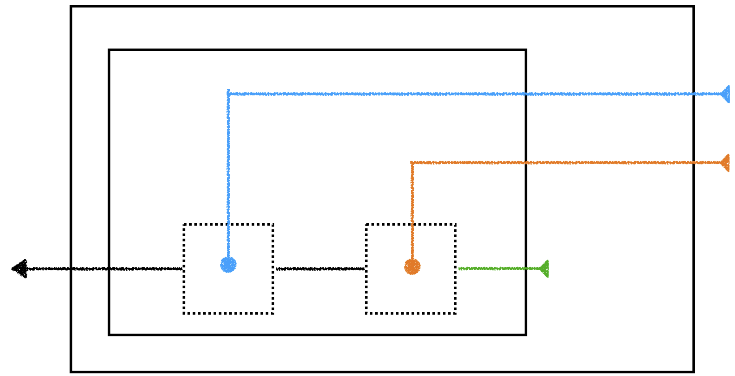
λ-Circuit : A Graphical Language for Functional Programming

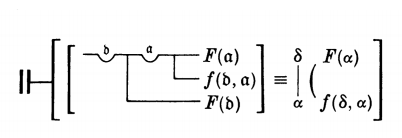


Chenchao Ding, Oct 26, 2023

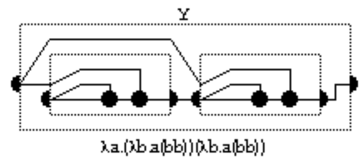
Powered by GNU $\mathrm{T_EX_{MACS}}$

Outline

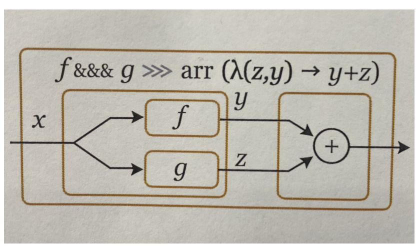
There is a long history about diagrams...



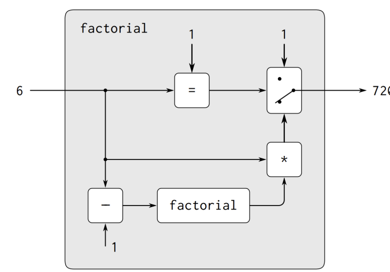
Begriffsschrift (Frege, 1879)



Graphical (Keenan, 1996)



Haskellian (1998 - modern+)

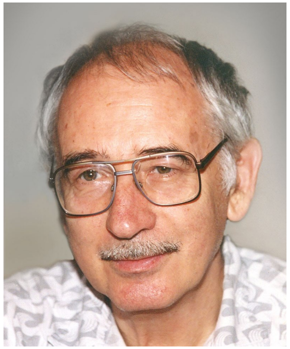


SICP (1984)

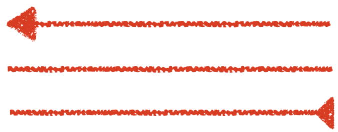
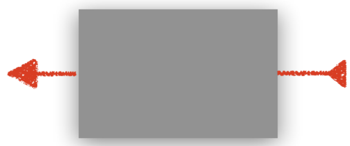
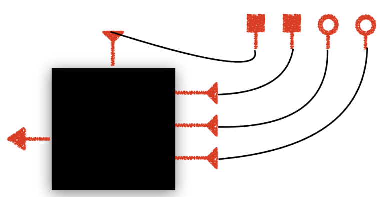
A program is seen as a machine. To make sense of it, one must observe its operation.

— Valentin Turchin, *The Concept of a Supercompiler*, 1986

Valentin Fyodorovich Turchin (1931-2010) was a Soviet and American physicist, cybernetician, and computer scientist. He developed the Refal programming language, the theory of meta-system transitions and the notion of supercompilation.

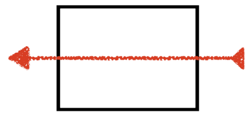


This work is mainly inspired by Yin Wang's scattered teaching, and based upon *Programming Languages as Notations* [?] by Chelsea Sierra Voss (csvoss).

