TREES: THE APL WAY

Aaron W. Hsu awhsu@Indiana.edu

PL Wonks Fall 2018, Indiana University

Viewer Discretion Advised:
May require suspension of disbelief.

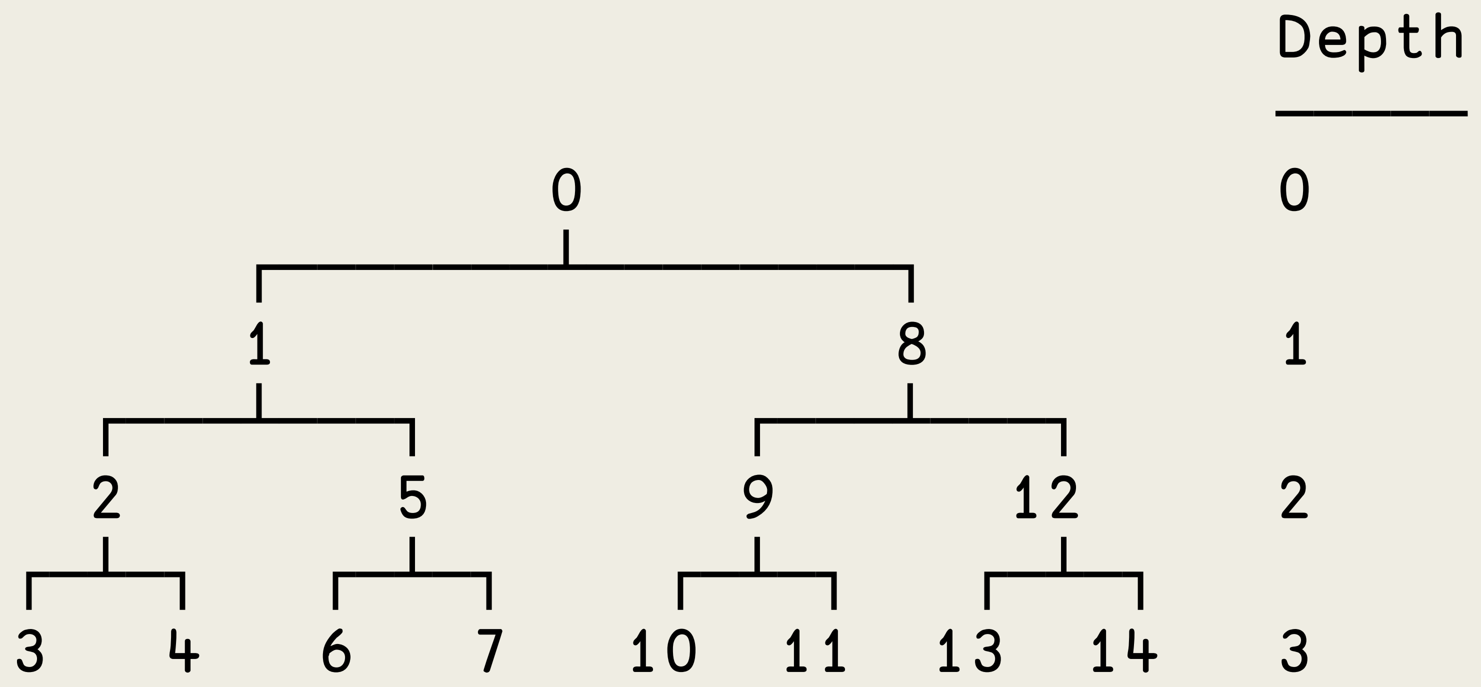
This talk is focused on the APL ecosystem, particularly Dyalog APL, and therefore contains claims that are specific to the realities of Dyalog APL's interpreted environment and current runtime implementation. Views expressed here may or may not directly translate effectively outside of the APL domain.

