Vectoral: only by trying new things will we ever get computer languages right

Alan Wray
NASA Ames

What it tries to be . . .

a language for expressing scientific and engineering computational tasks

a concise notation for such algorithms, as consistent as possible with normal mathematical notation

syntactically orthogonal

a self-documenting balance between terse and verbose

minimally pre-conceived and thus hopefully novel

A Brief History of the Compiler

ca. 1978: First implementation for Illiac IV
Written in PL/I, generated Illiac binary
Cross-compiler running on IBM 360/67

1979: Illiac cross-compiler ported to VAX-11

1980: Generator for Cray 1 binary
Compiler rewritten in Vectoral to run on Cray

1982: Cyber 205 binary, first cross-compiled on Cray
then ported to run natively on 205

1980's-now: Series of ports to XMP, YMP, Cray 2, C90, J90, T90, SV1

1989: Port to Intel i860-based parallel computers:
Gamma = 32 to 128 processors, Delta = 528 processors

A Brief History of the Language

Illiac days: Explicit vector notation, asynchronous I/O, user control of both stack and heap storage

Cray days: Vector "loops" replace explicit vector notation, array operations introduced

C-days: Object-oriented features, mutitasking added

Inspirations

Algol, Algol 68, Fortran, PL/I, Pascal, APL, CFD, C++, even Basic ...

Overall compiler structure

- Single pass parser with occasional local re-parsing and parse-ahead
- Parser generates n-tuple intermediate code
- Optimizer makes several passes over the i-code
- Code generator forms binary or C from optimized i-code
- For binary, a scheduler reorders/modifies the instruction stream

Major Contrasts with C

- True multidimensional, variable bounded arrays in static, stack, or heap
- Complex type is native
- Exponentiation is native (^)
- Type precisions can be explicit
- No reserved words
- Fewer delimiters, more agglutinators: (), [], {}
- Richer control, data structures
- Flexible, mnemonic call syntax
- Vectoring, tasking built in
- Many more scientific intrinsics

Earliest (Illiac) Vector Syntax

```
(* # spans the vector (64 or 128 elements) *)

Static real A[#,10,10], B[#,10,10]
         index i, j, k, m[#]
 ...
A[#, i, j] = B[# + i, j, k]^2 + 2*B[#, m[#], k]

A[m[#]>3, i, j] = 0.
```