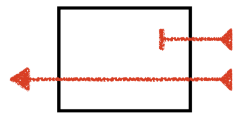
Idealized Circuits		**Lambda Calculus**
	$\Leftrightarrow$	$x, y, z, \ldots$
wires / super-conductors		variables
	$\Leftrightarrow$	$\lambda x.e, \lambda xy.e, \ldots$
circuit templates		abstractions
	$\Leftrightarrow$	$r\,e, r\,(e_1 \otimes e_2), \ldots$
wire connections		applications

Idealized Circuits

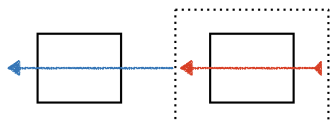
Lambda Calculus



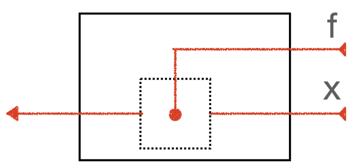
$\lambda x.x$



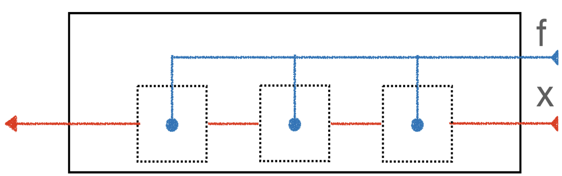
$\lambda xy.y$



$(\lambda x.x)(\lambda x.x)$



$\lambda fx.fx$



$\lambda fx.f(f(fx))$

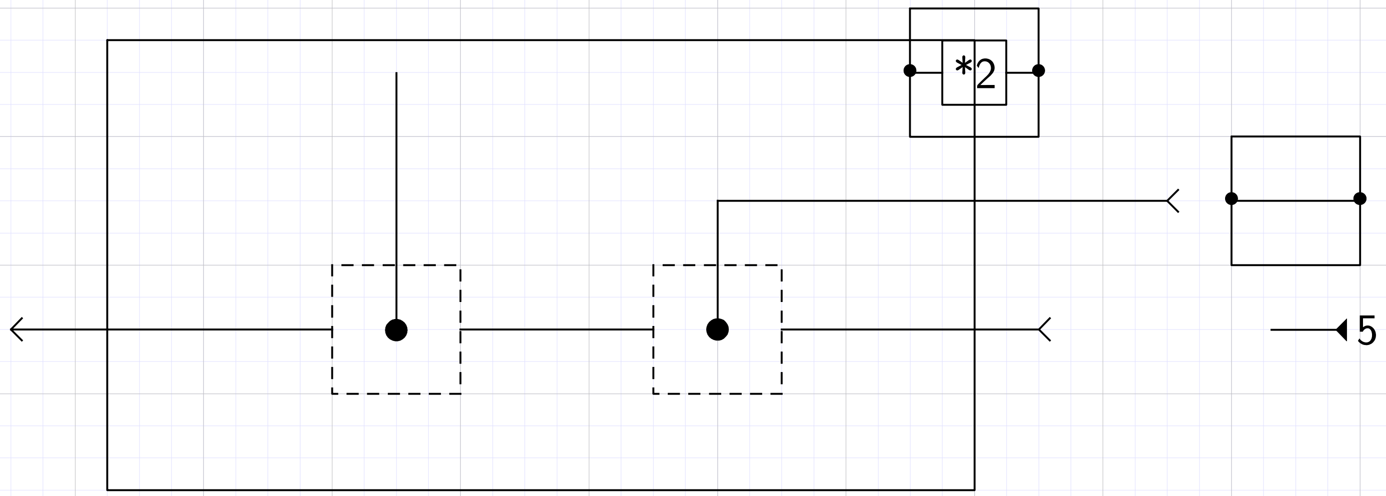
- electrons moving through wires $\Rightarrow$ quite intuitive...
- circuits moving through "wires" $\Rightarrow$ nonsensical

Wires are abstract trajectories in some spaces.

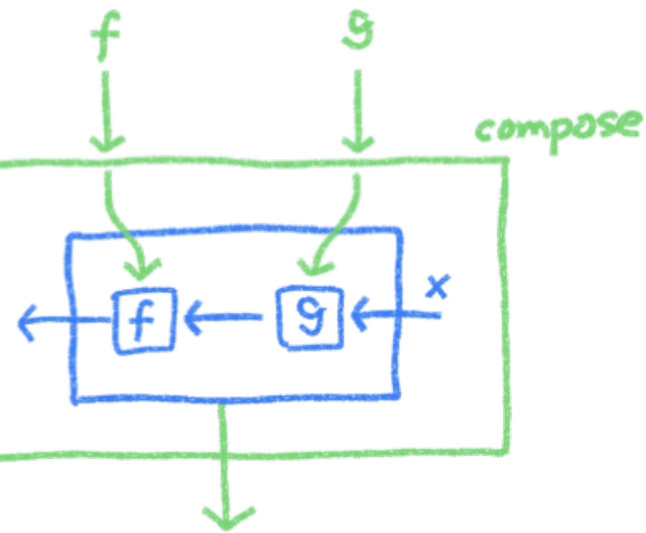
- circuits moving through "wires" $\Rightarrow$ to move a circuit from A place to B place
- there might be many different ways from A to B
- ideally we ignore the difference (lengths, twists, etc.) and focus only on two "endpoints"