The Normal Way

- Top-down
- Recursive
- Branching
- Node-at-a-time

The APL Way

- Bottom-up
- One-liner Idioms
- Masking/Selection
- Global



```
tree   ←(0 (1 (2 3 4) (5 6 7)) (8 (9 10 11) (12 13 14)))
depth  ← 0  1  2 3 3   2 3 3    1  2  3  3    2  3  3
parent ← 0  0  1 2 2   1 5 5    0  8  9  9    8 12 12
left   ← 0  1  2 3 3   2 6 6    1  9 10 10    9 13 13
```

Nested

- Widely Taught
- Easy to recurse
- Easy to manipulate*

- Top-down Focused
- Complicated to Parallelize
- Non-trivial control flow
- Difficult to Idiomize
- Not space efficient
- Pointer-semantics Required
- Performance Penalties (APL)

Depth

- Highly space efficient
- Amenable to Recursion
- Simple to read/understand
- Very fast global traversals

- Write inefficiencies
- Generalized Parallelization may have bad complexity
- Some complex computations may be space inefficient
- Non-recursive solutions may be overly complex

Parent

- Write-optimized
- Many simple idioms
- Highly “tunable”
- Exceptionally concise code
- Simplest control flow
- SIMD for “Free”

- Not widely taught
- Not recursion friendly
- Non-termination
- Idiomatic Discipline
- Pointers

Example Tree: Quadratic Formula

```
:Namespace quadratic

    quadf←{
        a←0⌷⍵
        b←1⌷⍵
        c←2⌷⍵
        ((-b)+¯1 1×((b*2)-4×a×c)*÷2)÷2×a
    }

:EndNamespace
```

Core Tree Operations

Adding (,←)

Deleting (⊢-1+⍸)

Pushing Down/Lifting Up (p[]←…)

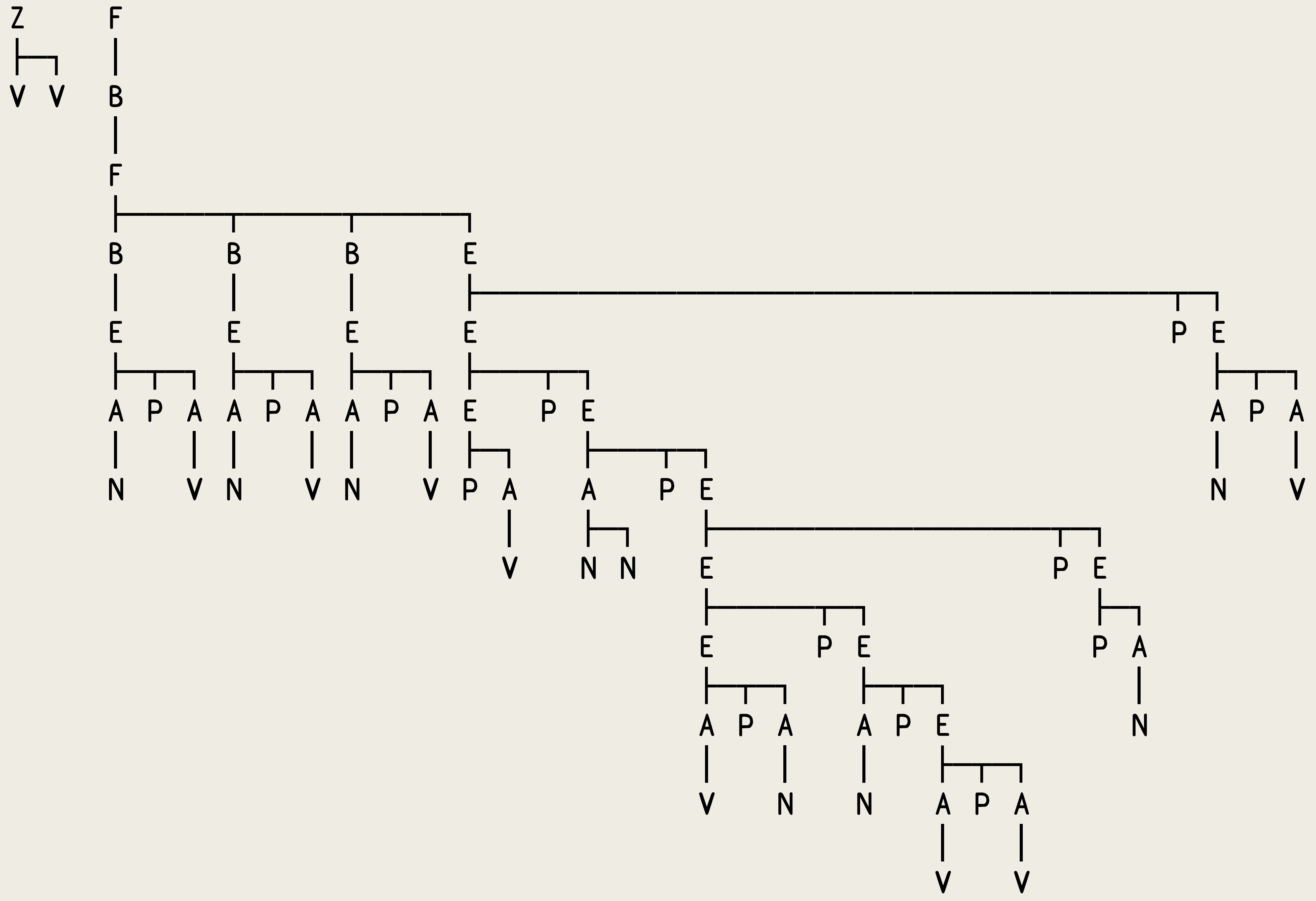
Inserting (l[]←…)

