Do Programming Language Features Deliver on their Promises?

Aaron W. Hsu aaron@dyalog.com, Dyalog, Ltd.

LambdaConf 2025, Estes Park, CO

I am Biased

Ascetical Programming

Seen and been near legacy, work in greenfield research

Spent time in a variety of programming communities

Unique Perspective

“Design” part of programming languages

Developer Experience

Path of Least Resistance

Path of Feel Goodiness

Exploratory, not Dogmatic.
And mostly meant to be fun and comedic.

But the best comics speak the truth, no?

Rich Hickey. 2011. “Simple Made Easy”. StrangeLoop.

Simple vs. Complex ≡ Disentangled vs. Entangled

“Simplicity is prerequisite for reliability.” – Edsger Dijkstra

Simple, not Easy.

Ultimate Goal: Simplicity

Traditional Software Crisis ≡ “Unmanaged complexity”

Modern Software Crisis ≡ “Readily accessible complexity”

What are the promises?

Did it meet those promises?

What were the unintended effects?

Does it tend towards encouraging incidental complexity?

Can we reframe its use to encourage simplicity?

Development Methods, Praxis, Architecture:

Unit Testing

Microservices

Test-driven Development

REST API

Low-hanging Fruit:

Object-orientation
Inheritance
Encapsulation

Rage bait:
Many functional programmers are championing
Object-oriented Programming under the guise of FP.

Encapsulation, Inheritance, Indirection and Abstraction.

“The lie is that if something is object-oriented,
it will be easier for someone to integrate,
because it is all encapsulated.

But the truth is the opposite.”

-- Casey Muratori, Wookash Podcast, Sep 22, 2024. YT

“I also must confess to a strong bias against the fashion for reusable code. To me, "re-editable code" is much, much better than an untouchable black box or toolkit. I could go on and on about this. If you’re totally convinced that reusable code is wonderful, I probably won’t be able to sway you anyway, but you’ll never convince me that reusable code isn’t mostly a menace.”

-- Donald Knuth, InformIT Interview. Apr 25, 2008.

Token AI Slide:

Natural Language Programming

Present-day AI is very good at writing code
that never should have been written in the first place.

The value of AI for writing code is inversely proportional
to the quality of the programming language.

Just joking! But not really.

Tacit/Points-free Programming

What’s the promise?

[T]he strict use of composition results in programs that are well adapted for equational reasoning. … The lack of argument naming gives point-free style a reputation of being unnecessarily obscure, hence the epithet "pointless style".
-- Tacit programming - Wikipedia

“The sum of a collection of numbers divided by the count of numbers in the collection.”

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

(/ (fold-right + 0 elems) (length elems))

+⌿÷≢

More concise!