Boxes sliding along wires doesn't change circuits.

$$\texttt{compose} = \lambda f g . \lambda x . f \bullet (g \bullet x)$$



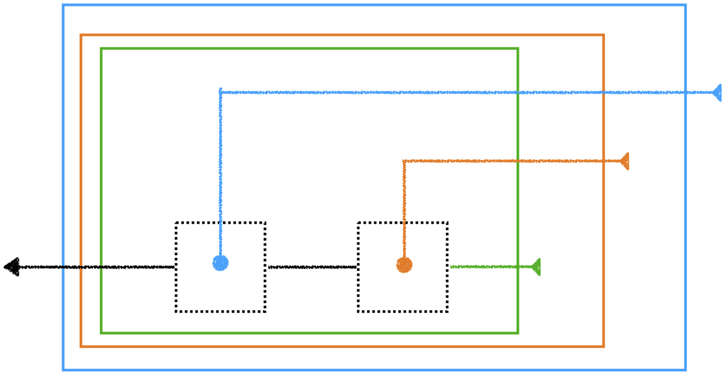
Here is a diagram made from Yin Wang:




```
compose = \f g -> (\x -> f (g x))
```

- theoretically the above diagram should be the compose circuit
- for convenience, we can align two output arrows into one so that it "pokes around"

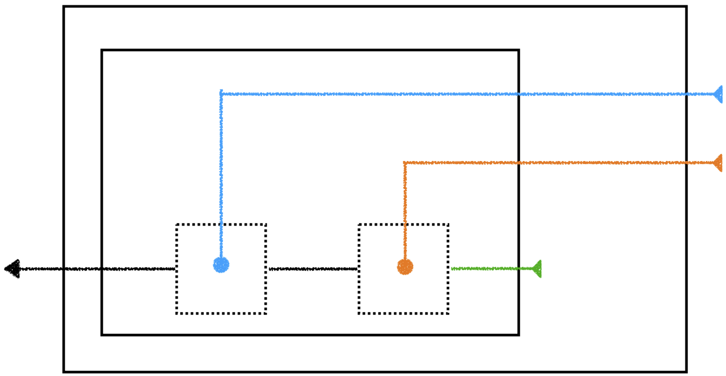
- Boxes are "scope delimiters"
- Pins (wires crossing boxes) are perfect binders



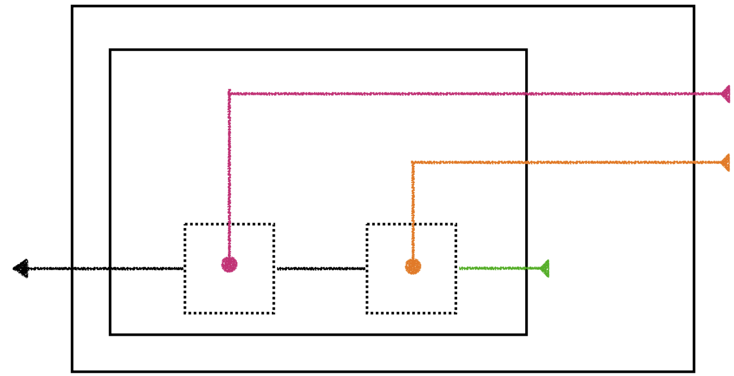
compose = (λ (λ (λ (3 (2 1)))))

There is no explicit and literal "λ" in λ-circuit - but wires and boxes preserves the "binding structures" and scopes.