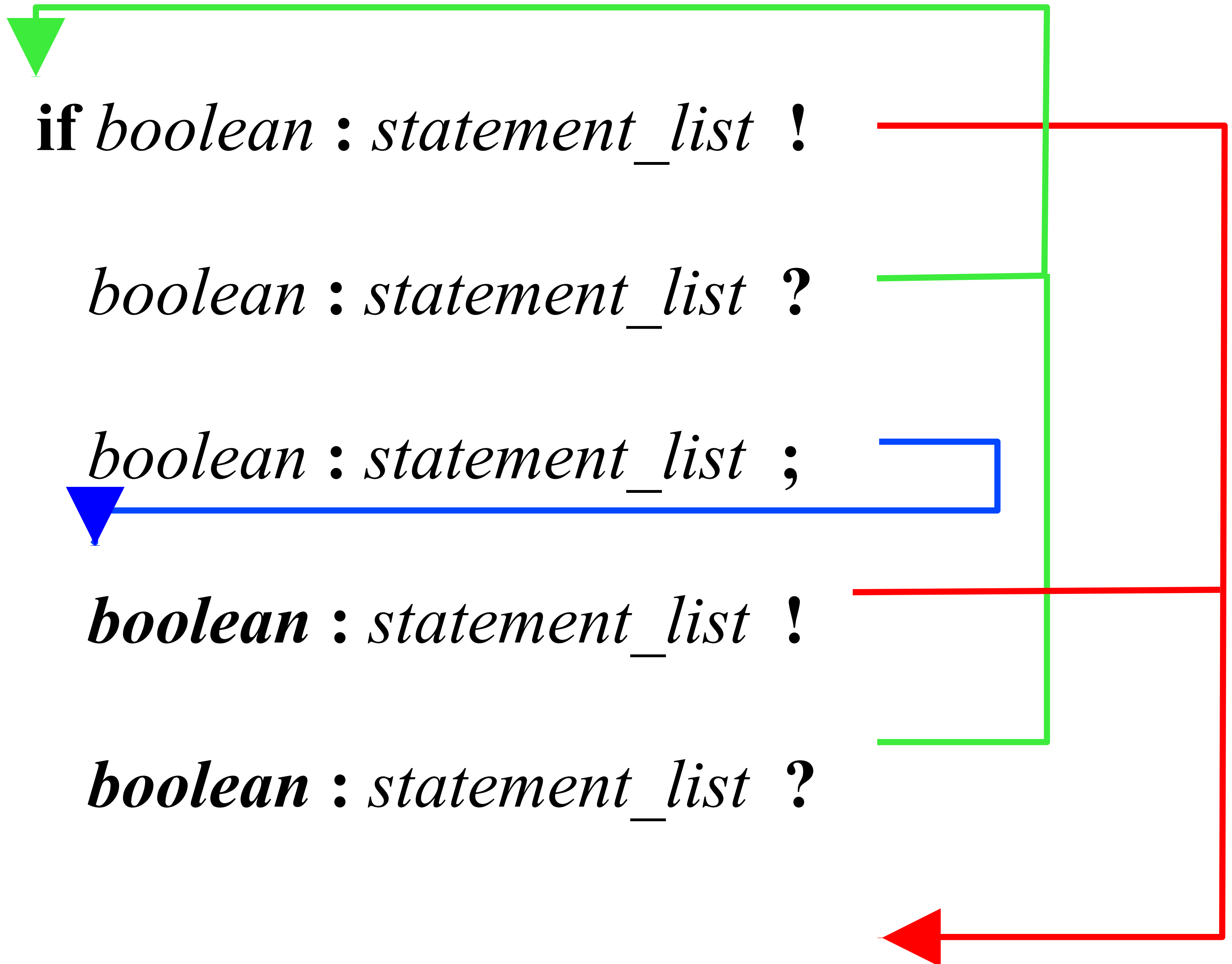
Revised Vector Syntax

(* n is some variable or constant expression giving the vector length for this part of the code *)

```
Static real A[n,10,10], B[n,10,10],
        index v, i, j, k, m[n]
 ...
For 1<=v<=n:
    A[v, i, j] = B[v+i, j, k]^2 + 2×B[v, m[v], k]

    if m[n]>3: A[v, i, j] = 0.  !
!
```

Control Structures 1: if



QUIZ:

Q: How do you do a plain "else"?

QUIZ:
Q: How do you do a plain "else"?
A: Use an empty boolean:

if *boolean* **:** *then_list* **!**
: *else_list* **!**

QUIZ:

Q: How about "while" or "do ... until"

QUIZ:
Q: How about "while" or "do ... until" ?
A: Use looping and empty booleans:

if *boolean* **:** *while_list* **?**

if **:** *do_list* **;**
not(*boolean***)** **:** **?**

QUIZ:

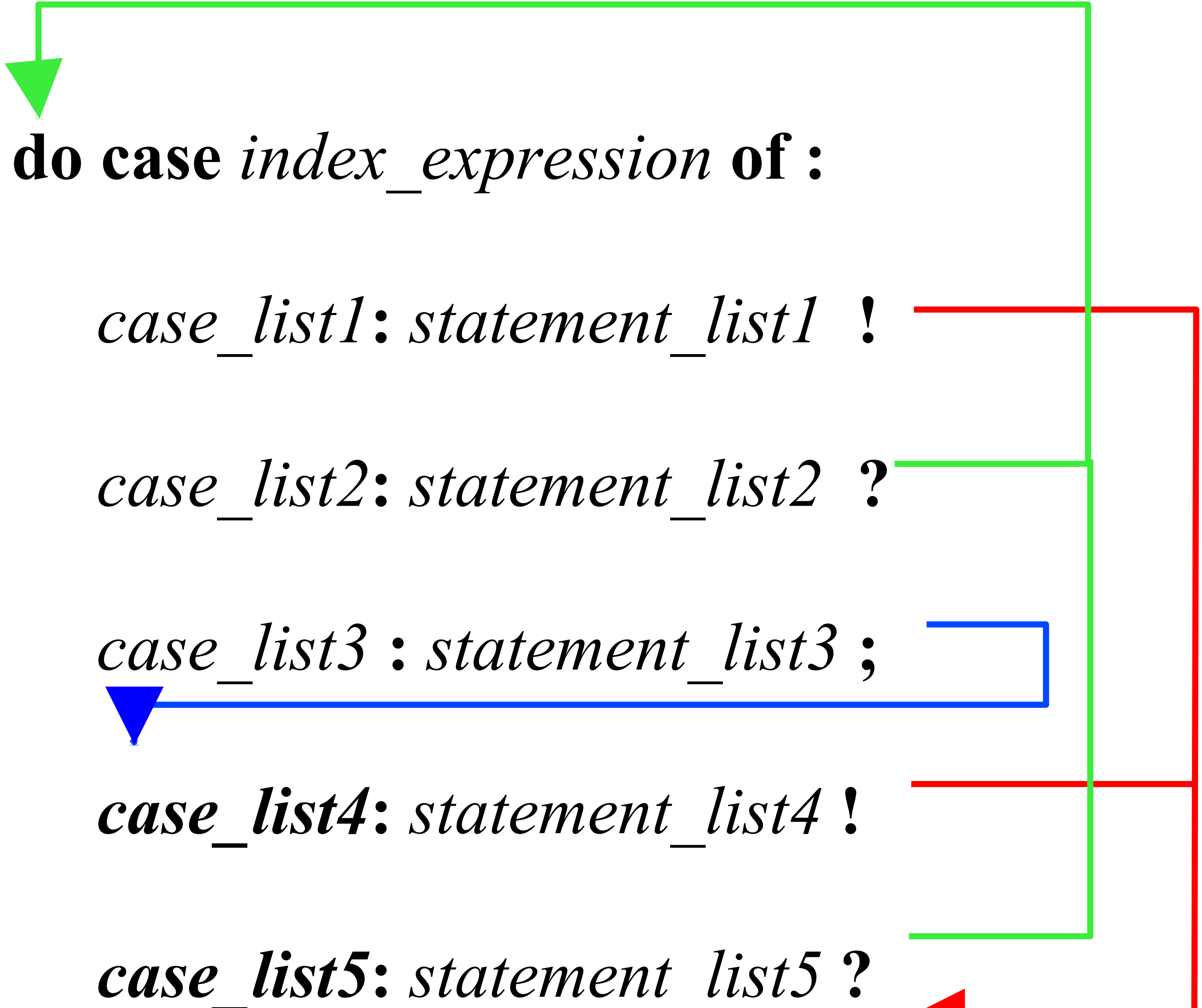
Q: Go to ???

QUIZ:

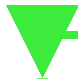
Q: Go to ???

A: Sorry.

Control Structures 2: do case



Control Structures 3: iteration



```
For i = 1 to n  and  -5 to -10 by -2  and  17 :
...
?
```

```
For m <= j <= n :
...
!
```

```
For n+1 > j > m-1:     -- equivalent to the last for
...
!
```

Control Structures 3, continued

Conditional vectorization:

For m <= j <= n, vectorize if n-m>10 :
...
!

Control Structures 4: defining procedures

lobal index a, b, string s, real r

ocal DOIT(real X , Y , Z) =
X = a*Y + b*Z

a=b: DOIT = : X = a*(Y + Z) ! !

lobal DOIT(string X, Y, Z) = : X = (a+b)*Y + Z !