Removes unnecessary names

More compositional

Less boilerplate

“Simpler expressions”

```apl
scp←(1,1↓Fm)⊂[0]⊢
prf←((≢↑¯1↓(0≠⊢)(/∘⊢)⊢)⍤1↑∘r)⊢
blg←{⍺←⊢ ⋄ ⍺((prf(⌈/(⍳∘≢⊢)×⍤1(1↓⊣)∧.(=∨0=⊢)∘⍉⊢)⍺⍺(⌿∘↑)r)⌷⍤0 2 ⍵⍵(⌿⍨)⍺⍺)⍵}
enc←⊂⊣,∘⊃((⊣,'_',⊢)/(⊂''),(⍕¨(0≠⊢)(/∘⊢)⊢))
veo←∪((⊂'%u'),(,¨prims),⊣)~⍨∘{⊃,/{⊂⍣(1≡≡⍵)⊢⍵}¨⍵}¯1↓⊢(/⍨)(∧/¨0≠((⊃0⍴⊢)¨⊢))
ndo←{⍺←⊢ ⋄ m⊃∘(⊂,⊢)¨⍺∘⍺⍺¨¨⍵⊃∘(,∘⊂⍨⊂)¨⍨m←1≥≡¨⍵}
n2f←(⊃,/)((1=≡)⊃,∘⊂⍨∘⊂)¨

rn←⊢,∘↓(1+d)↑⍤¯1(+⍀d∘.=∘⍳1+(⌈/0,d))
rd←⊢,(+/↑∘r∧.(=∨0=⊢)∘⍉∘↑∘r⊢(⌿⍨)Fm∧1∊⍨k)
df←⊢(⌿⍨)(+\1=d)(~⊣∊⊣(/⍨)(1=d)∧(~'b'∊⍨k)∧Om∨Fm)⊢
dua←((~Gm)∧Fm∨↓∘prf∊r∘Fs)(⊣(⍀∘⊢)(⊣(⌿∘⊢)0∊⍨n)(0,1↓(¯1⌽⊣)∧⊢=¯1⌽⊢)⊣(⌿∘⊢)d)⊢
du←(~dua∨(∨/(prf∧.(=∨0=⊢)∘⍉dua(⌿∘⊢)prf)∧↑∘r∧.≥∘⍉dua(⌿∘⊢)↑∘r×0=prf))(⌿∘⊢)⊢
lfh←(0≠1⌷⊣)⊃(⊂∘⍉∘⍪0'M'0 '',0,⍨(⊂⊣)),∘⊂∘⍉∘⍪1'F'1,('fn'enc⊣),(⊂⊣),5↓∘,1↑⊢
lfn←(d,'Of',3↓⊢)⍤1 at(Fm∧'b'∊⍨k)(d,'Vf',('fn'enc∘⊃r),4↓⊢)⍤1 at(Fm∧1∊⍨k)
lf←(⊃⍪/)(1,1↓Fm∧1∊⍨k)blg(↑r)(⊂lfh⍪∘(((⊢-(⊃-2⌊⊃))d),1↓⍤1⊢)lfn)⌸1↓⊢
dn←((0∊⍨n)∧(Am∧'v'∊⍨k)∨Om∧'f'∊⍨k)((~⊣)(⌿∘⊢)(d-¯1⌽⊣),1↓[1]⊢)⊢
mrep←(1+⊃),'P'0(,'⊢'),(⊂''),⍨¯1↓4↓∘,1↑⊢
mreu←⊃,'E' 'u',(⊂''),⍨¯1↓3↓∘,1↑⊢
mre←(⊃⍪/)(-∘⊃Vm∨Am)∘⊃∘⌽(↓,(((⊢⍴⍨(≢⍉),⍨≢×2<≢)mreu⍪mrep⍪(1+d),1↓⍤1⊢)¨↑))⊢
mrs←⊢⊂[0]⍨1,1↓d=1+∘⊃d
mrk←(-∘(+/∧\)∘⌽Lm)(↑⍪⍨∘(mre(mre mrs)¨at(Gm∘(⊃⍪/)1↑¨⊢)∘mrs)↓)⊢
mr←(⊃⍪/)((1↑⊢),(mrk¨1↓⊢))∘scp
ur←((2↑⊢),1,('um'enc∘⊃r),4↓⊢)⍤1at(Em∧'u'∊⍨k)
rt←⊢,(∨\Fm)+(+⌿prf∧.(=∨0=⊣)∘⍉⍨∘↑∘r Ms⍪Gs)-Fm
nm←((3↑⊢),('fe'enc∘⊃r),4↓⊢)⍤1 at((0∊⍨n)∧Em∨Om∨Am)
lgg←(⍪/1↓⊢)⍪∘⊃⍨⊣(((¯1+d),2,⍨t,k,n,r,∘⍪s)⍪⊣⍪3,'V','a',3(↓⍤1)1↑⊢)∘⊃1↑⊢
lg←(⊃⍪/)⊢((⊂⊣(⌿⍨∘~)(∨\⊢)),(((1↑⊢)lgg⊢⊂[0]⍨d=1+⊃)¨⊂[0]⍨))Gm∧1⌽Em
fet←(d,'V'0,3↓⊢)⍤1 at(0,1↓Em∨Om∨Am)(d,'Av',3↓⊢)⍤1 at(Em∧'b'∊⍨k)
fee←(⍪/⌽)(Mm∨Em∨Om∨Am)blg⊢((⊃∘⌽⊢)(⊂(d--⍨∘⊃),1↓⍤1⊢)∘fet⊣⍪¯1↓⍤1⊢)⌸⊃,⍨1↓⊢
fe←(⊃⍪/)(+⍀d≤g)(⊂(⊢↑⍨1=∘≢⊢)⍪∘⊃∘fee⊢)⌸⊢
can←(+\Am∨Om)((,1↑⊢),∘(⊂(¯1+2⌊≢)⊃(⊂∘⊂⊃),⊂)∘n 1↓⊢)⌸⊢
cas←(¯1⌽(Am∨Om)∧'vf'∊⍨k)∨(↓prf)∊∘r⊢(⌿⍨)Am∧'n'∊⍨k
ca←(can⊢(⌿⍨)cas∨Am∨Om∧'f'∊⍨k)⊣at(Am∨Om∧'f'∊⍨k)⍬,∘⊂⍨⊢(⌿⍨∘~)cas
lj←(⊃⍪/)(1↑scp),((⊢⍪2'L'0 0,2'',⍨¯2↓4↓∘,1↑⊢)¨1↓scp)
sd←(⊃⍪/)(1↑scp),(n Fs)(d,'Vf',(⊂⊣),4↓⊢)⍤1 at((⊂,'∇')∊⍨n)¨1↓scp
inm←∨⌿¯1(⌽∨⊢)1 2(⌽∨⊢)(¯1 ¯2⌽Em∧[1]1 2∘.=k)∧⍤1Vm∧n∊∘n Fs
inp←(Em∧⊣)∨1,2≠/⊣
inza←(1↑1↓⊣)(⌿⍨∘≢)at((⊂,'⍺')∊⍨n)(¯1↑⊣)(⌿⍨∘≢)at((⊂,'⍵')∊⍨n)⊢
inz←(1↑⊣)(d,t,k,3↓⍤1(⌿⍨∘≢))at(0,⍨2≠/∘⌽(∨\∘⌽Em))inza
inn←(3↑⍤1⊢),((⊣⍴¨⍨1+0⌈(⌈/∘n Gs))(('fe'≡2↑⊢)⊃(⊂⊢),∘⊂'fe',(⍕⊣),2↓⊢)¨n),(4↓⍤1⊢)
ins←⊣(d,t,k,((1000×1+⊣)+1+n+(⌈/n)),4↓⍤1⊢)at(Lm∨Gm)inn
inr←1,∘⍪⊢inz¨(⍳∘≢⊢)ins¨((⊃∘n¨⊣)⍳((⊃n(⌿⍨)Vm∧'f'∊⍨k)¨⊢))⊃¨(⊂1↓¨⊣),∘⊂¨⊢
in←(⊃⍪/)∘(⊢/)(1↓scp)inr∘((0⍴⊂0 8⍴0),⊢/)at(⊣/)inm((⊃¨inp⊂Em∧⊣),∘⍪inp⊂[0]⊢)⊢
```

```apl
