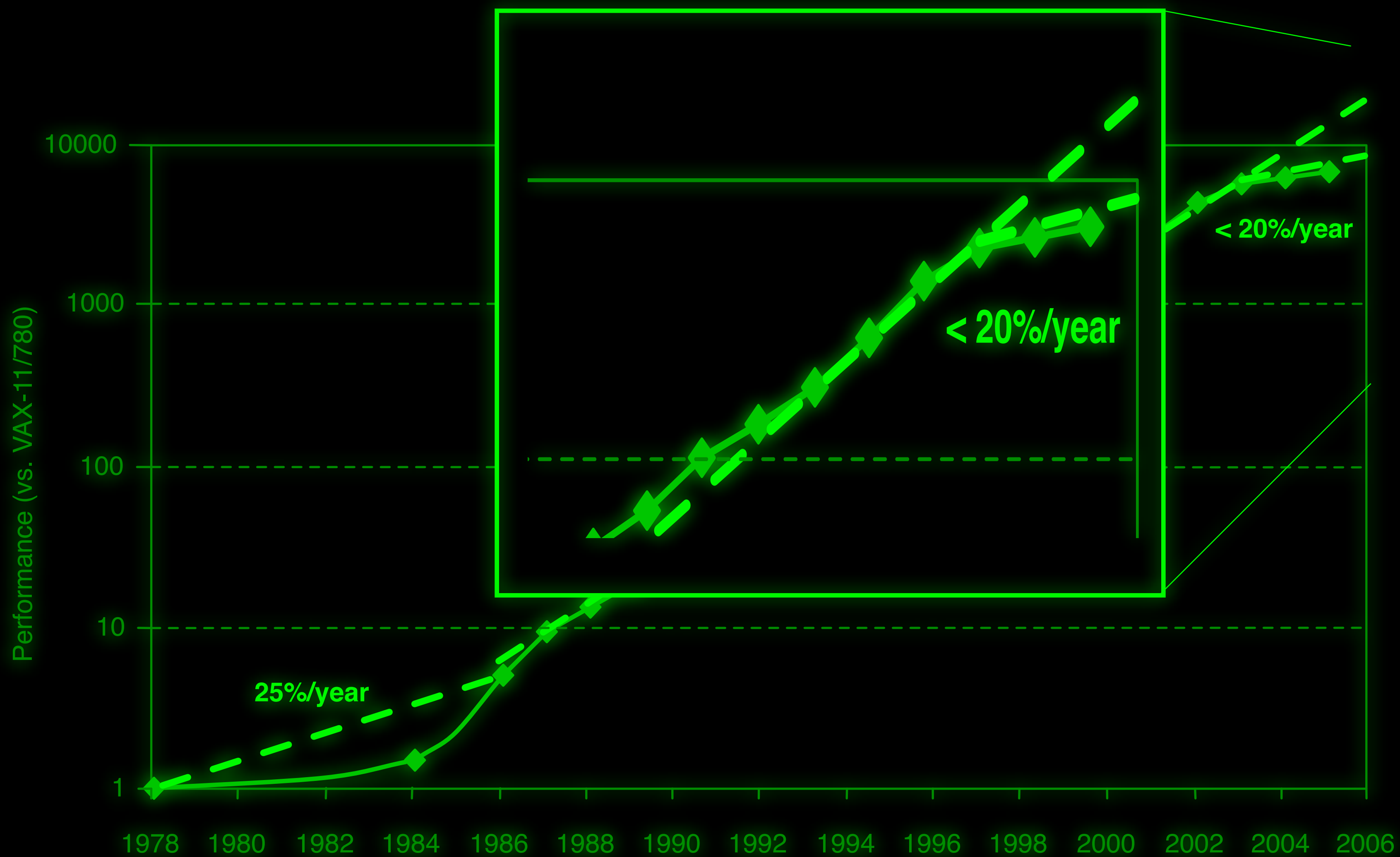
The Great Computing Depression

Thorbiörn Fritzon
Sr. Systems Engineer

Diminishing Returns



Diminishing Returns



Conventional Wisdoms: 1

old wisdom:

Power is free, but transistors are expensive

new wisdom:

"Power Wall": Power is expensive, but transistors are "free". That is, we can put more transistors on a chip than we have the power to turn on.

Conventional Wisdoms: 3

old wisdom:

Monolithic uniprocessors in silicon are reliable internally, with errors occuring only at the pins.

new wisdom:

As chips drop below 65 nm feature sizes, they will have high soft and hard error rates. [Borkar 2005] [Mukherjee et al 2005]

Conventional Wisdoms: 7

old wisdom:

Multiply is slow but load and store is fast

new wisdom:

“Memory Wall”: Load and store is slow, but multiply is fast. Modern microprocessors can take 200 clock cycles to access RAM, but even floating point multiplications can be done in four clock cycles on a modern FPU. [Wulf and McKee 1995]

Conventional Wisdoms: 8

old wisdom:

We can reveal more instruction-level parallelism (ILP) via compilers and architecture innovation. Examples from the past include *branch prediction*, *out-of-order execution*, *speculation*, and *Very Long Instruction Word* systems.

new wisdom:

“ILP Wall”: There are diminishing returns on finding more ILP. [Hennessy and Patterson 2007]

Conventional Wisdoms: 9

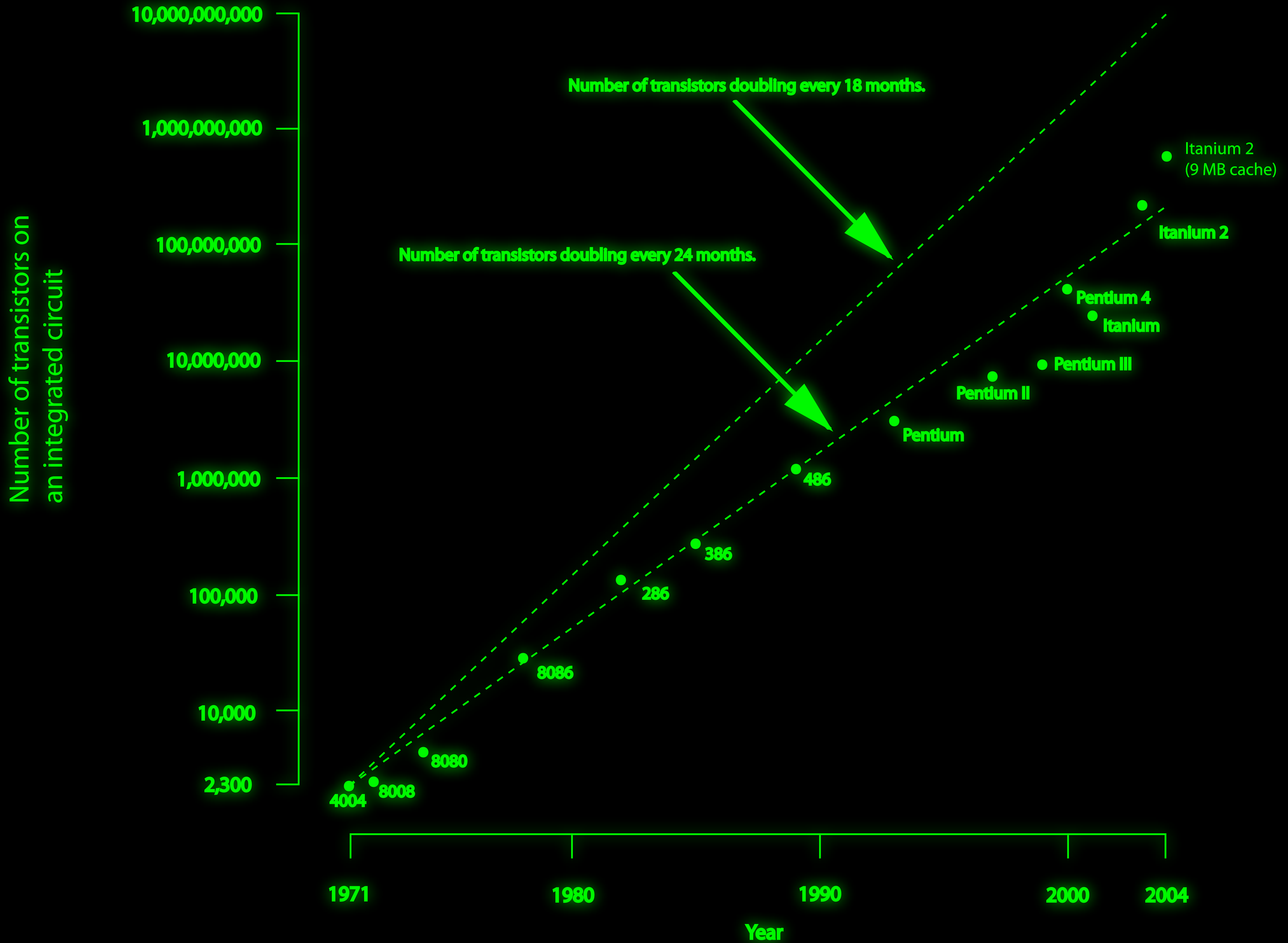
old wisdom:

Uniprocessor performance doubles every 18 months. (Moore's Law?)

new wisdom:

Power Wall + Memory Wall + ILP Wall = Brick Wall. In 2006, performance is a factor of three below the traditional doubling every 18 months that we enjoyed between 1986 and 2002. The doubling of uniprocessor performance may now take 5 years.