OIT(s, "q", ".") -- calls the 2nd **DOIT**

OIT(r, 1, 3) -- calls the 1st **DOIT**

Control Structures 4: calling procedures

Local DOIT(real X , Y , Z)

raw call: DOIT(A, B, C)

keyword call: **DOIT(Y=B X=A Z=C)**

mixed call: **DOIT(Y=B, C, X=A)**

Control Structures 4: calling procedures with default arguments

Local DOIT(real X=A , Y , Z=C)

raw call: DOIT(, B)

keyword call: **DOIT(Y=B)**

ontrol Structures 4: calling procedures wi prepositional separators

ocal DOIT(real X with Y from Z)

aw call: **DOIT(A with B from C)**

or **DOIT(A from C with B)**

nixed call: **DOIT(A with B, Z=C)**

Control Structures 4: disambiguating calls

cal DOIT(real X with Y from Z),
DOIT(real x into y and z)

OIT(1., 2., 3.) -- compile-time error: ambiguous

OIT(X=1., 2., 3.) -- call 1st based on argument nam

OIT(a from c with b) -- call 1st based on separator(s)

OIT(1., y=a, b) -- call 2nd based on argument nam

Control Structures 5: Tasking

Dynamic task index i

For 1 <= i <= n:
-- a process is spawned for each i
!

For 1 <= i <= n, groups of m:
-- a process is spawned for each **m** **i**'s
!

Data Structures 1: Storage Classes

Memory classes

Global : name known to all compilations in a link
Local : name known throughout a compilation
Static : memory allocated once per run
Dynamic: memory allocated on entering scope

Definitional Classes

Symbolic : compile-time parameterized macro
Valoric : run-time temporary
Declarator: parameterized abstract type declaration

Data Structures 2: arrays and areas

Arrays may be multidimensional, variable bounded:

```
Global index m, n
Static real f[m, n, n+1],
          complex g[-1 to n, m to 2×m, n+1]

Dynamic integer(16) R[10, n]
```

Data Structures 2: arrays and areas, cont'd

reas are basically instantiated **structs**, but unambiguous **ferences to contained identifiers need not be qualified**

lobal dims{index m, n}

atic data{real f[m, n],

complex g[-1 to n, m to 2×m]}[n+1]

ynamic {integer(16) R[10, n], f[2,2,3]}

1,2,3] is the same as **data[3]{g[1,2]}** or **data{g[1,2]}**

data{g[1,2,3]}

Data Structures 3: Symbolic

ymbolic PI = 3.14159265358979,

CUBEROOT(x) =

if(x>0: x^[1./3.] !   x=0: 0 !   : -|x| ^ [1./3.] !)

Delta(A, I) = A[I+1] - A[I-1] ,

Diag[array=D, position] = array[position, position]

iag[A,i] is the same as **A[i,i]**

nd

Data Structures 3: Symbolic, cont'd

ymbolics can be assigned-to and used as compile-time op variables, tested, etc., to expand the source code at ompile-time:

```
ymbolic i, j, k=3

For i=1 to 10 :  a[i] = b[i]×c[k]
                 $if mod(i,2)=0: a[i] = a[i] + 1 !
                 $j = i + 1
                 d[j] = a[i]^2
                 $if i<5: k = k+1 !
                        : k = i     !
```

Data Structures 4: Declarator

eclarator **character c = bits(8) c,**
Array[n] q = q[n],
real_matrix[m,n] M = real M[m,n],
data_area(x,y) D = D{real x,y},
field(type, variable, dimension) F
= F{type variable}[dimension]

ocal character c_a, c_b[20]

equivalent to

ocal bits(8) c_a, c_b[20]