pcc←(⊂⊢(⌿⍨)Am∨Om∧'f'∊⍨k)∘((⍳∘∪⍨n)⌷⍤0 2(1⌈≢)↑⊢)∘(⊃⍪⌿)∘⌽(⌿∘⊢)
pcb←((,∧.(=∨0=⊣)∘⍪)⍤2 1⍨∘↑∘r Ms⍪Fs)pcc⍤1((⊢(⌿⍨)(d=g)∧Am∨Em∨Om)¨scp)
pcv←(d,'V',('af'⊃¨∘⊂⍨Om),(⊃¨v),r,s,(⊂⍬),⍨∘⍪g)at(Om∨Am∧'v'∊⍨k)
pc←(⊃⍪/)pcb{(pcv d(⊣,1↓⍤1⊢)(⍺↑⍨1⌈≢⍺)⌷⍤0 2⍨(n⍺)⍳n)at(Vm∧(n⍺)∊⍨n)⍵}¨scp
da←⊢(⌿⍨∘~)(Am∧d=g)∨(0,⍨2∧/Lm)∨(Lm∧¯1⌽Am∧d=g)∨Om∧('f'∊⍨k)∧1≠d
fce←(⊃∘n Ps){⊂⍕' ⍵',⍨(≢⍵)⊃''(⍺,'⊃')('⊃',⍺,'/')⊣⍵}(∨ As)
fcm←(∧/Em∨Am∨Pm)∧'u'≢∘⊃∘⊃k
fc←((⊃⍪/)(((d,'An',3↓¯1↓,)1↑⊢),fce)¨at(fcm¨))('MFOEL'∊⍨t)⊂[0]⊢
ce←(+\Fm∨Gm∨Em∨Om∨Lm)((¯1↓∘,1↑⊢),∘⊂(⊃∘v 1↑⊢),∘(Am⊃¨∘↓n,∘⍪∘n2f v)1↓⊢)⌸⊢
ll←(⊢(⌿⍨)1⌽Lm)(((⊂⊂'%l'),∘⊂¨∘n⊣),⍨¯1↓⍤1⊢)at Lm⊢
fv←(⊃⍪/)(((1↓⊢)⍪⍨(,1 7↑⊢),∘⊂∘n ¯1↑⊢)¨scp)
nvi←((¯1↓⊢),({,¨⍺'['⍵}/∘⊃v))⍤1at((Em∨Om)∧'i'∊⍨k)
nvv←(⊂'%u' '%f' '%u'),(⊂'%u' '%i',⊢),(⊂(⊂'%u'),⊢)
nv←(¯1↓⍤1((2↑⊢),2,(3↓⊢))⍤1at((Em∨Om)∧'i'∊⍨k)),((¯1⊖≢⊃nvv,∘⊂⊢)¨v∘nvi)
lt←(⊂⍬),⍨⊢
val←(n⍳∘∪n),¨⊢(⊢+(≢⊣)×0=⊢)(⌈/(⍳≢)×⍤1(∪n)∘.((⊂⊣)∊⊢)(n2f¨v))
vag←∧∘~∘(∘.=⍨∘⍳≢)⍨(∘.(((1⌷⊢)>0⌷⊣)∧(0⌷⊢)<1⌷⊣)⍨val)
vae←(∪n)(⊣,⍤0⊣(⌷⍨⍤1 0)∘⊃((⊢,(⊃(⍳∘≢⊣)~((≢⊢)↑⊣)(/∘⊢)⊢))/∘⌽(⊂⍬),∘↓⊢))vag
vac←(((0⌷∘⍉⊣)⍳∘⊂⊢)⊃(1⌷∘⍉⊣),∘⊂⊢)ndo
va←((⊃⍪/)(1↑⊢),(((vae Es)(d,t,k,(⊣vac n),r,s,g,y,∘⍪⍨(⊂⊣)vac¨v)⊢)¨1↓⊢))scp
avb←{(((,¨'⍺⍵')↑⍨1↓⍴)⍪⊢)⍺⌷⍨⍤2 0⊢⍺⍺⍳⍺⍺n⍨(↓(⌽1+∘⍳0⍳⍨⊢)((≢⊢)↑↑)⍤0 1⊢)⊃r ⍵}
avi←¯1 0+(⍴⊣)⊤(,⊣)⍳(⊂⊢)
avh←{⊂⍵,(n⍵)((⍺⍺(⍵⍵ avb)⍵){⍺⍺ avi ndo(⊂⍺),⍵})¨v⍵}
av←(⊃⍪/)(+\Fm){⍺((⍺((∪∘⌽(0⍴⊂''),n)Es)⌸⍵)avh(r(1↑⍵)⍪Fs ⍵))⌸⍵}⊢
rlf←(⌽↓(((1⊃⊣)∪⊢~0⌷⊣)/∘⌽(⊂⍬),↑)⍤0 1⍨1+∘⍳≢)(⊖1⊖n,⍤0(⊂⊣)veo¨v)
rl←⊢,∘(⊃,/)(⊂∘n Os⍪Fs)rlf¨scp
vc←(⊃⍪/)(((1↓⊢)⍪⍨(1 7↑⊢),(≢∘∪∘n Es),1 ¯3↑⊢)¨scp)
eff←(⊃⍪/)⊢(((⊂∘⍉∘⍪d,'Fe',3↓,)1↑⊣),1↓⊢)(d=∘⊃d)⊂[0]⊢
ef←(Fm∧¯1=∘×∘⊃¨y)((⊃⍪/)(⊂⊢(⌿⍨)∘~(∨\⊣)),(eff¨⊂[0]))⊢
ifn←1 'F' 0 'Init' ⍬ 0 1,(4⍴0) ⍬ ⍬,⍨⊢
if←(1↑⊢)⍪(⊢(⌿⍨)Om∧1=d)⍪((⊢wrap⍨∘ifn∘≢∘∪n)⊢(⌿⍨)Em∧1=d)⍪(∨\Fm)(⌿∘⊢)⊢
fgz←(1↑⊢)⍪(((¯1+d),1↓⍤1⊢)1↓⊢)⍪2,'G',1,3↓⍤1(¯1↑¯1↓⍤1⊢),∘n 1↑⊢
fg←(⊃⍪/)(fgz¨at(Gm∘(⊃⍪/)1↑¨⊢)⊢⊂[0]⍨d=2⌊g)
fft←(,1↑⊢)(1 'Z',(2↓¯5↓⊣),(∨⊣),n,y,(⊂2↑∘,∘⊃∘⊃e),l)(¯1↑Es)
ff←((⊃⍪/)(1↑⊢),(((1↑⊢)⍪(((¯1+d),1↓⍤1⊢)1↓⊢)⍪fft)¨1↓⊢))scp
fzh←((∪n)∩(⊃∘l⊣))(¯1⌽(⊂⊣),((≢⊢)-1+(⌽n)⍳⊣)((⊂⊣⊃¨∘⊂(⊃¨e)),(⊂⊣⊃¨∘⊂(⊃¨y)),∘⊂⊣)⊢)⊢
fzf←0≠(≢∘⍴¨∘⊃∘v⊣)
fzb←(((⊃∘v⊣)(⌿⍨)fzf),n),∘⍪('f'∘,∘⍕¨∘⍳(+/fzf)),('s'∘,∘⍕¨∘⍳∘≢⊢)
fzv←((⊂⊣)(⊖↑)⍨¨(≢⊣)(-+∘⍳⊢)(≢⊢))((⊢,⍨1⌷∘⍉⊣)⌷⍨(0⌷∘⍉⊣)⍳⊢)⍤2 0¨v
fze←(¯1+d),t,k,fzb((⊢/(-∘≢⊢)↑⊣),r,s,g,fzv,y,e,∘⍪l)⊢
fzs←(,1↑⊢)(1⊖(⊣((1 'Y',(2⌷⊣),⊢)⍪∘⍉∘⍪(3↑⊣),⊢)1⌽fzh,¯1↓6↓⊣)⍪fze)(⌿∘⊢)
fz←((⊃⍪/)(1↑⊢),(((2=d)(fzs⍪(1↓∘~⊣)(⌿∘⊢)1↓⊢)⊢)¨1↓⊢))(1,1↓Sm)⊂[0]⊢
fd←(1↑⊢)⍪((1,'Fd',3↓⊢)⍤1 Fs)⍪1↓⊢

tta←(fc∘da∘(pc⍣≡)∘mr⍣≡)∘in⍣3∘sd∘lj∘ca∘fe∘lg∘nm∘rt∘mr∘dn∘lf∘du∘df∘rd∘rn
tt←fd∘fz∘ff∘fg∘if∘ef∘vc∘rl∘av∘va∘lt∘nv∘fv∘ll∘ce∘ur∘tta
```