$\Rightarrow$ same functions as deBruijn indices / numbers



$\overset{\alpha}{=}$



compose = (λ (f g) (λ (x) (f (g x))))

compose = (λ (h g) (λ (x) (h (g x))))

α-equivalence : the name/tag of a wire always remains consistent. (silent)

Free Variables & Scoping

What about functions with free variables?

```
let x = 4 in
  let f = \y -> x + y in
    let x = 2 in
      f 3
```

What about functions with free variables?

```
let x = 4 in
  let f = \y -> x + y in
    let x = 2 in
      f 3
```

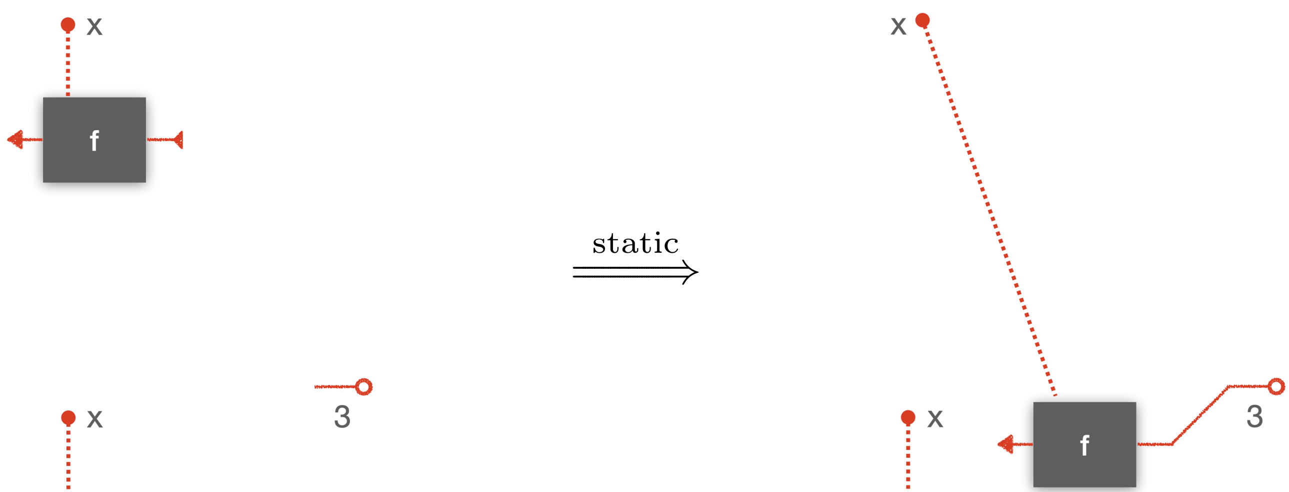
Free variables are **covert channels**

Analogy - WiFi (wireless connections):

- connect to the unique WiFi named "home" once initiated $\Rightarrow$ unconvenient but safe
- connect to a closest WiFi named "home" wherever you go $\Rightarrow$ convenient but dangerous

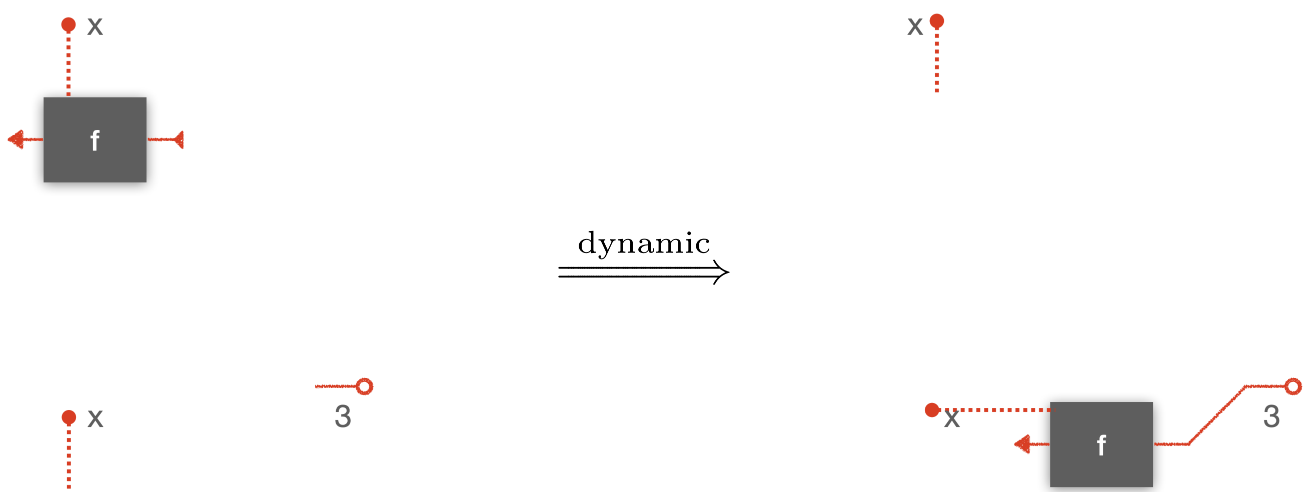
Free Variables & Scoping

```
let x = 4 in
  let f = \y -> x + y in
    let x = 2 in
      f 3
```