Moore's Law



Wikipedia:

High-Performance Multi-Core Processors

- AMD
  - Athlon 64, Athlon 64 FX and Athlon 64 X2 family, dual-core desktop processors.
  - Opteron, dual- and quad-core server/workstation processors.
  - Phenom, triple- and quad-core desktop processors.
  - Sempron X2, dual-core entry level processors.
  - Turion 64 X2, dual-core laptop processors.
  - Radeon and FireStream multi-core GPU/GPGPU (10 cores, 16 5-issue wide superscalar stream processors per core)
- Azul Systems Vega 2, a 48-core processor.
- Cavium Networks Octeon, a 16-core MIPS MPU.
- HP PA-8800 and PA-8900, dual core PA-RISC processors.
- IBM
  - POWER4, the world's first dual-core processor, released in 2001.
  - POWER5, a dual-core processor, released in 2004.
  - POWER6, a dual-core processor, released in 2007.
  - PowerPC 970MP, a dual-core processor, used in the Apple Power Mac G5.
  - Xenon, a triple-core, SMT-capable, PowerPC microprocessor used in the Microsoft Xbox 360 game console.
- IBM, Sony, and Toshiba Cell processor, a nine-core processor with one general purpose PowerPC core and eight specialized SPUs (Synergystic Processing Unit) optimized for vector operations used in the Sony PlayStation 3.
- Intel
  - Celeron Dual Core, the first dual-core processor for the budget/entry-level market.
  - Core Duo, a dual-core processor.
  - Core 2 Duo, a dual-core processor.
  - Core 2 Quad, a quad-core processor.
  - Core i7, a quad-core processor, the successor of the Core 2 Duo and the Core 2 Quad.
  - Itanium 2, a dual-core processor.
  - Pentium D, a dual-core processor.
  - Teraflops Research Chip (Polaris), an 3.16 GHz, 80-core processor prototype, which the company says will be released within the next five years[6].
  - Xeon dual-, quad- and hexa-core processors.
- IntellaSys seaForth24, a 24-core processor.
- Nvidia
  - GeForce 9 multi-core GPU (8 cores, 16 scalar stream processors per core)
  - GeForce 200 multi-core GPU (10 cores, 24 scalar stream processors per core)
  - Tesla multi-core GPGPU (8 cores, 16 scalar stream processors per core)
- Sun Microsystems
  - UltraSPARC IV and UltraSPARC IV+, dual-core processors.
  - UltraSPARC T1, an eight-core, 32-thread processor.
  - UltraSPARC T2, an eight-core, 64-concurrent-thread processor.
- Tilera TILE64, a 64-core processor

Erik Hagersten:

High-Performance Single-Core Processors

Vector CPU

- Early 60's: Solomon Project
- Cray-1 (2, X-MP, Y-MP)
- SIMD
  - Single Instruction, Multiple Data
- MIMD
  - Multiple Instruction, Multiple Data
- AltiVec: Velocity Engine, VMX
- Cell
- GPU: GPGPU

Multi-Core and CMT

Parallel
Thread Execution

C — Compute Cycles

M — Memory Wait Cycles

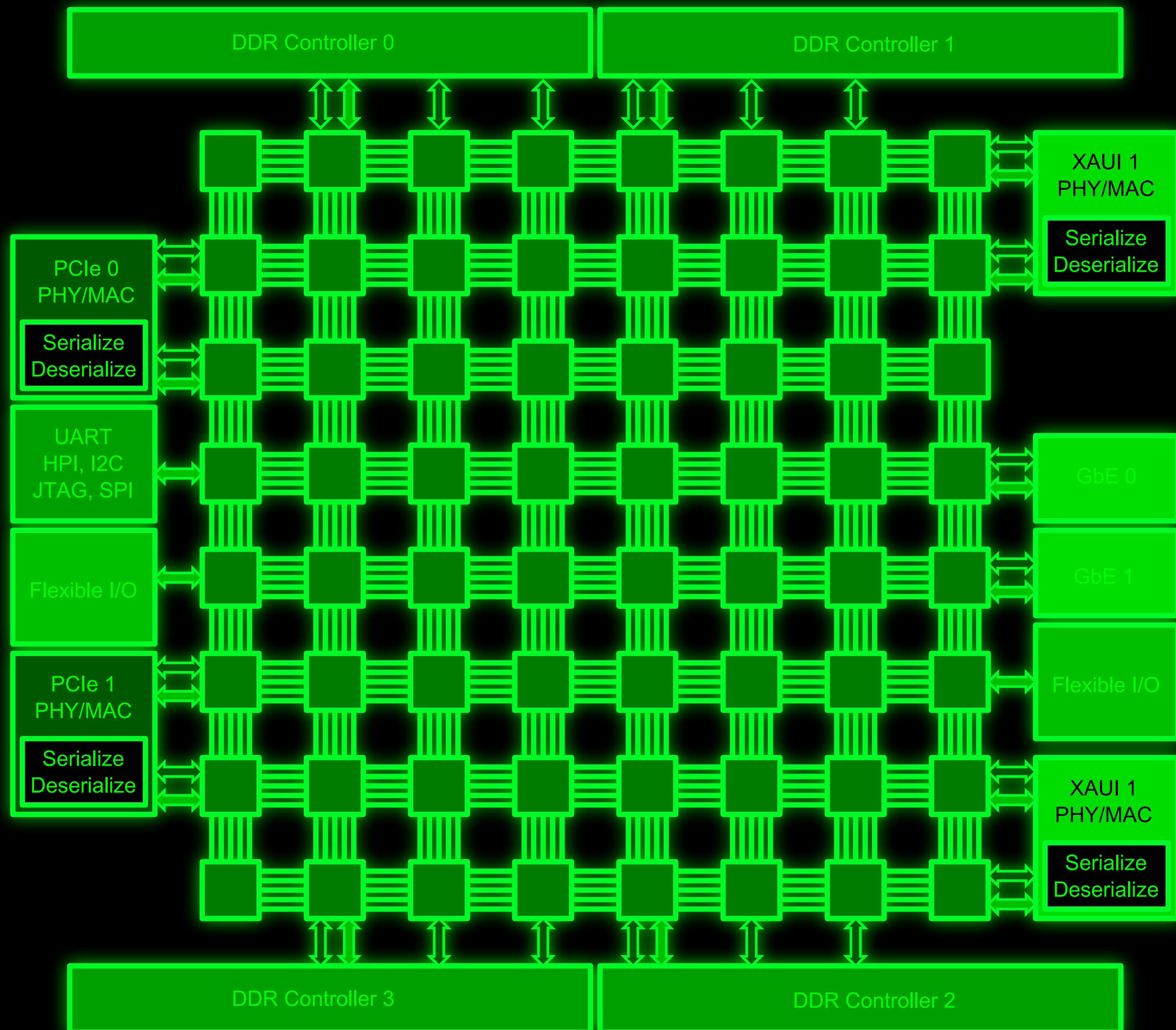
Core 2

C M C M C M C M
C M C M C M C M
C M C M C M C M
C M C M C M C M

Core 1

C M C M C M C M
C M C M C M C M
C M C M C M C M
C M C M C M C M

Many-Core



?

Homogenous or Heterogenous?

“In six years (2014), we will have 1000 cores”
-Prof. Anant Agarwal, MIT & Tilera

Programming in a Dark Future

The Problem

- Write a program for 1 CPU, scale it linear to n CPUs
- But, we don't understand:
  - What hardware support is needed
  - What language to use
  - How to get it efficient
  - How to get it right
  - What education is needed
- We live in interesting times

Concurrency and Parallelism Models

- Explicit
  - Threads, Processes, Co-Routines
  - Actors
  - Futures
- Hinting
  - Quasi-Static Scheduling
- Implicit
  - Stream-Based Programming
  - Pure Functional Programming
- Data Parallelism
  - Arrays
  - Generators
  - Map/Reduce

Futures

```
interface ArchiveSearcher { String search(String target); }
 class App {
   ExecutorService executor = ...
   ArchiveSearcher searcher = ...
   void showSearch(final String target)
       throws InterruptedException {
     Future<String> future=
             executor.submit(new Callable<String>() {
         public String call() {
             return searcher.search(target);
         }});
     displayOtherThings(); // do other things while searching
     try {
       displayText(future.get()); // use future
     } catch (ExecutionException ex) { cleanup(); return; }
   }
 }
```

Futures

IO Programming Language:

```
// async, immediately returns a Future
futureResult := obj @foo
// Do something else while we wait
print futureResult
```

Map/Reduce

Haskell: $(n^2)!$

n!:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

or:

```
factorial n = foldr (*) 1 [1..n]
```

map a function to a list:

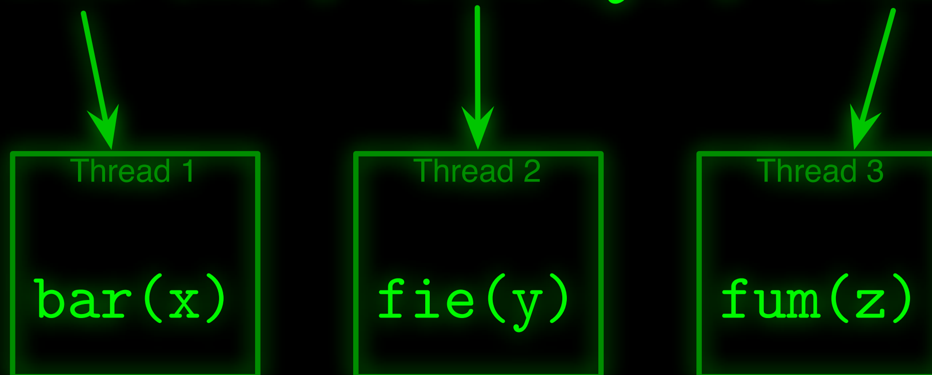
```
map (\n -> n*n) [1..n]
```

hence, $(n^2)!$:

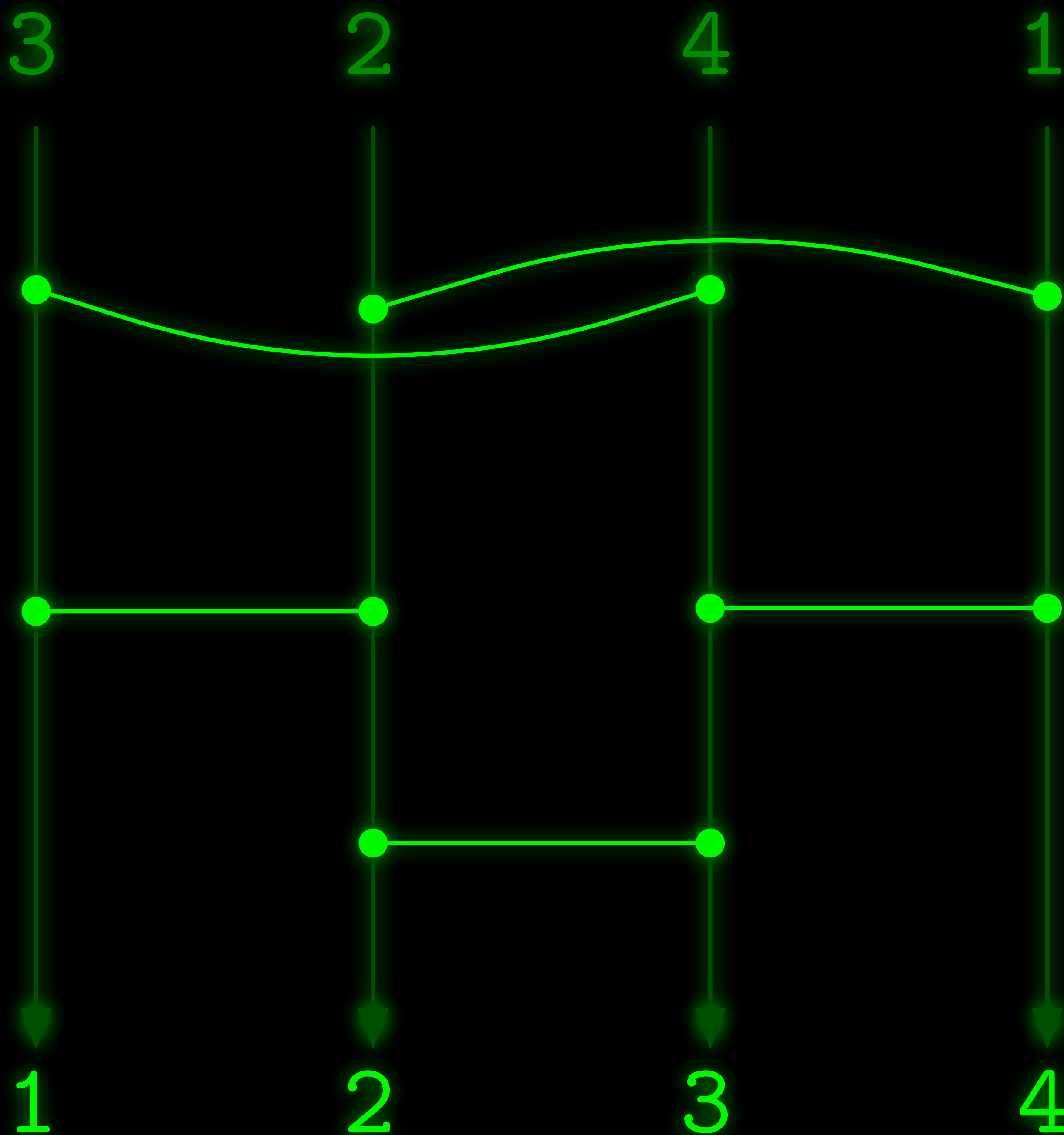
```
foldr (*) 1 (map (\n -> n*n) [1..n])
```