More concise!	→	But caused duplication
Removes unnecessary names	→	But introduced new ones
More compositional	→	For compositions of one
Less boilerplate	→	With more control flow
“Simpler expressions”	→	Substituted complexity

Incidental Complexity:
Encourages introducing more names,
just what it was supposed to reduce!

Reframing:
A valuable syntax for reducing total points of the system,
but not at the expense of introducing new names.

At scale, points-free by de-naming

```
⍝ Lift Lambdas to the top-level
p,←n[i]←(≢p)+⍳≢i←⍸(t=F)∧p≠⍳≢p ⍝ New nodes at top-level
t k n lx mu pos end r(⊣,I)←⊂I  ⍝ Copy data to new nodes
p r I⍨←⊂n[i]@i⊢⍳≢p             ⍝ Point fn bodies @ top nodes
t[i]←V                         ⍝ Old nodes become variables
k[i]←3+5 11⍸k[i]               ⍝ …that point to lifted fns
```

You can reduce names without needing special syntax!

DSLs/Syntactic Abstraction

Programming Languages are not a good fit for our domain

→

Write specialized domain knowledge behind a DSL

→

Profit…?

Non-programmers won’t need programming!

Declarative solutions! Bespoke?

The tool is always ideally suited to the job!

“Right tool for the right job” is now a reality.

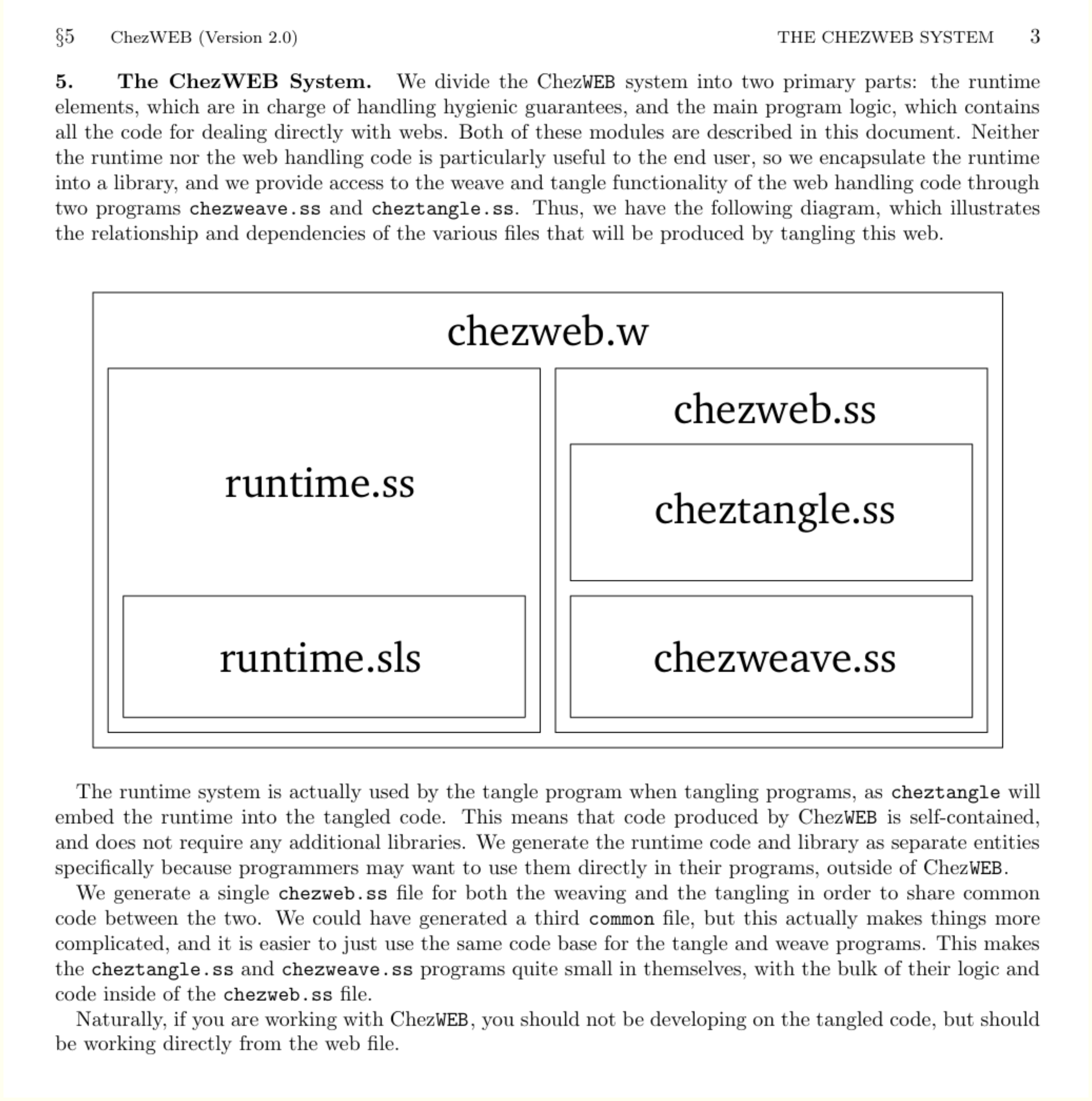
There’s a library for that….