Updating (n[]←…)

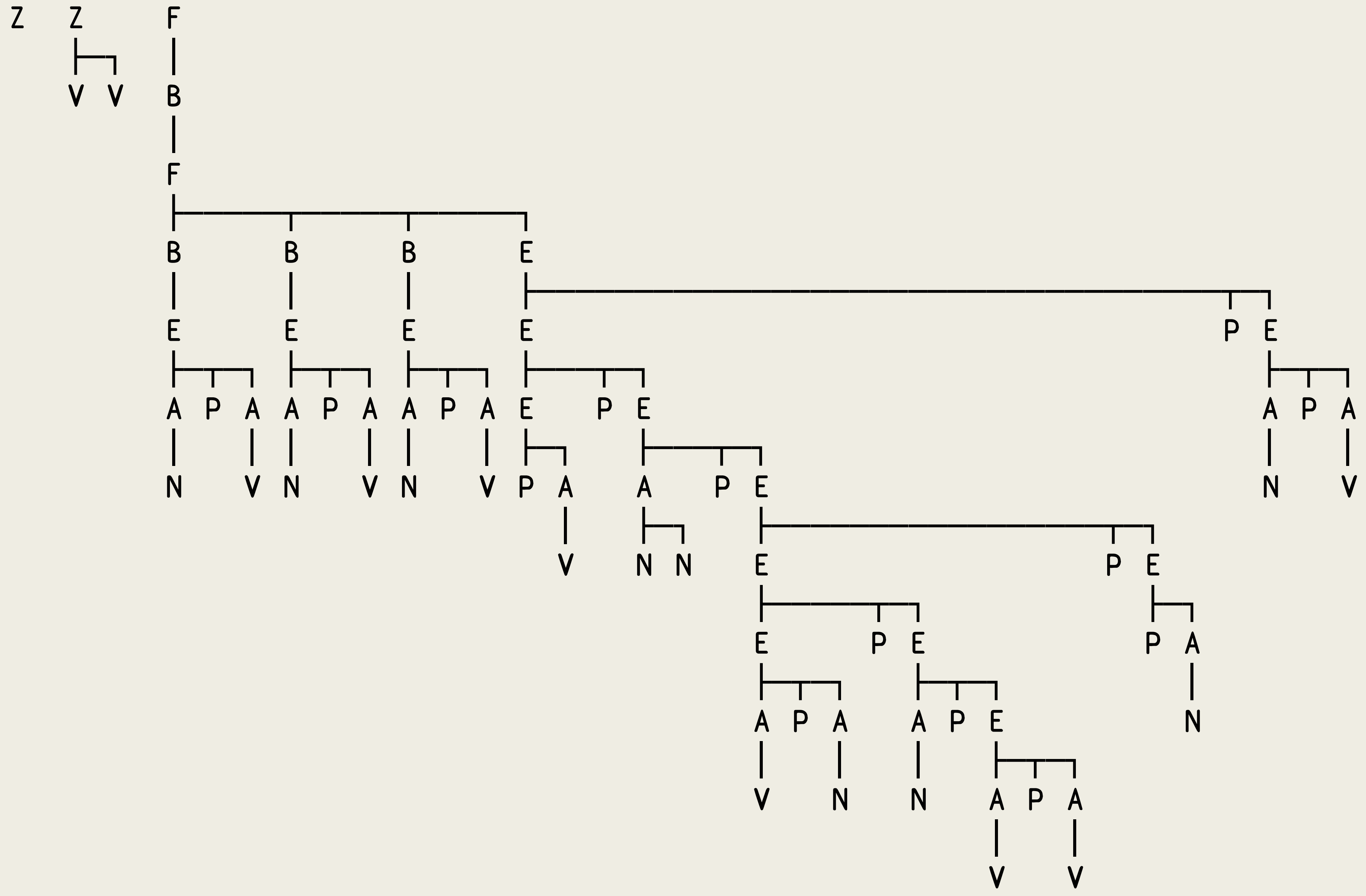
The Passes

```
⍝ Init Flags [ADD]
i←⍸(t=3)∧p=⍳s←≢p ⋄ p,←Δ←s+⍳≢i ⋄ l,←¯1⍴⍨≢i ⋄ t k,←11 2⍴⍨¨≢i ⋄ n,←i
l[Δ,⍸(p=⍳≢p)∧l=⍳≢l]←(⊃Δ),Δ

⍝ Lift Functions [LIFT, INSERT, ADD]
i←⍸(t=3)∧p≠⍳s←≢p ⋄ l←i(s+⍳)@{⍵∊i}l ⋄ p l(⊣,I)←⊂i ⋄ t k,←10 1⍴⍨¨≢i ⋄ n,←i
p[i]←i ⋄ l[j]←⊃(⌽i),j←⍸(p=⍳≢p)∧l=⍳≢l ⋄ l[i]←(≢i)↑(⊃i),i

⍝ Wrap Return Expressions [DOWN, ADD]
i←⍸(t[p]∊3 4)∧(t∊0 2)∨(t=1)∧(k=0)∧~(⍳≢l)∊¯1@{⍵=⍳≢⍵}l ⋄ p,←p[i]
p[i]←(≢l)+⍳≢i ⋄ l←i((≢l)+⍳)@{⍵∊i}l ⋄ l,←l[i] ⋄ l[i]←i ⋄ t k n,←2 0 0⍴⍨¨≢i

⍝ Lift Expressions [LIFT, INSERT, ADD, TRAVERSE]
i←⍸(t∊8,⍳3)∧m←t[p]≠3 ⋄ xw←x@(l[x])⊢x@(x←⍸m)⊢l ⋄ l←i((≢l)+⍳)@{⍵∊i}l
p,←Δ←p[i] ⋄ l,←l[i] ⋄ t,←10⍴⍨≢i ⋄ k,←(8∘=∨k[i]∧1∘=)t[i] ⋄ n,←i
l[∪Δ]←Δ⊢∘⊃⌸i ⋄ net←{~t[⍵]∊8,⍳5} ⋄ wk←xw I p∘I@(xw∘I=⊢)⍣≡
l[j]←((j×⊢)+×∘~)∘net⍨wk@net⍣≡wk⊢j←i~Δ ⋄ p[i]←p I@(3≠t∘I)⍣≡Δ
```