Pure Functional

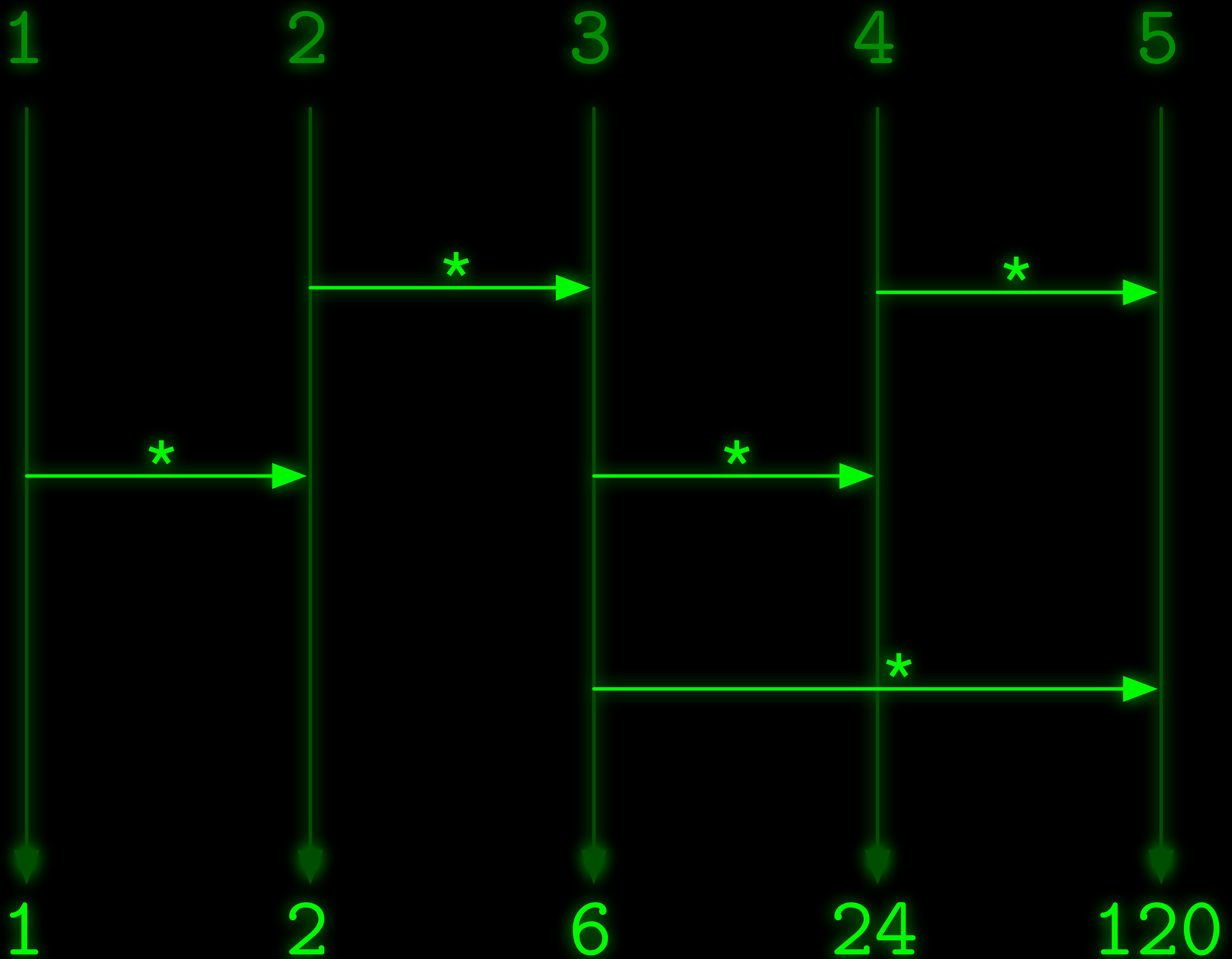
```
foo( bar(x), fie(y), fum(z) )
```



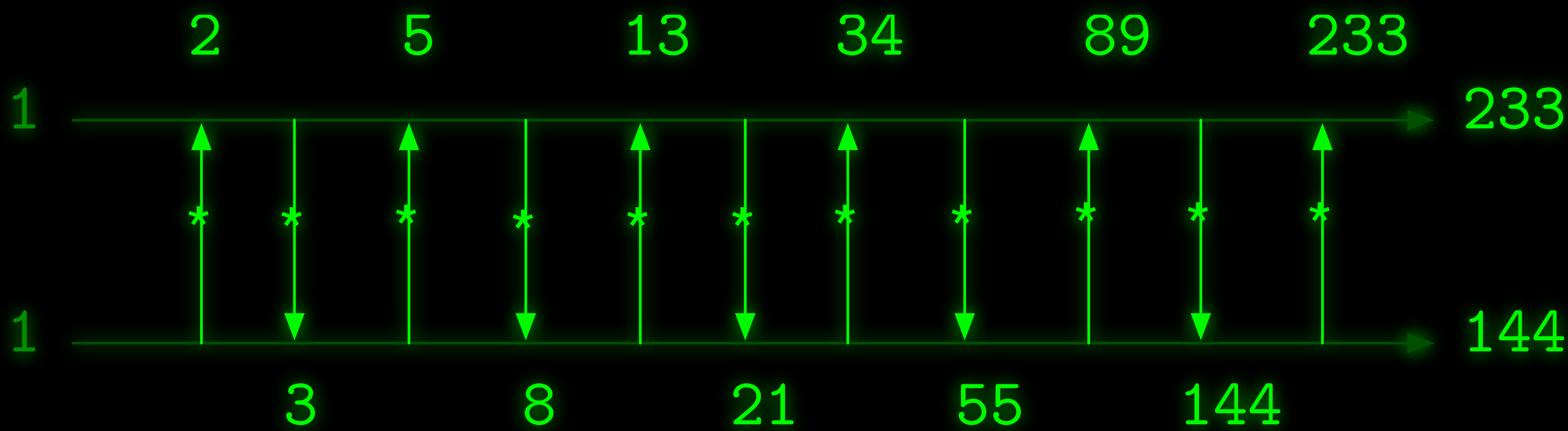
Sorting Networks



Calculating Networks



Calculating Networks



“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.”

-Guy Steele

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.

Why?

- Good sequential algorithms minimize space usage.
  - Clever tricks to reuse storage.Clever
  - Good parallel code often requires extra space to permit temporal decoupling.

Why?

- 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

```
SUM = 0        // OOPS!

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

What Does a Mathematician Say?

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