PyTorch
NumPy
BLAS
SQL
ORMs
HTML
SAP
Excel
Leftpad

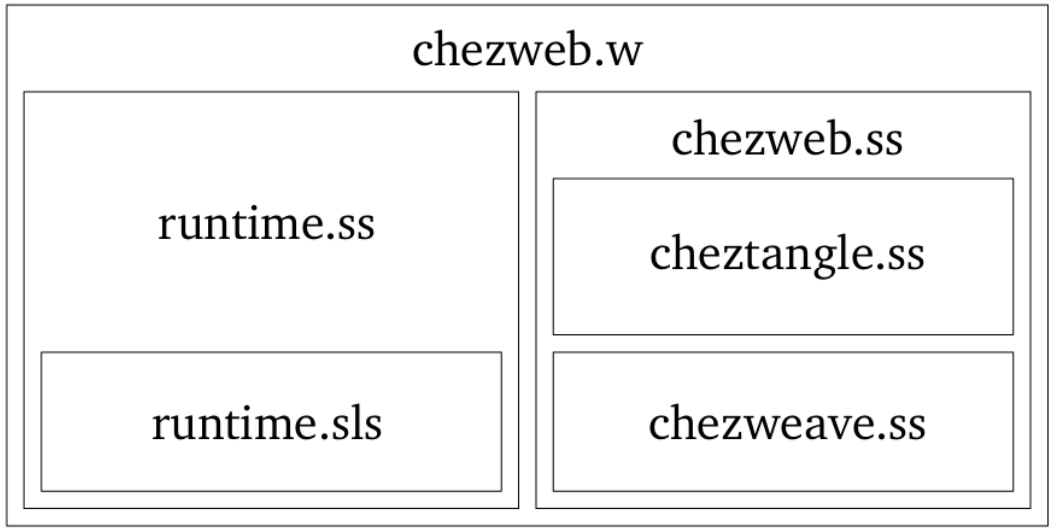
Leftpad

```
(⊃⍛⌊∘≢)↓(⊃⍴⌿)⍛⍪
```

Hygienic Literate Programming. ChezWEB. Scheme Workshop 2011.



5. The ChezWEB System. We divide the ChezWEB system into two primary parts: the runtime elements, which are in charge of handling hygienic guarantees, and the main program logic, which contains all the code for dealing directly with webs. Both of these modules are described in this document. Neither the runtime nor the web handling code is particularly useful to the end user, so we encapsulate the runtime into a library, and we provide access to the weave and tangle functionality of the web handling code through two programs `chezweave.ss` and `cheztangle.ss`. Thus, we have the following diagram, which illustrates the relationship and dependencies of the various files that will be produced by tangling this web.



The runtime system is actually used by the tangle program when tangling programs, as `cheztangle` will embed the runtime into the tangled code. This means that code produced by ChezWEB is self-contained, and does not require any additional libraries. We generate the runtime code and library as separate entities specifically because programmers may want to use them directly in their programs, outside of ChezWEB.

We generate a single `chezweb.ss` file for both the weaving and the tangling in order to share common code between the two. We could have generated a third `common` file, but this actually makes things more complicated, and it is easier to just use the same code base for the tangle and weave programs. This makes the `cheztangle.ss` and `chezweave.ss` programs quite small in themselves, with the bulk of their logic and code inside of the `chezweb.ss` file.

Naturally, if you are working with ChezWEB, you should not be developing on the tangled code, but should be working directly from the web file.

Scheme, the language for writing DSLs

→

No one speaks the same language.

Every sufficiently popular DSL becomes a poorly designed general purpose programming language (c.f. TeX)

Lesson: Tower of Babel

Collaboration and Focused Effort should transfer, not isolate

DSLs are an appeal to Jargon’s advantage,
ignoring its disadvantages

Jargon obscures common knowledge transfer more than we realized

The user is always limited by and held hostage by the implementor

Back to the beginning, what was the justification?

Language/Domain Mismatch

Design languages to address problems
concisely, directly, and simply.

Focus on holistic economy of expression as a metric.

Stop trying to hide the magic, reframe your core concepts/constructs.

Example:

Parallel programming hard.

Introduce iteration combinators (map, fold, scan) and a data-flow style of programming as the “first language”.

Result: Parallel programming easy.

c.f. – CMU’s experiments with 1st year programming education

Caveat: I still really love syntactic abstraction,
I just can’t really justify it most places.

Dynamic Languages + Reflection
(Thanks Steven W.)