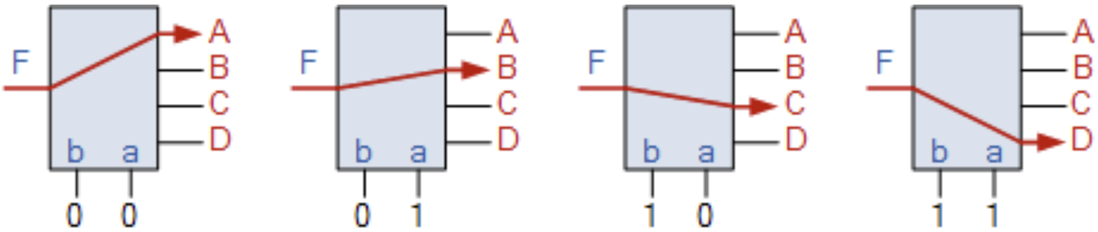


Free Variables & Scoping

```
let x = 4 in
  let f = \y -> x + y in
    let x = 2 in
      f 3
```

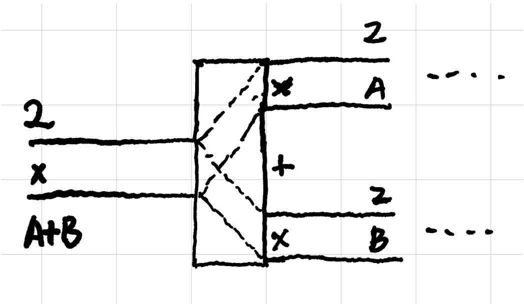


Conditionals are **demultiplexers** (DEMUX)



This can be encoded in a type isomorphism (distribution law):

$$2 \times (A + B) \;\simeq\; (2 \times A) + (2 \times B)$$



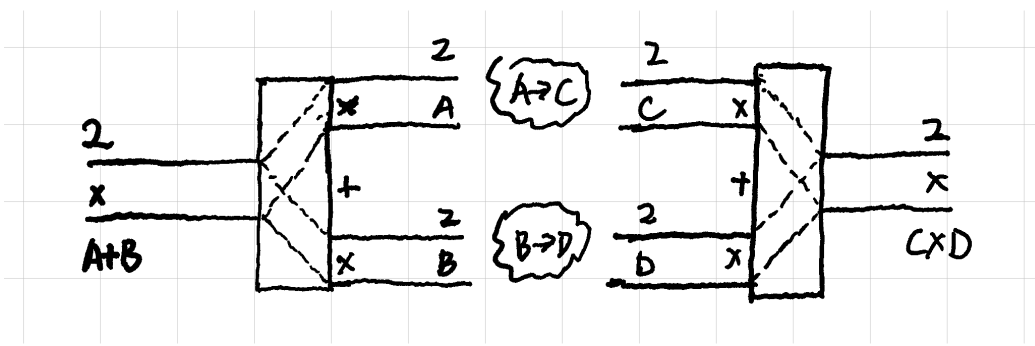
Joins are multiplexers (MUX)

Joins are implicit in our daily programming experience. They are explicit in static analysis and kind of explicit in logic programming. Consider the program below:

```
fact n = case n of
    0 -> 1
    n -> n * fact (n-1)
```

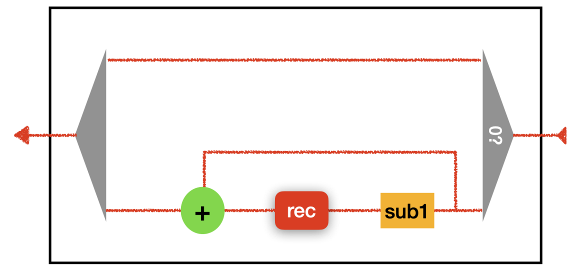
There are two possible outputs (one for each branch), but the number of output signal is 1.