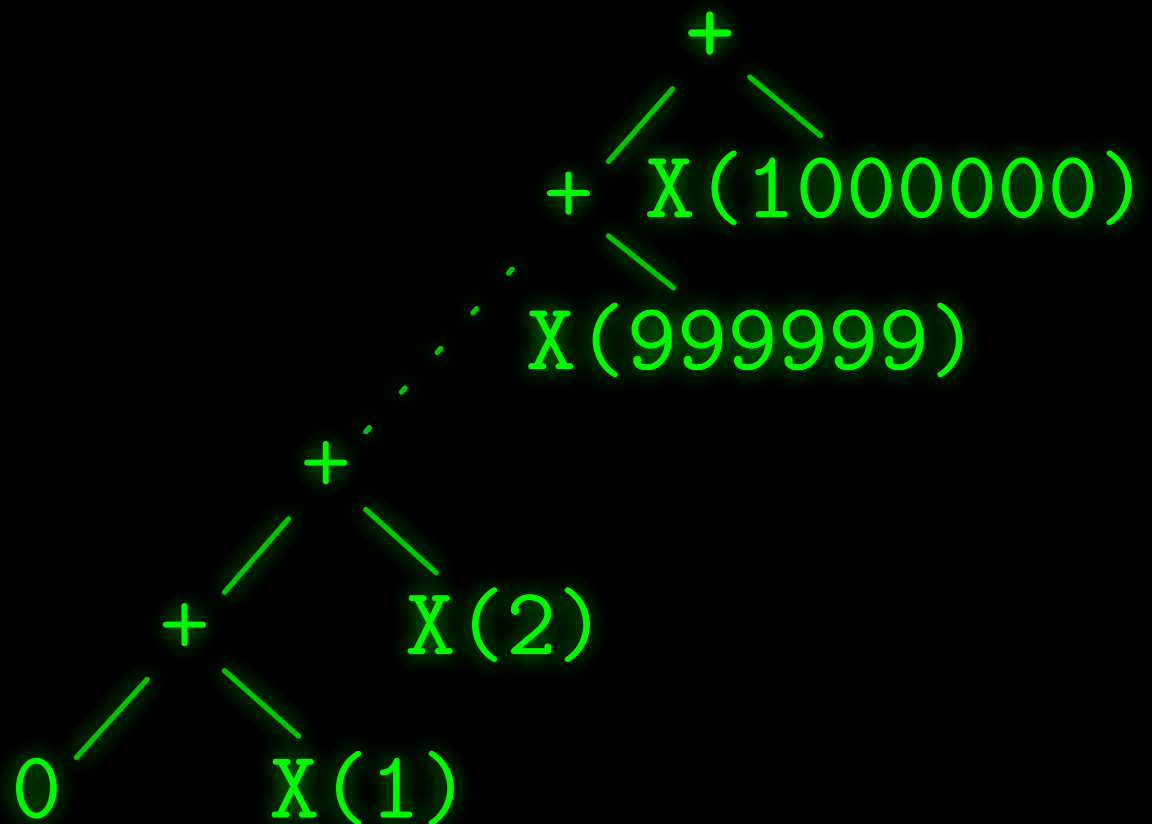
Compare FORTRAN 90: SUM(X).

What, not how?

No commitment to strategy. This is good.

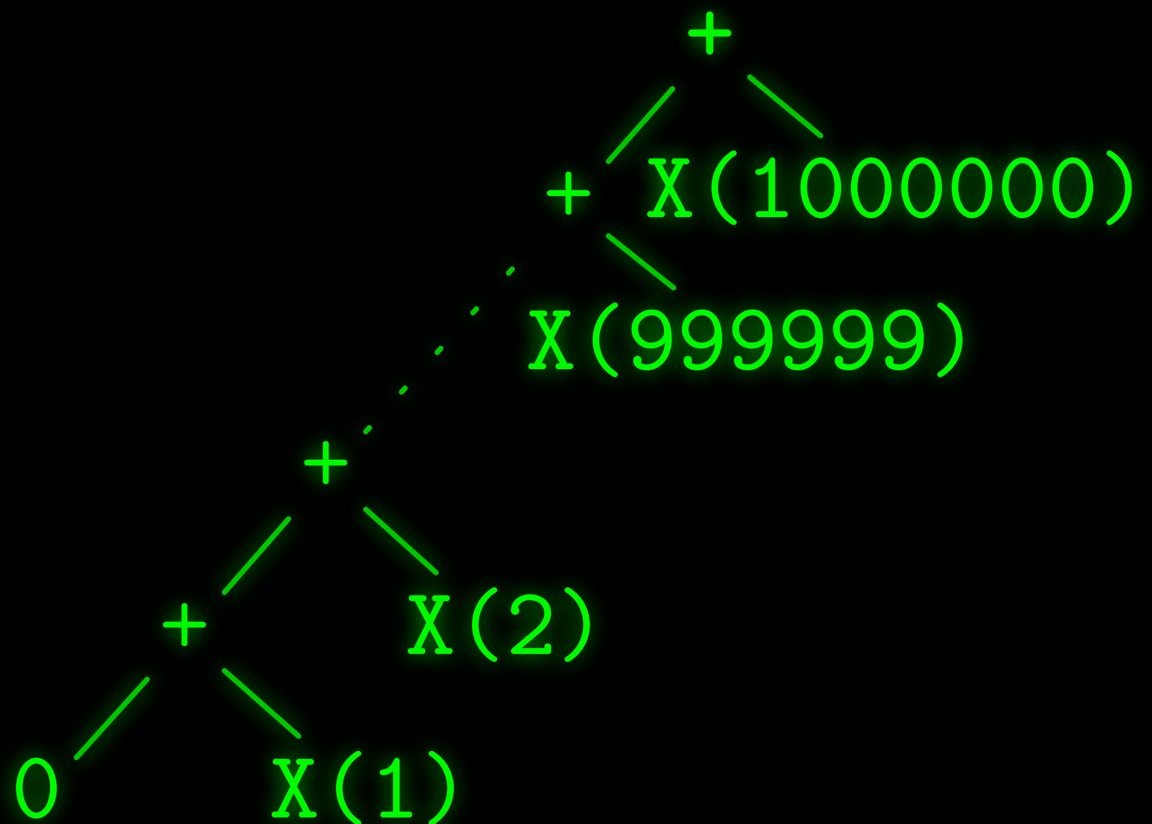
Let’s Add a Bunch of Numbers

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



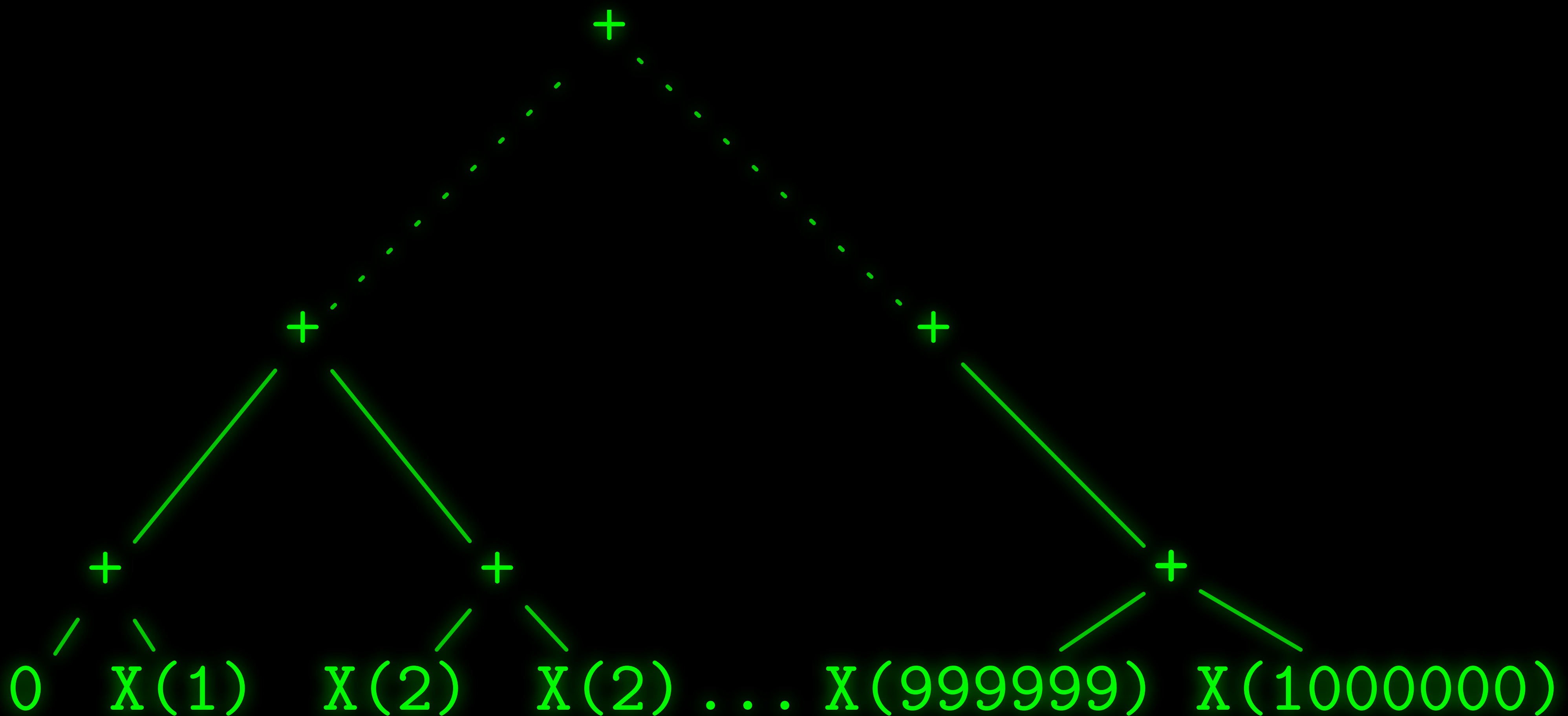
Atomic Update Computation Tree

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

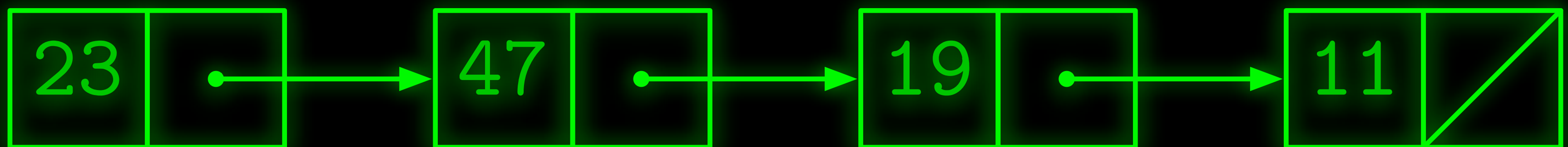


Parallel Computation Tree

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

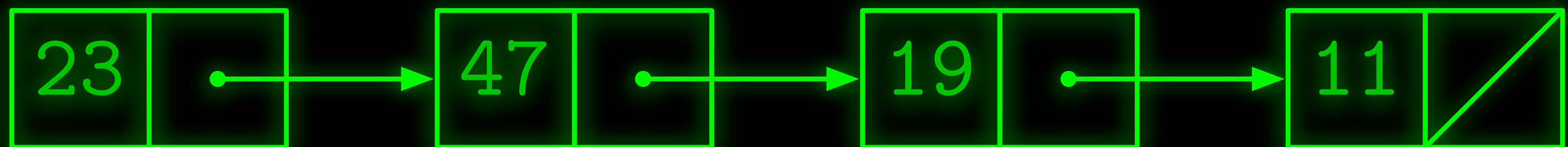


Finding the Length of a LISP List, recursive




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

Finding the Length of a LISP List, iterative