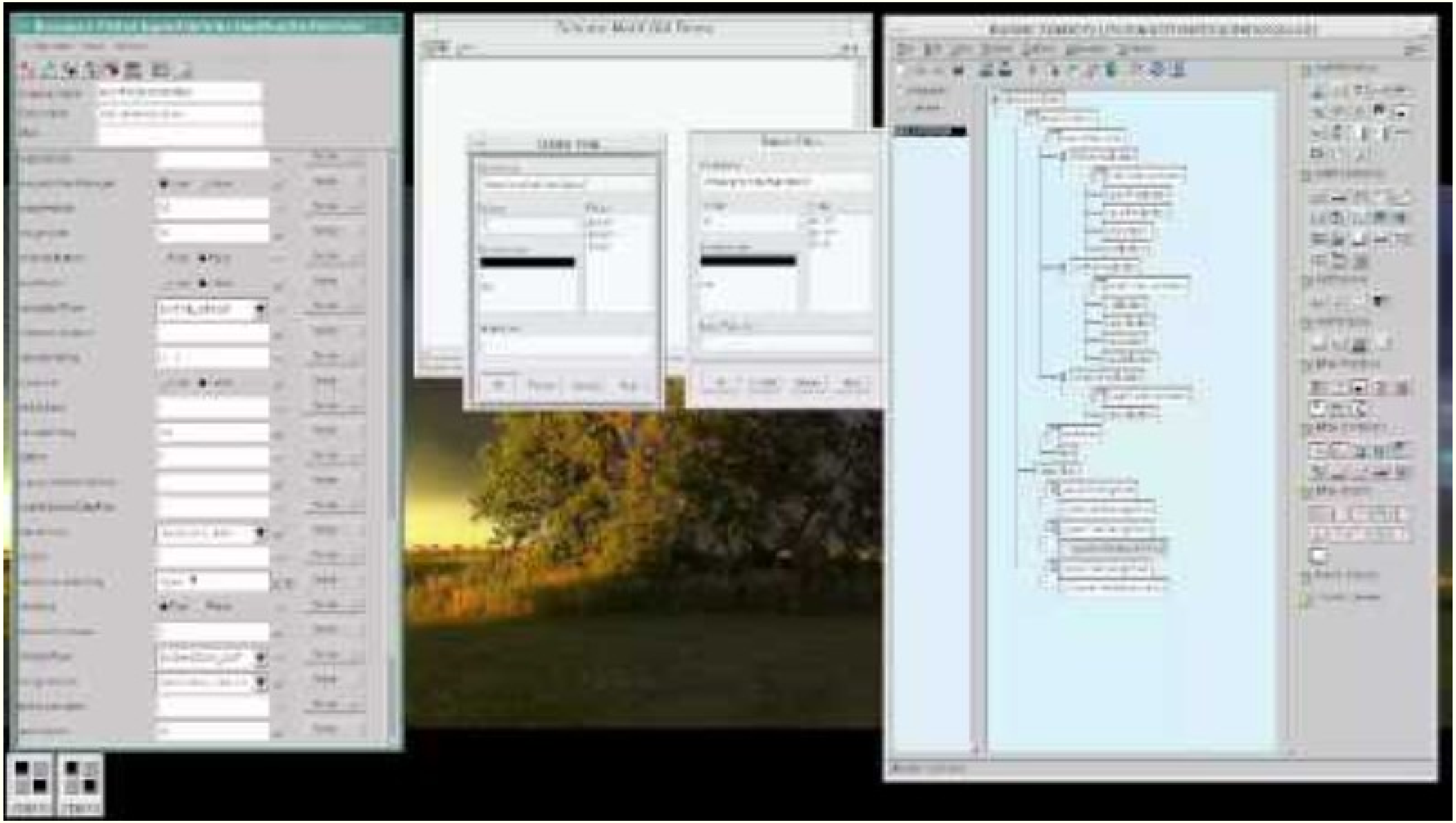
Superior Tooling Support

Monkey patching

Meta Programming

Agility/Adaptation

Richer behaviors



Ruby Compiler

→

Pain, tears, and abandonment

Endless and intractable uses of reflection

SATAN: Static Analysis Tool for APL Notation
(Brandon Wilson)

APL is highly dynamic:
Type-sensitive parsing, dynamic/lexical scope, eval-everywhere

Real World APL code is “statically dynamic”.

Reflection often encourages Indirection/Abstraction

Nothing is ever where you think it is
(paraphrasing Ed Amsden)

Most of the time, you don’t want it, you don’t need it, you shouldn’t use it.

But…

I can’t fully dismiss the value of tools and introspection.

Read-only introspection is hugely useful.

Mutable reflection is highly questionable.

There is often a spiritually static way to leverage dynamic features.

Avoid indirection.

One of my favorite tools:

The computed goto.

Sorry, not sorry.

New Feature:

24× more memory
59× more code
3× more concepts to learn
9× slower on the CPU
56x slower on the GPU

(Actually, you can’t run your code on the GPU with this new feature. So, the slowdown is infinite.)

Hooray!

Generalized Pointers/Garbage Collection

C pointers have a nice connection with array indexing.

$\rightarrow$

Higher level languages abstracted pointers into opaque references, uncomputable things that encouraged data-abstraction via structures of pointers

Singular object management with malloc/free was error-prone.

GC is framed as a solution to this problem.

We already have good solutions to this without GC!