Starting Tree



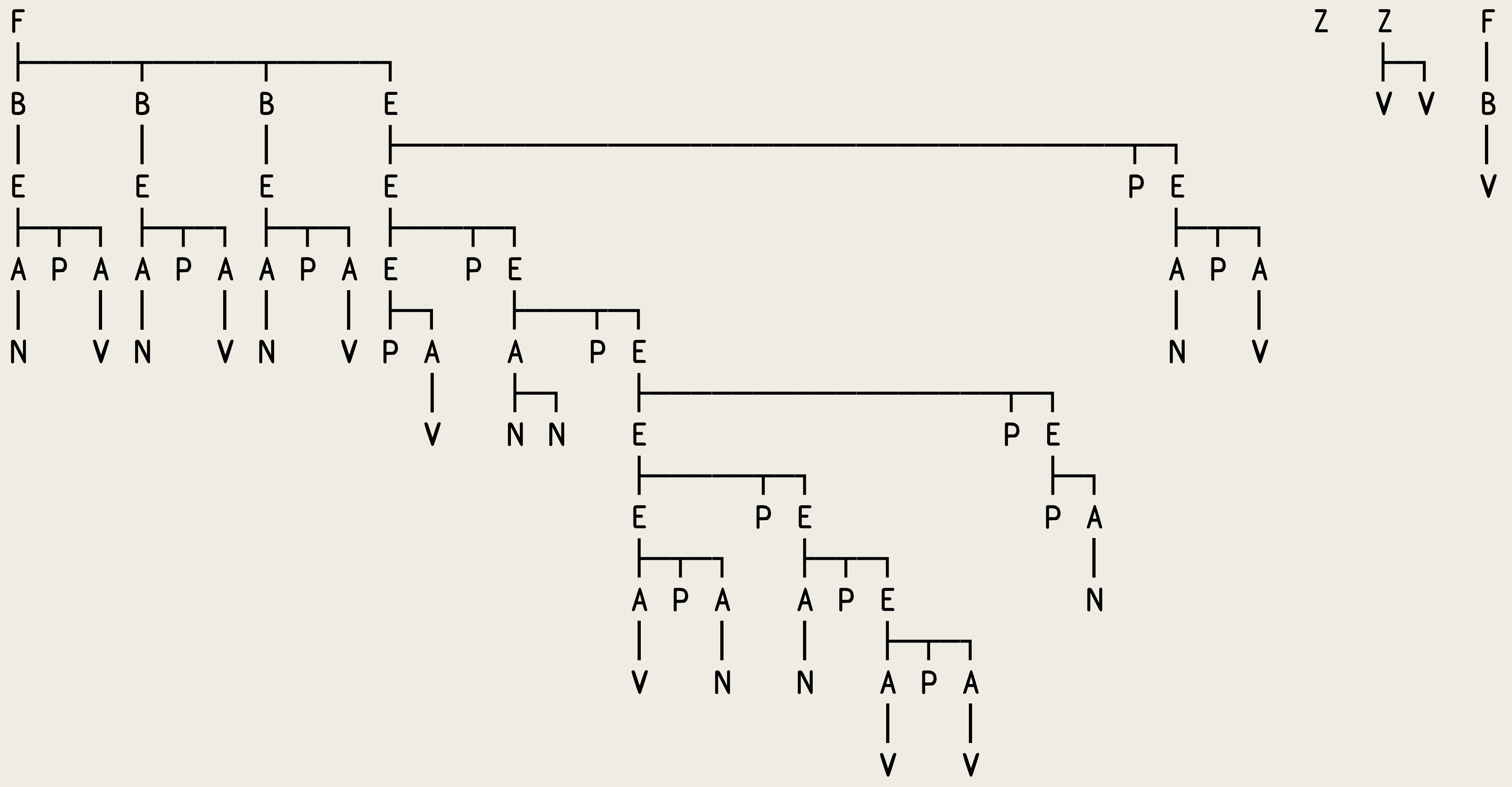
Init Functions



Init Functions

```
                           ⍝ Init Flags [ADD]
i←⍸(t=3)∧p=⍳s←≢p           ⍝ IDs of top-level functions
p,←Δ←s+⍳≢i                 ⍝ New nodes are root nodes
l,←¯1⍴⍨≢i                  ⍝ Prep siblings with empty value
t k,←11 2⍴⍨¨≢i             ⍝ Nodes are (11,2)
n,←i                       ⍝ Refer to top-level functions
l[Δ,⍸(p=⍳≢p)∧l=⍳≢l]←(⊃Δ),Δ ⍝ Link to yourselves and first node
```

Function Lifting



Lift Functions

```
                              ⍝ Lift Functions [LIFT, INSERT, ADD]
i←⍸(t=3)∧p≠⍳s←≢p              ⍝ All non-top-level functions
l←i(s+⍳)@{⍵∊i}l               ⍝ Right siblings → new reference variable
p,←p[i] ⋄ l,←l[i]             ⍝ New variables the place of function nodes
t k,←10 1⍴⍨¨≢i                ⍝ Function variable type (10,1)
n,←i                          ⍝ New variables refer to their functions
p[i]←i                        ⍝ Functions are lifted to the top-level
l[j]←⊃(⌽i),j←⍸(p=⍳≢p)∧l=⍳≢l   ⍝ Functions are placed in front of their
l[i]←(≢i)↑(⊃i),i              ⍝   top-level module nodes in the AST
```

Wrapping Returns

```
F
├─────┬─────┬─────┐
B     B     B     E
│     │     │     │
E     E     E     E
├─┬─┐ ├─┬─┐ ├─┬─┐ ├───────────────────────────────────┬─┐
A P A A P A A P A E                                   P E
│   │ │   │ │   │ ├───┬─┐                               ├─┬─┐
N   V N   V N   V E   P E                               A P A
                  ├─┐   ├───┬─┐                         │   │
                  P A   A   P E                         N   V
                    │   ├─┐   ├─────────────────┬─┐
                    V   N N   E                 P E
                              ├─────┬─┐           ├─┐
                              E     P E           P A
                              ├─┬─┐   ├─┬─┐         │
                              A P A   A P E         N
                              │   │   │   ├─┬─┐
                              V   N   N   A P A
                                          │   │
                                          V   V
```

```
Z  Z    F
   ├─┐  │
   V V  B
        │
        V
```