```
(define length (list)
 (do ((x list (rest x))
      (n 0 (+ n 1)))
   ((null x) n)))
```

Length of an Object-Oriented List

```
class List<T> {
  abstract int length();
}

class Empty extends List {
  int length() { return 0; }
}

class Node<T> extends List<T> {
  T first;
  List<T> rest;
  int length() { return 1 + rest.length(); }
}
```

Linear versus Multiway Decomposition

- These are important program decomposition strategies, but inherently sequential.
  - Mostly because of the linearly organized data structure.
  - 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!

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 (“This is a sample”)
  println words (“ Here is   another sample ”)
  println words (“JustOneWord”)
  println words (“ ”)
  println words (“”)
- Expected output:
  ⟨ This, is, a, sample ⟩
  ⟨ Here, is, another, sample ⟩
  ⟨ JustOneWord ⟩
  ⟨ ⟩
  ⟨ ⟩

Splitting a String into Words (3)

```
words (s:String) = do
  result : List⟦String⟧ := ⟨⟩
  words : String := “”

  c : 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 (4)

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words

Splitting a String into Words (5)

```
maybeWord(s : String):List⟦String⟧ =
  if s = “” then ⟨⟩ else ⟨s⟩ 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 =
      Segement(s || other.l, other.A, other.r)
end

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 (7)

$$processChar(c\!:\!\text{String})\!:\!\text{WordState} =$$

$$\texttt{if}\ \ (c = \text{`` ''})\ \ \texttt{then}\ \ \text{Segment}(\text{``''}, \langle\rangle, \text{``''})$$

$$\texttt{else}\ \text{Chunk}(c)$$

$$\texttt{end}$$

$$words(s\!:\!\text{String})\ = \texttt{do}$$

$$g = \bigoplus_{k \leftarrow 0 \# length(s)} processChar(substring(s, k, k+1))$$

$$\texttt{typecase}\ \ g\ \ \texttt{of}$$

$$\text{Chunk} \Rightarrow maybeWord(g.s)$$

$$\text{Segment} \Rightarrow maybeWord(g.l)\ ||\ g.A\ ||\ maybeWord(g.r)$$

$$\texttt{end}$$

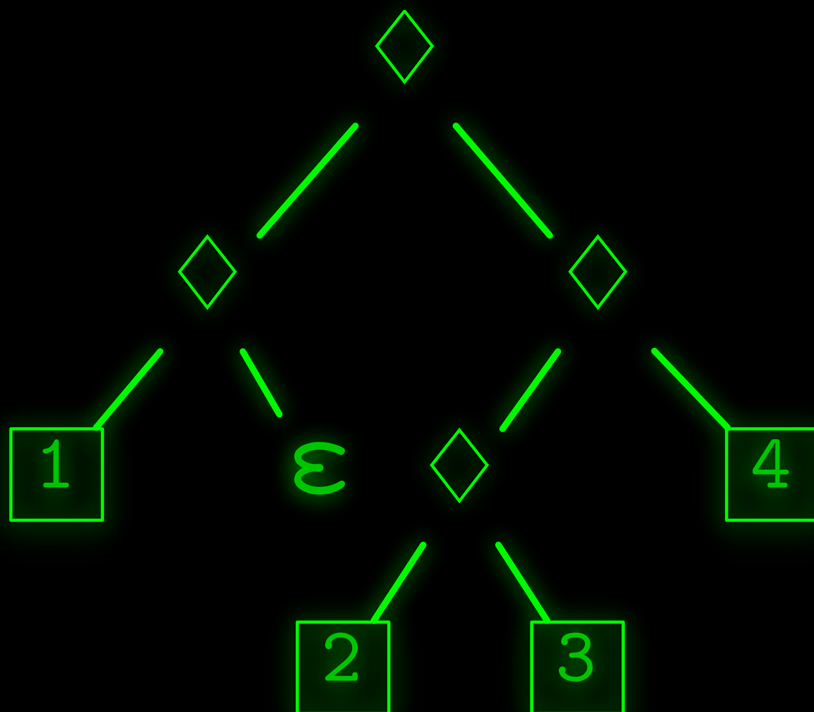
$$\texttt{end}$$

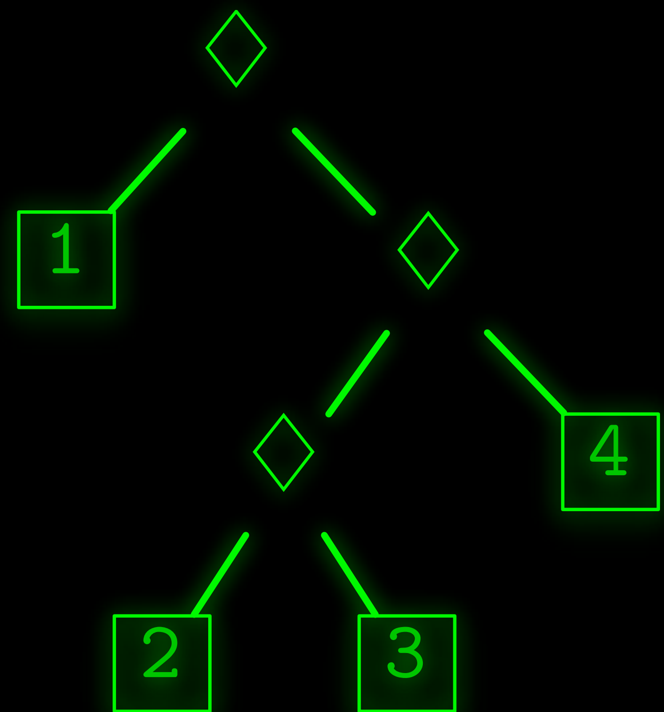