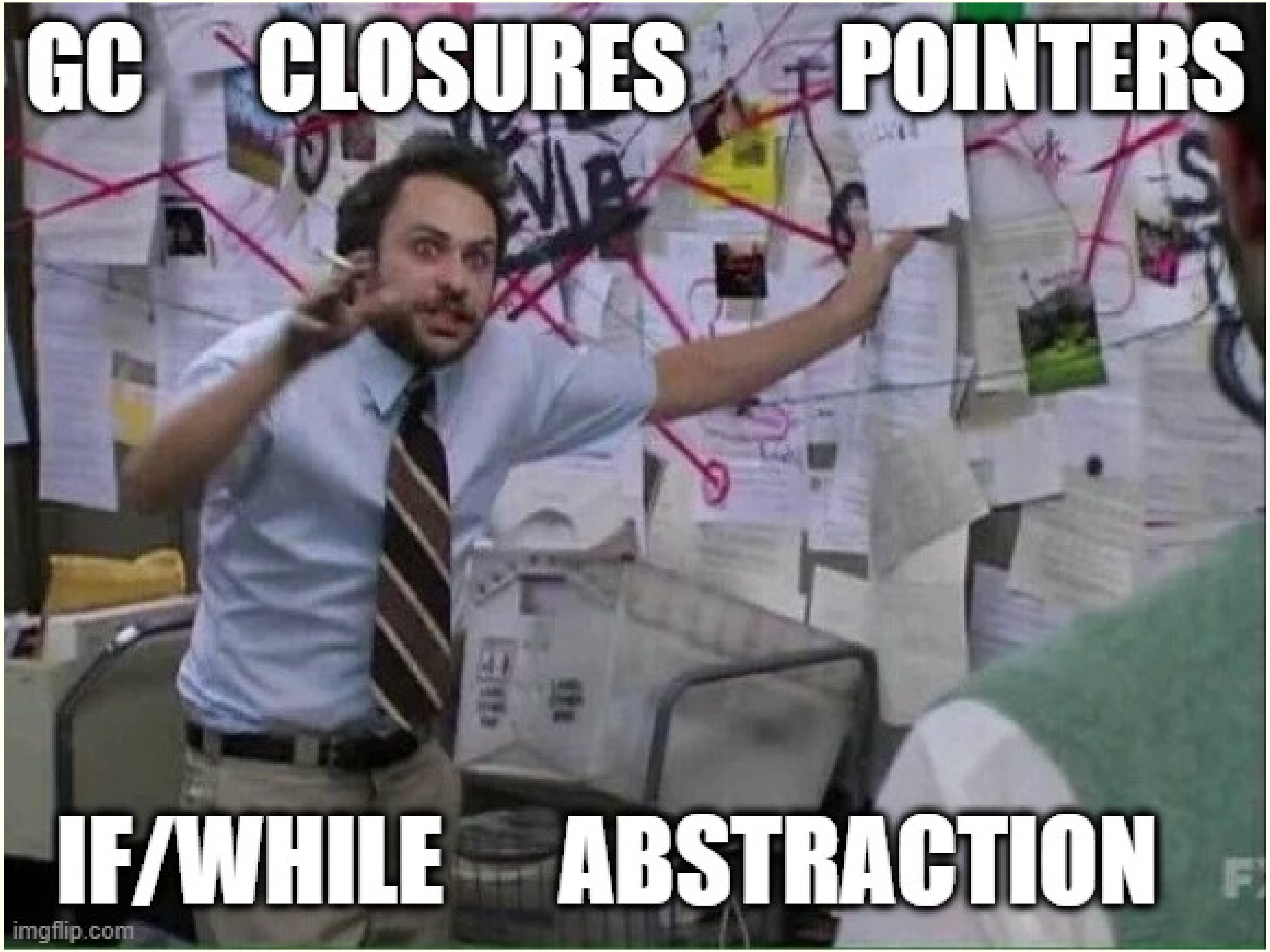
GC is actually the story of

Generalized Pointers + Infinite Lifetimes

First-class Procedures + Structured Programming

$\leftarrow\rightarrow$

Generalized Pointers + Infinite Lifetimes


GC
CLOSURES
POINTERS
IF/WHILE
ABSTRACTION
imgflip.com

Am I crazy?
“Nodejs memory leak”

Stop Node.js Memory Leaks - End-to-end Tracing + Logging **AD**

Preventing and Debugging Memory Leaks in Node.js

node.js - Detecting memory leaks in nodejs - Stack Overflow

Advanced Techniques for Detecting Memory Leaks in Node.js ...

Finding And Fixing Node.js Memory Leaks: A Practical Guide - Marmelab

Stop relying on indefinite extent
Stop using constructors of boxed data
Stop defaulting to encapsulation and closures

Make (app/biz/logic) state global and highly controlled
Manual memory management is easy in a high-level language

Favor relational, table, entity-component,
collection-based data organization
using Struct of Arrays encodings (APL’s Inverted Table)

Sophisticated Control Flow
Recursion
Polymorphic dispatch
Callback-driven event systems
Agent-features

AKA – Structured Programming and its evil spawn.

“Goto considered harmful.”

No more spaghetti code.

Simpler code – it even has less arbitrary labels!

So foundational, people don’t even consider whether it should exist or not.

“All problems in computer science
can be solved by another level of indirection,
except for the problem of too many layers of indirection.”
– David J. Wheeler

“It is easier to move a problem around…
than it is to solve it.”
-- RFC 1925

+/ιn	
+/⌽ιn	+ is associative and commutative
((+/ιn)+(+/⌽ιn))÷2	(x+x)÷2←→x
(+/((ιn)+(⌽ιn)))÷2	+ is associative and commutative
(+/((n+1)ρn))÷2	Lemma
((n+1)×n)÷2	Definition of ×

-- Notation as a Tool of Thought,
Ken Iverson, ACM Turing Award Lecture

Compare proof logics (c.f. Dijkstra) for structured programs

Recursive programs (ACL2)