$$(2 \times C) + (2 \times D) \ \simeq \ 2 \times (C + D)$$

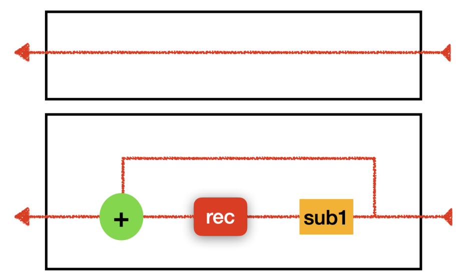


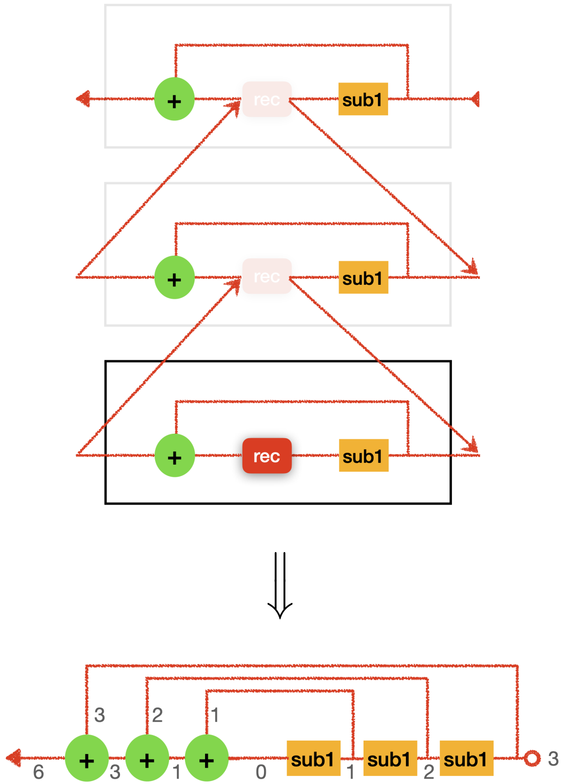
Recursion as Dynamic Circuits | sum


```
sum n = case n of
    0 -> 0
    n -> n + sum (sub1 n)
```




```
sum 0 = 0
sum n = n + sum (sub1 n)
```