What’s Going On Here?

- 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?

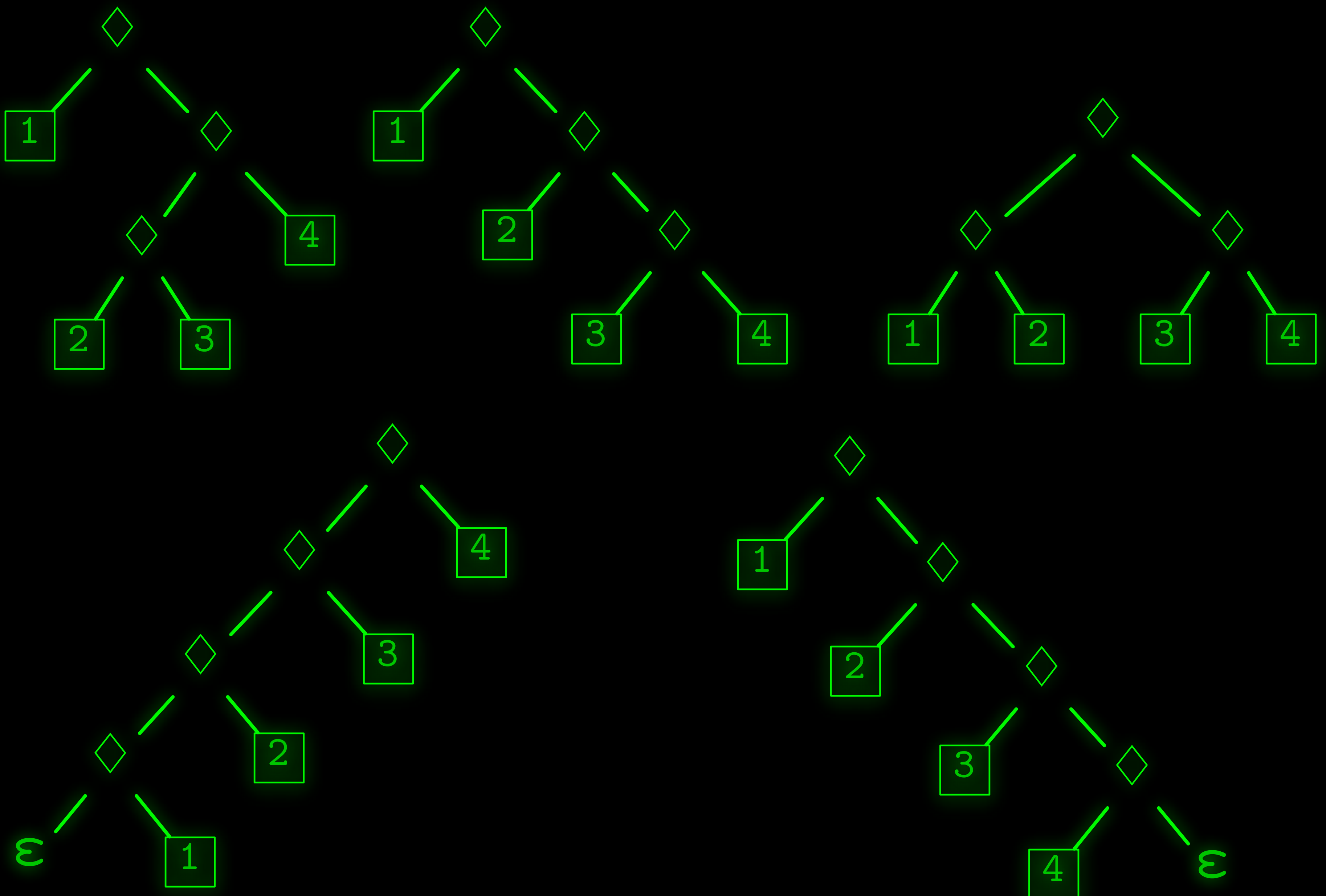
Representation of Abstract Collections

- Binary Operator: ◇
- Leaf operator (“unit”): □
- Optional empty collection (“zero”): ε





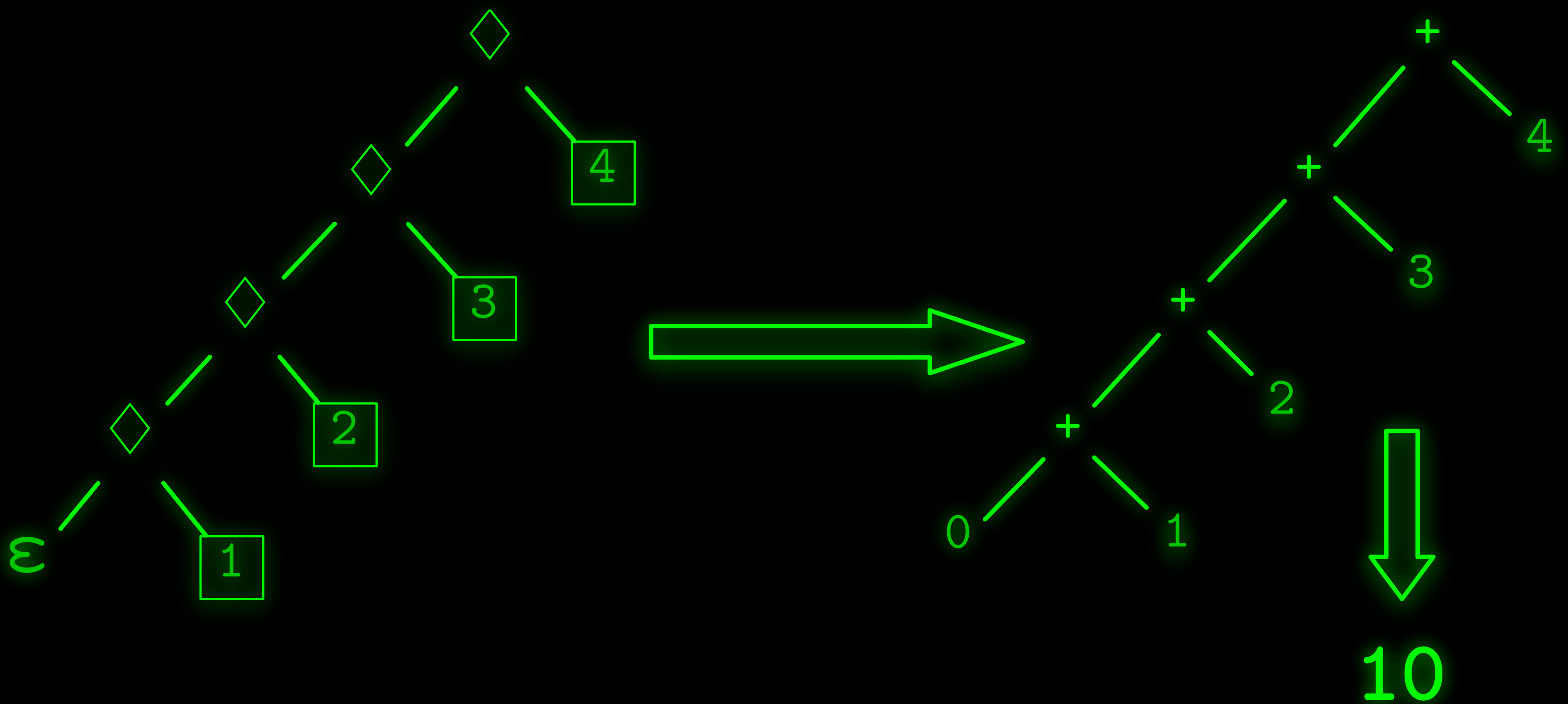
Associativity



These are all considered to be equivalent

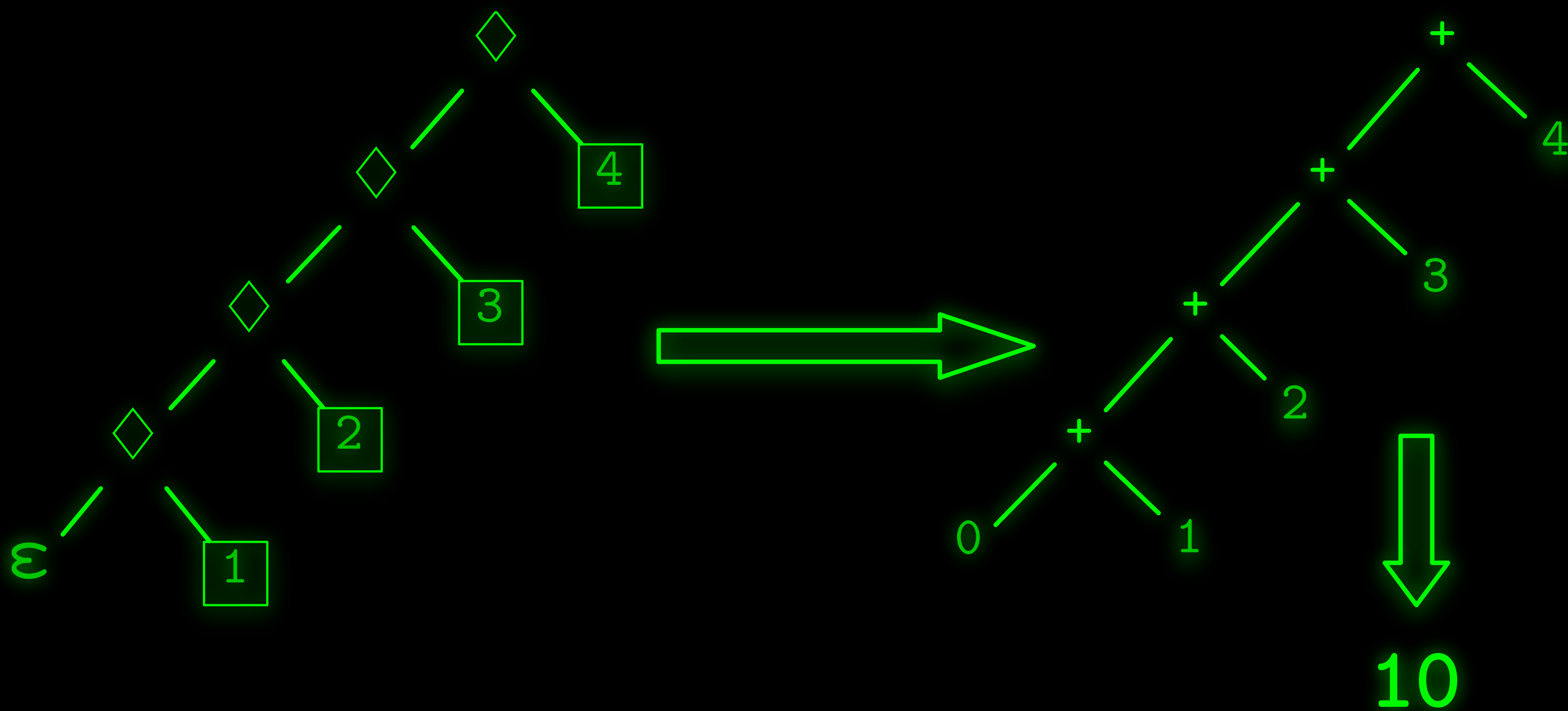
Catamorphism: Summation

- Replace ◊, □, ε with +, *identity*, 0



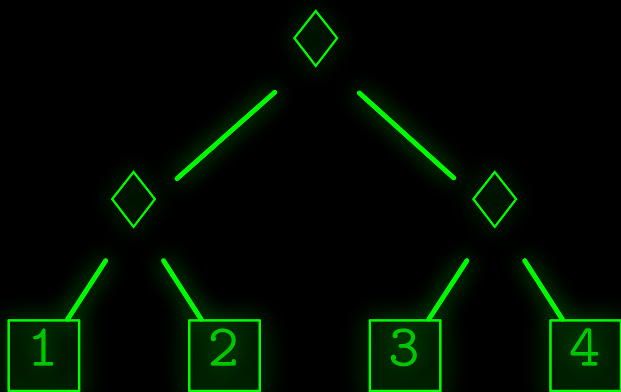
Computation: Summation

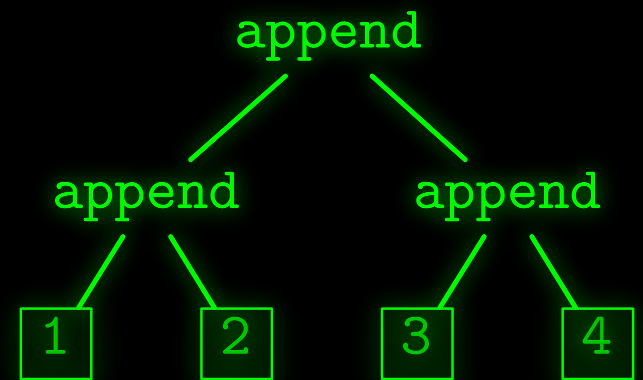
❖ Replace ◊, □, ε with +, *identity*, 0

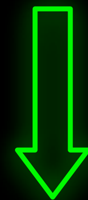


Catamorphism: Lists

- Replace $\Diamond$, $\Box$, ε with *append*, $\langle - \rangle$, $\langle\rangle$



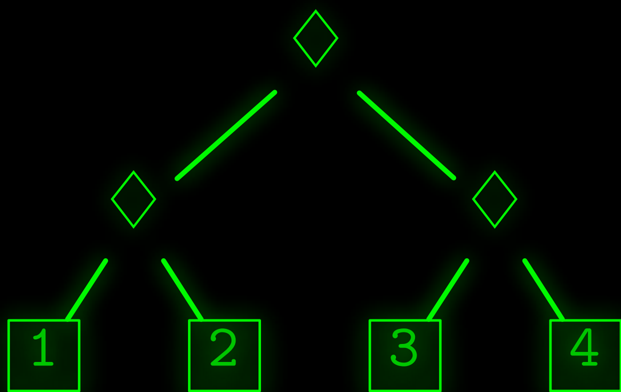


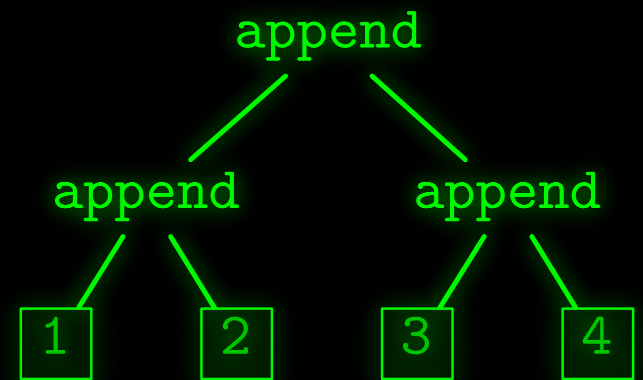


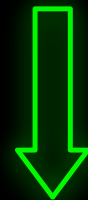
$\langle 1,2,3,4 \rangle$

Computation: Lists

- Replace ◊, □, ε with *append*, ⟨–⟩, ⟨⟩



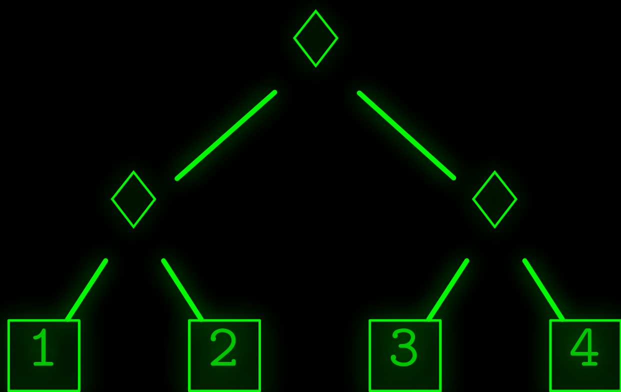


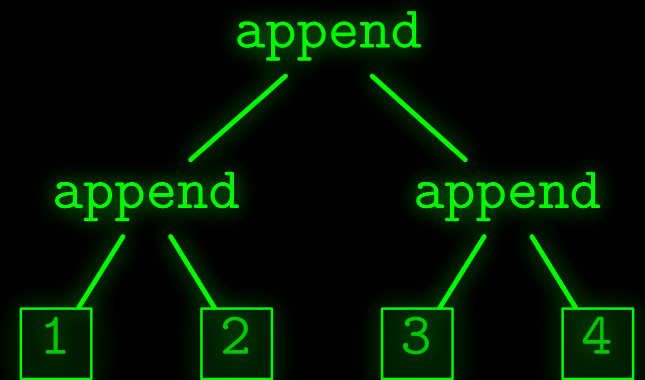


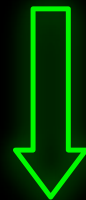
⟨1,2,3,4⟩

Representation: Lists

- Replace ◊, □, ε with *append*, ⟨-⟩, ⟨⟩





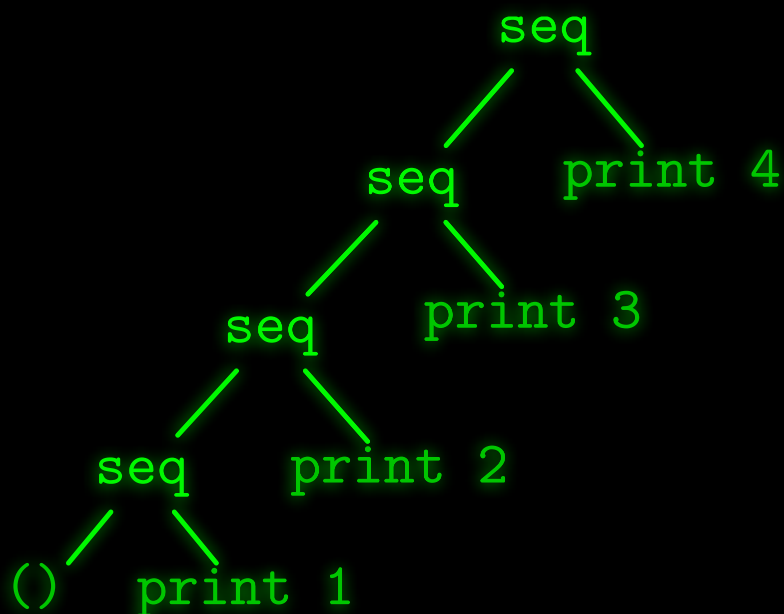


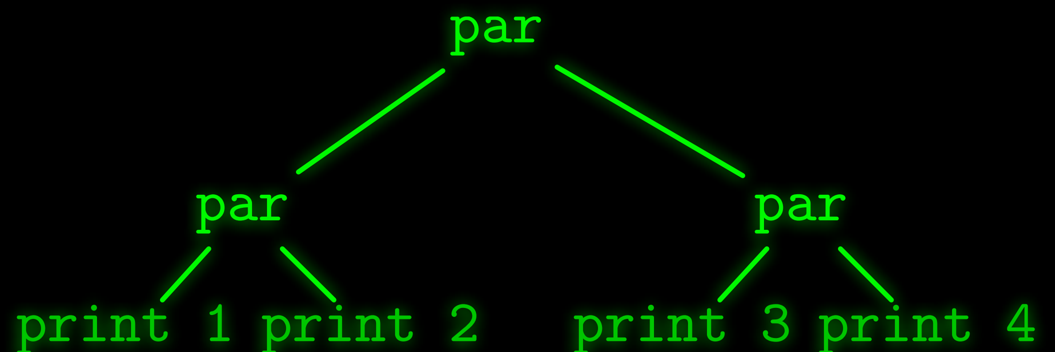
⟨1,2,3,4⟩

Computation: 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

To Summarize: A BIG Idea

- Loops and summations and list constructors are alike!

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

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

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

  - 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 just 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

  $s := s_0;$ **for** $i \leftarrow seq(1:100000)$ **do** $s := f_i(s)$ **end**

  compute $s(\underset{i \leftarrow 11000000}{\circ} f_i) s_0$
- 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

We Need a New Mindset

- DO loops are so 1950s!
- So are linear linked lists!
- 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.

Fortress