Higher-order dependently typed programs

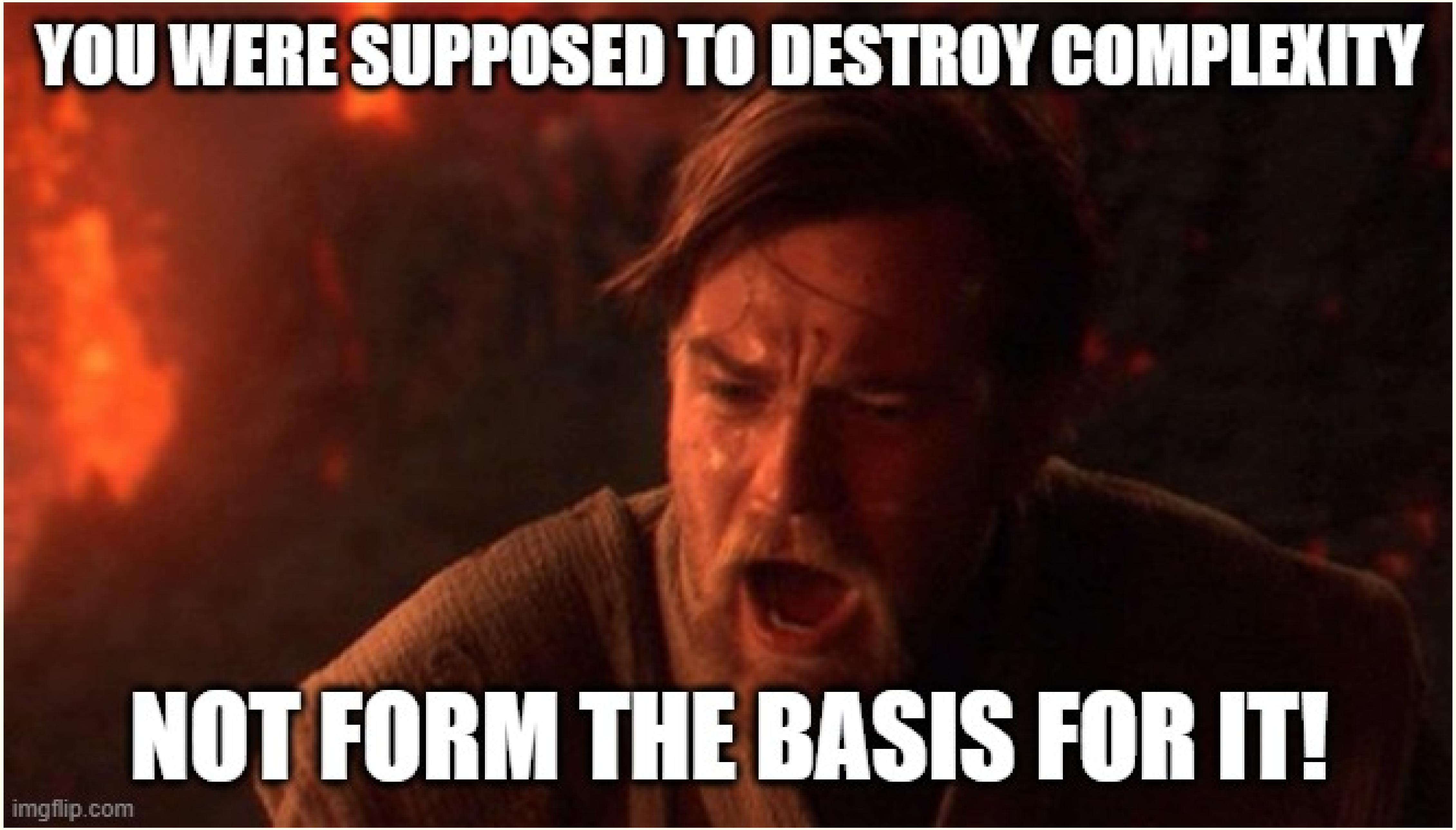
These are all fundamentally more complex.

Branching is responsible for CPUs having branch predictors,
undermining some of the most elegant CPU designs

Every branch, jump, and indirect dispatch
introduces more edges into an increasingly complex graph

In EWD249, Dijkstra especially points out the value
of constraining and disciplining control flow.

The whole model of structured programming was intended
to reduce control flow complexity.


YOU WERE SUPPOSED TO DESTROY COMPLEXITY
NOT FORM THE BASIS FOR IT!
imgflip.com

Structured programming encourages single-item thinking

Single-item thinking encourages casing, specialization (types)

This incentivizes all the high-level branch and dispatch

Which is just fancier goto programming that Dijkstra was avoiding!

Dynamically paged, eval-dependent, goto-driven APL
developed over the course of 30+ years

Eliminate non-linear, irregular control flow

Combinators (map, fold, scan, key, stencil, etc.)

Data-parallel programming
Data-driven, data-oriented design

Compute over collections/sets/arrays, not individual items

Avoid traditional type abstraction

See tomorrow’s talk for my favorite approach to handling inherently branchy, event-driven code

Static typing gets a pass…

Because I already ranted about it for a whole talk:
Does APL Need a Type System? at #FnConf18

But…maybe just a taste?

Static typing as I’ve heard it
has sometimes claimed elegance, but never brutal simplicity.

The story I’ve always heard is…

“We’ll make things harder on you initially,
so the machine can help you in the future.”

The code has always been more complex.

“Scrap your boilerplate”
Ralf Lammel and Simon Peyton Jones

+

Multiple follow-up papers

But…static typing has mostly delivered
on its technical promises.

It has failed to show that the trade-off is the right direction.

Conspiracy Theory: Types are a great way to make money off of PL,
particularly if you want to make money off of research grants.

Static type annotations have a generally complecting influence.

Rigorous proof systems and implicit type inference don’t suffer the same hinderances to the code quality (simplicity).

But…systems like Rust have had a positive impact
on people willing to take more control over their own code again.

Think about the types, but design your system to make them obvious architecturally and in your code

Don’t obscure code with static type annotations

Stop doing excessive type abstraction and attempting to hide implementation details
(Yes, that includes you, dynamically typed folks)

Not Yet Ripe:

Formal Methods

“80% of development is spent on plumbing”
(Isaac Shapira quoting someone else)

Irony:

I avoid using all the features designed to avoid the plumbing.

But I end up writing no plumbing.

Example at “scale”

Compiler building frameworks vs Co-dfns:

A data parallel compiler hosted on the GPU.
Aaron Hsu. 2019. Indiana University.

A near-zero abstraction compiler written in APL for APL.

Now also does (context-sensitive, type-dependent) parsing
in the same style.

1500 vs. (50,000 – 100,000)

“[O]ne of my central themes will be that any two things that differ in some respect by a factor of already a hundred or more,
are utterly incomparable.”
-- Edsger Dijkstra, EWD249

Conclusion

arcfide@sacrideo.us

Featuritis is a real thing in language design
Avoid language features that inherently introduce complexity

The use of language features is often a crutch
to avoid simplification or tackling harder root problems

https://www.sacrideo.us/last-minute-discount-for-apl-workshop/