Data Structures 4: Declarator

```
eclarator character c = bits(8) c,
           Array[n] q = q[n],
           real_matrix[m,n] M = real M[m,n],
           data_area(x,y) D = D{real x,y},
           field(type, variable, dimension) F
                          = F{type variable}[dimension]
```

```
ynamic  complex Array[12] u, v, w,
        real Array[`2, 4`] U1, U2
```

equivalent to

```
ynamic  complex u[12],  v[12],  w[12],
        real U1[2, 4], U2[2, 4]
```

Data Structures 4: Declarator

eclarator character c = bits(8) c,
Array[n] q = q[n],
real_matrix[m,n] M = real M[m,n],
data_area(x,y) D = D{real x,y},
field(type, variable, dimension) F
= F{type variable}[dimension]

atic real_matrix[5,10] a, b, c[2]

equivalent to

atic real a[5,10], b[5,10], c[5,10,2]

Data Structures 4: Declarator

eclarator character c = bits(8) c,

Array[n] q = q[n],

real_matrix[m,n] M = real M[m,n],

data_area(x, y) D = D{real x, y},

field(type, variable, dimension) F

= F{type variable}[dimension]

atic data_area(a, b) A1,

data_area(p[10], q[5,4]) A2{real r[3]}

equivalent to

atic A1{real a, b},

A2{real p[10], q[5,4], real r[3]}

Data Structures 4: Declarator

```
eclarator  character c = bits(8) c,
           Array[n] q = q[n],
           real_matrix[m,n] M = real M[m,n],
           data_area(x,y) D = D{real x,y},
           field(type, variable, dimension) F
                        = F{type variable}[dimension]
```

```
ynamic  field(type=real   variable=`U,V`   dimension=`nx,ny`)
locity[2]
```

equivalent to

```
ynamic  velocity{real U,V}[nx, ny, 2]
```

Data Structures 5: Hoards continued

eclarator H(m) a = hoard a{index n, real x[m]} :

a == new(H(m)) a{n}=m !

ynamic hoard h{real x, y, z}

ynamic H(100) A

== new(h)

br1(h, A)

== renew(A as H(1000))

br2(h, A)

Data Structures 3: Hoards

```
clarator
 velocities     V = V{ real u, v, w },
 coordinates X = X{ real x, y, z },
 flow_data     Q = Q{ coordinates X, velocities V,
                           flow_data hoard neighbor[6] }

namic  flow_data data[nx, ny, nz]

r i=1 to nx:  for j=1 to ny:  for k=1 to nz:
  data{neighbor[1]}[i,j,k] ==  if(i=1  :  % ! : data[i-1, j, k] !)
  data{neighbor[2]}[i,j,k] ==  if(i=nx:  % ! : data[i+1, j, k] !)
  data{neighbor[3]}[i,j,k] ==  if(j=1  :  % ! : data[i, j-1, k] !)
  data{neighbor[4]}[i,j,k] ==  if(j=ny:  % ! : data[i, j+1, k] !)
  data{neighbor[5]}[i,j,k] ==  if(k=1 :  % ! : data[i, j, k-1] !)
  data{neighbor[6]}[i,j,k] ==  if(k=nz: % ! : data[i, j, k+1] !)
```

Array Operations

Global dimensions{ index m, n, p, q }
Declarator mat[a,b] M = M[a,b],
thing(a,b) T = T{ real b[a] }[b]
Dynamic mat[m,n] a,b,   thing(p,q) Q, R

a = 0.
...
a = b + cos(a/2) + 1

Q = R

Q{b}[i] = Q{b}[i] + 1

Array Operations, continued

lobal dimensions{ index m, n, p, q }, subr(real a[3])
eclarator mat[n1,n2] M = M[n1,n2],
thing(n1,n2) T = T{ real b[n1] }[n2]
ynamic mat[m,n] a,b, thing(p,q) Q, R, index i,j

= [[i+j for i=1 to 10, i-j for i=11 to m] for j=1 to n]

[2,3] = [1,2,3,4, -i^2+1 for i=1 to 10] [p]

= [{ a[i,j]+b[i,j] for i=1 to p } for j=1 to q]

ubr([1., 2., 4.])