```
trait BinaryPredicate⟦T extends BinaryPredicate⟦T, ~⟧, opr ~⟧
  opr ~(self, other: T): Boolean
end
trait Symmetric⟦T extends Symmetric⟦T, ~⟧, opr ~⟧
    extends { BinaryPredicate⟦T, ~⟧ }
  property ∀(a:T, b:T )(a ~ b) ↔ (b ~ 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
```

Fortress: A Parallel Language

- High productivity for multicore, SMP, and cluster computing
- Hard to write a program that isn't potentially parallel
- Support for parallelism at several levels
  - Expressions
  - Loops, reductions, and comprehensions
  - Parallel code regions
  - Explicit multithreading
- Shared global address space model with shared data
- Thread synchronization through atomic blocks and transactional memory

These Are All Potentially Parallel

$f(x) + g(x)$

$L = \langle find(k, x) | k \leftarrow 1 : n, x \leftarrow A \rangle$

$$s = \sum_{k \leftarrow 1:n} c_k x^k$$

for $k \leftarrow 1 : n$ do

$a_k := b_k$

$sum \mathrel{+}= c_k x^k$

end

do

$f(a)$

also do

$g(b)$

end

do

$T_1 =$ spawn

$T_2 =$ spawn

$T_1.wait(); T_2.wait()$

end

Mathematical Syntax 1

- Integrated mathematical and object-oriented notation
- Supports a stylistic spectrum that runs from Fortran to Java™ - and sticks out at both ends!
  - More conventionally mathematical than Fortran
    - Compare a*x**2+b*x+c and $a\,x^2 + b\,x + c$
  - More object-oriented than Java
    - Multiple inheritance
    - Numbers, booleans, and characters are objects
    - If you prefer $\#S$, defining it is a one-liner.
  - To find the size of a set S: either $|S|$ or $S.size$

Mathematical Syntax 2

- Full Unicode character set available for use, including mathematical operators and Greek letters:

$\times$ $\times$ $\oplus$ $\ominus$ $\otimes$ $\oslash$ $\odot$ $\approx$ α β γ δ

$\boxplus$ $\boxminus$ $\boxtimes$ $\leftrightarrow$ $\wedge$ $\vee$ $\equiv$ Γ ϵ ζ η θ

$\leq$ $\geq$ $\sum$ $\prod$ $\prec$ $\preccurlyeq$ $\succcurlyeq$ $\succ$ ι κ λ μ

$\cap$ $\cup$ Θ $\subset$ $\subseteq$ $\supseteq$ $\supset$ $\in$ ξ π ρ σ

$\sqcap$ $\sqcup$ $\sqsubset$ $\sqsubseteq$ $\sqsupseteq$ $\sqsupset$ $\neg$ $\notin$ ϕ χ ψ ω

$\lfloor$ $\rfloor$ $\lceil$ $\rceil$ $\langle$ $\rangle$ ⅄ $\curlyvee$ τ and so on

- Use of “funny characters” is under the control of libraries (and therefore users)

Project Fortress

- http://projectfortress.sun.com
- An open-source project with international participation
- Open source since January 2007
- University participation includes:
  - University of Tokyo: matrix algorithms
  - Rice University: code optimization
  - Aarhus University: syntactic abstraction
  - University of Texas at Austin: static type checking
- Also participation by many individuals

A Growing Library

- The Fortress library now includes over 12,000 lines of code.
  - Integer, floating-point, and string operations
  - Big integers, rational numbers, intervals
  - Collections (lists, sets, maps, heaps, etc.)
  - Multidimensional arrays
  - Sparse vectors and matrices
  - Generators and reducers
    - Implement loops, comprehensions, and reductions
    - Support implicit parallelism
  - Fortress abstract syntax trees
  - Sorting

What works NOW?

- Parallelism in loops, reductions, comprehensions, tuples
- Automatic load balancing via work-stealing

What Works NOW?

- Object-oriented type system with multiple inheritance
- Overloaded methods and operators with dynamic multimethod dispatch
- Sets, arrays, lists, maps, skip lists
- Pure queues, deques, priority queues
- Integers, floating-point, strings, booleans
- Big integers, rational numbers, interval arithmetic
- Syntactic abstraction (just barely)

Next steps:

- Full static type checker (almost there!)
- Static type inference to reduce “visual clutter”
- Parallel nested transactions
- Compiler
  - Initially targeted to JVM for full multithreaded platform independence
  - After that, VM customization for Fortress-specific optimizations

The Parallel Future

- We need to teach 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.
- This is our only hope for program portability in the future.

Thorbiörn Fritzon
<fritzon@sun.com>