Wrapping Return Expressions

```
                                ⍝ Wrap Return Expressions [DOWN, ADD]
i←⍸(t[p]∊3 4)…                  ⍝ Ret. Exps. Have function or guard parents
 ∧(t∊0 2)                       ⍝ and are either atomics or expressions
  ∨(t=1)                        ⍝ or they are binding nodes
   ∧(k=0)                       ⍝ that bind values
    ∧~(⍳≢l)∊¯1@{⍵=⍳≢⍵}l         ⍝ and are the last expr of a function/guard
p,←p[i]                         ⍝ Enclosing return is at same place as expr.
p[i]←(≢l)+⍳≢i                   ⍝ and they enclose the ret. Exprs.
l←i((≢l)+⍳)@{⍵∊i}l              ⍝ Ret. Exps. Right siblings point to new nodes
l,←l[i]                         ⍝ inserted where ret. Exps. were
l[i]←i                          ⍝ Ret. Exps. Are now single children
t k n,←2 0 0⍴⍨¨≢i               ⍝ of type (2,0) return nodes
```

Lifting Expressions/Remove-complex-opera*

```
F
├─┬─┬─────┬─┬─┬─┬─────┬─┬─┬─┬─────┬─┬─┬─┬─────┬─┬───┬─┬─┬─────┬─┬─────┬─┬─┬─────┬─────┬─────┬───┬─────┬─┬───┬─────┬─────┐
A A E     B A A E     B A A E     B A A E     A E   A A E     A E     A A E     E     E     A   E     A E   E     E     E
│ │ ├─┬─┐ │ │ │ ├─┬─┐ │ │ │ ├─┬─┐ │ │ │ ├─┬─┐ │ ├─┐ │ │ ├─┬─┐ │ ├─┬─┐ │ │ ├─┬─┐ ├─┬─┐ ├─┬─┐ ├─┐ ├─┬─┐ │ ├─┐ ├─┬─┐ ├─┬─┐ │
V N V P V V V N V P V V V N V P V V V N V P V N P V V V V P V N V P V N V V P V V P V V P V N N V P V V P V V P V V P V V

Z  Z    F
   ├─┐  │
   V V  B
        │
        V
```

Lifting Expressions

```
                                       ⍝ Lift Expressions
i←⍸(t∊8,⍳3)∧m←t[p]≠3                   ⍝ All expressions w/o function parents
xw←x@(l[x])⊢x@(x←⍸m)⊢l                 ⍝ An “execution walk” path
l←i((≢l)+⍳)@{⍵∊i}l                     ⍝ Point right siblings to new variables
p,←∆←p[i]                              ⍝ New variables take the place of
l,←l[i]                                ⍝  the old expression nodes
t,←10⍴⍨≢i                              ⍝ And are variables
k,←(8∘=∨k[i]∧1∘=)t[i]                  ⍝  of type 1 or 0 depending on the Expr
n,←i                                   ⍝ New variables refer to expr nodes
l[∪∆]←∆⊢∘⊃⌸i                           ⍝ Exprs go just above their parents
net←{~t[⍵]∊8,⍳5}                       ⍝ Predicate: not expression type
wk←xw I p∘I@(xw∘I=⊢)⍣≡                 ⍝ Fn: find new sibling
l[j]←((j×⊢)+×∘~)∘net⍨wk@net⍣≡wk⊢j←i~∆ ⍝ Leaf exprs need distant siblings
p[i]←p I@(3≠t∘I)⍣≡∆                    ⍝ Expr parents are now nearest function
```

Parent/Sibling Style

- Idiomatic
- Effectively zero interpreter overhead
- Strong leveraging of optimized APL primitives and idioms
- Allows for the use of APL primitives as “tree primitives”
- Correlates core tree manipulations with core APL operations
- Allows for arbitrary tree reordering for optimization, tuning, and simpler code
- Allows for verifying/proving correctness without complex recursion theory
- Scales to SIMD architectures like the GPU (Co-dfns)
- Allows for very concise, compact code
- Easier to “optimize” from a human and compiler standpoint

Thank you.

http://www.sacrideo.us
https://www.patreon.com/arcfide
https://github.com/Co-dfns