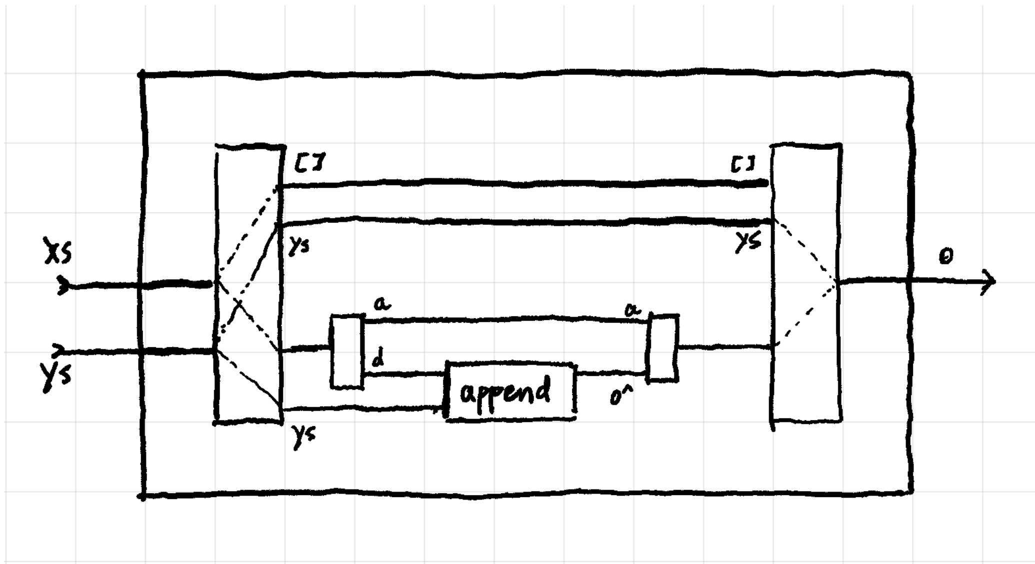




sum 3

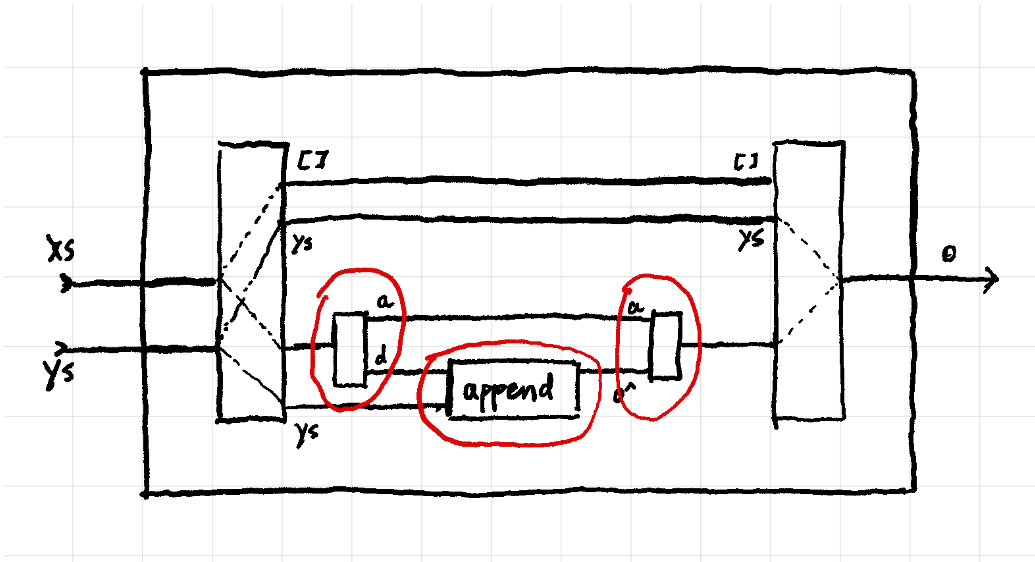
Calling `(sum 3)` will dynamically unfold (building) circuits through the `rec` endpoint. It looks like a stack. All values are stored and transmitted through wires.

Re-arrange the dynamically unfolded circuits…




```
append xs ys = case xs of
  []  -> ys
  a:d -> a:(append d ys)
```

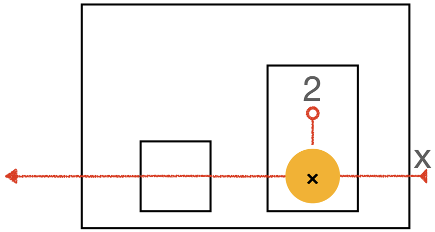
```
append xs ys = case xs of
  []  -> ys
  a:d -> let o^ = append d ys in
           a:o^
```




```
appendo [] ys o = [(== o ys)]
appendo xs ys o = [(== xs a:d)
                   (== o a:o^)
                   (appendo d ys o^)]
```

Logic programming languages can be better “circuit discription languages”!

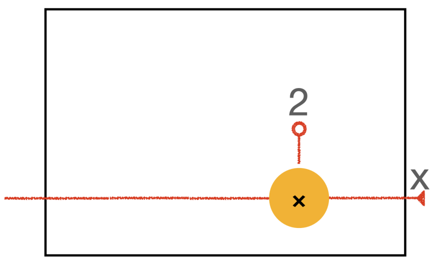
Circuit Simplification



$\overset{\lambda}{\longrightarrow}$ $\underset{\varsigma}{\longleftarrow}$

$$\lambda x.\,[(\lambda x.x)((\lambda x.2 \times x)\, x)]$$

$\downarrow \varphi$ $\qquad \sim \qquad$ $\downarrow \sigma$



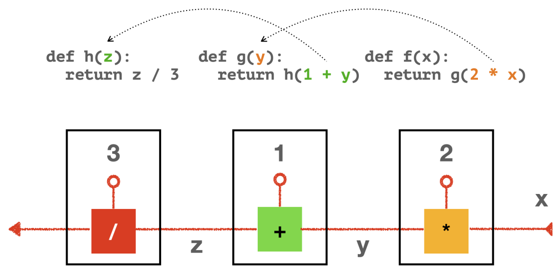
$\overset{\lambda}{\longrightarrow}$ $\underset{\varsigma}{\longleftarrow}$

$$\lambda x.2 \times x$$

λ, ς - converting between circuits and lambdas

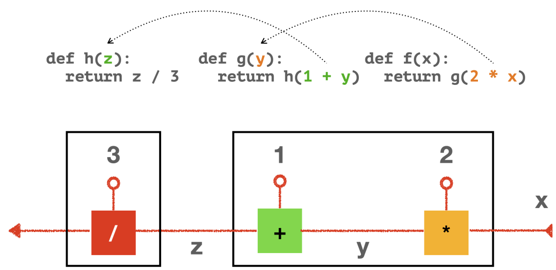
φ - circuit simplification

σ - supercompilation / partial evaluation



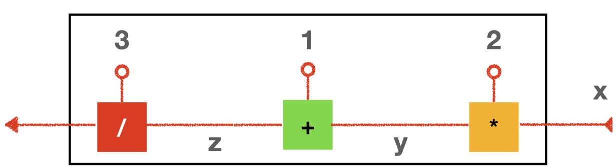
⇒

```
def f(x):
    y = 2 * x
    return g(y)
```



⇒

```
def f(x):
    y = 2 * x
    z = 1 + y
    return h(z)
```



⇒

```
def f(x):
    y = 2 * x
    z = 1 + y
    return z / 3
```

Questions : Are they still “functional” programs? In what sense?

Few observations here:

- variables and function arguments are the same thing in essence
- they are both "points"/ "anchors" of data flow (can always be exposed)
- in λ-circuits they are both wires / conductors
- functional programs $\neq$ point-free programs

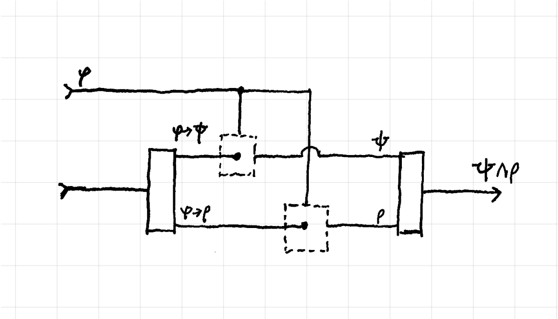
For circuit simplification $\sim$ partial evaluation:

- remove boxes (scope delimiters) $\Rightarrow$ β-reduction
- adding boxes $\Rightarrow$ η-expansion, Kleene's S^n_m theorem
- swallowing boxes $\Rightarrow$ inlining
- splitting boxes
- annihilation of constructors and eliminators / MUX and DEMUX
- collapsing of DEMUX
- ...

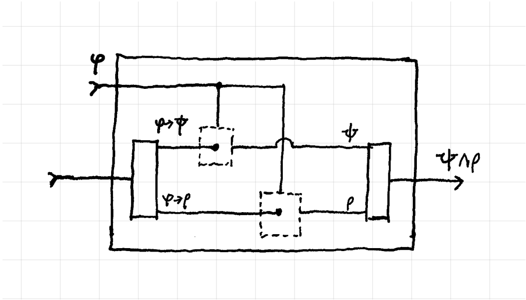
Type Inference

see old slides…

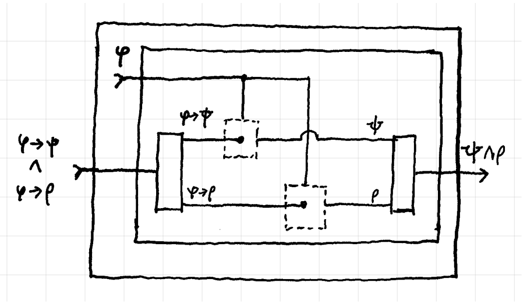
$$\dfrac{\dfrac{\varphi \qquad \dfrac{(\varphi \to \psi) \wedge (\varphi \to \rho)}{\varphi \to \psi}}{\psi} \qquad \dfrac{\varphi \qquad \dfrac{(\varphi \to \psi) \wedge (\varphi \to \rho)}{\varphi \to \rho}}{\rho}}{\psi \wedge \rho}$$



$$\dfrac{\dfrac{\dfrac{[\varphi] \qquad \dfrac{(\varphi \to \psi) \wedge (\varphi \to \rho)}{\varphi \to \psi}}{\psi} \qquad \dfrac{[\varphi] \qquad \dfrac{(\varphi \to \psi) \wedge (\varphi \to \rho)}{\varphi \to \rho}}{\rho}}{\psi \wedge \rho}}{\varphi \to \psi \wedge \rho}$$



$$\dfrac{\dfrac{\dfrac{\dfrac{[\varphi] \qquad \dfrac{[(\varphi \to \psi) \wedge (\varphi \to \rho)]}{\varphi \to \psi}}{\psi} \qquad \dfrac{[\varphi] \qquad \dfrac{[(\varphi \to \psi) \wedge (\varphi \to \rho)]}{\varphi \to \rho}}{\rho}}{\psi \wedge \rho}}{\varphi \to \psi \wedge \rho}}{(\varphi \to \psi) \wedge (\varphi \to \rho) \to \varphi \to \psi \wedge \rho}$$



TODO:

- to explore type classes like Applicative, Traversable, and Monad through λ-circuits
- focus on Monad and Arrow
- for Monad, focus on continuation monad and its implications

Discussion

- can we imagine wires / conductors with some kind of resistance?
- how to extend λ-circuits to first-order types and more (see lambda cube)?