Programming Languages as Notations

Chelsea Voss

@csvoss

Software Engineer at Wave
chelsea@wave.com

April 20, 2017

About me: oneliner-izer talk

About me: oneliner-izer talk

- **Claim**: it's possible to write any Python program as one line of code

About me: oneliner-izer talk

- **Claim**: it's possible to write any Python program as one line of code
- **Proof**: by lambda calculus

About me: oneliner-izer talk

- **Claim**: it's possible to write any Python program as one line of code
- **Proof**: by lambda calculus

Today: notations $\leftrightarrow$ programming languages

Visual notations

Visual notations

Lots of notations used in CS, math, and science are highly visual.

Visual notations in computer science

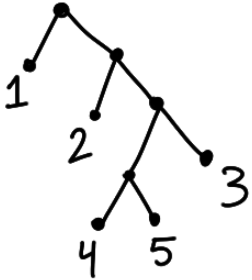
Binary tree.

Visual notations in computer science

Binary tree.

```
Tree(1,
     Tree(2,
          Tree(Tree(4, 5),
               3)))
```

→



Visual notations in computer science

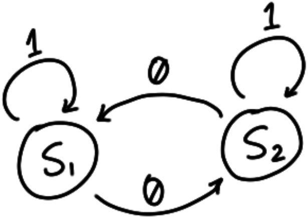
State machine.

Visual notations in computer science

State machine.

```
[
 Transition(S1, 0, S2),
 Transition(S1, 1, S1),
 ...
]
```

→



Visual notations in computer science

Boolean circuits.

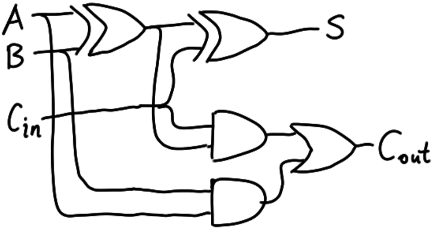
Visual notations in computer science

Boolean circuits.

C_{out} = `Or(And(`A`, `B`),`
`And(`C_{in}`, Xor(`A`, `B`)))`

S = `Xor(Xor(`A`, `B`), `C_{in}`)`

$\rightarrow$



Visual notations beyond computer science

Protein signalling pathways.

Visual notations beyond computer science

Protein signalling pathways.

```
A inhibits B
A inhibits C
B activates C
C activates D
D activates B
```

Visual notations beyond computer science

Protein signalling pathways.

```
A inhibits B
A inhibits C
B activates C
C activates D
D activates B
```

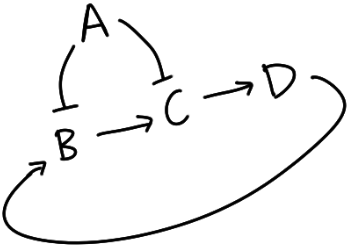
$\rightarrow$

Visual notations beyond computer science

Protein signalling pathways.

```
A inhibits B
A inhibits C
B activates C
C activates D
D activates B
```

$\rightarrow$



Visual notations beyond computer science

Feynman diagrams.

Visual notations beyond computer science

Feynman diagrams.

- Quantum electrodynamics

Visual notations beyond computer science

Feynman diagrams.

- Quantum electrodynamics
- Sample problem: electron scattering

Visual notations beyond computer science

Feynman diagrams.

- Quantum electrodynamics
- Sample problem: electron scattering

"The formalism was notoriously cumbersome, an algebraic nightmare of distinct terms to track and evaluate. . . Individual contributions to the overall calculation stretched over four or five lines of algebra."

– David Kaiser, in "Physics and Feynman's Diagrams," *American Scientist*, volume 93.

Visual notations beyond computer science

Feynman diagrams.

$$e^2 \int \int K(3,5)K(4,6)\gamma_\mu \delta(s_{56}^2)$$
$$\gamma_\mu K(5,1)K(6,2)d^4x_5 d^4x_6$$

Visual notations beyond computer science

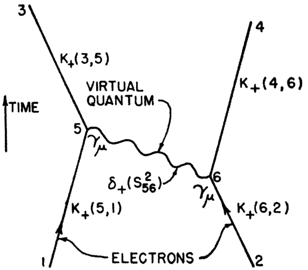
Feynman diagrams.

$$e^2 \int \int K(3,5)K(4,6)\gamma_\mu \delta(s_{56}^2) \quad \rightarrow$$
$$\gamma_\mu K(5,1)K(6,2)d^4 x_5 d^4 x_6$$

Visual notations beyond computer science

Feynman diagrams.

$$e^2 \int \int K(3,5)K(4,6)\gamma_\mu \delta(s_{56}^2) \gamma_\mu K(5,1)K(6,2)d^4x_5 d^4x_6 \quad \rightarrow$$



Richard Feynman, *Space-Time Approach to Quantum Electrodynamics*, 1949.
David Kaiser, "Physics and Feynman's Diagrams", *American Scientist* volume 93, 2005.

Visual notations for computer science?

Visual notations for computer science?

```
dbm = 0
for dim in _DAYS_IN_MONTH[1:]:
    _DAYS_BEFORE_MONTH.append(dbm)
    dbm += dim
del dbm, dim

def _is_leap(year):
    "year -> 1 if leap year, else 0."
    return year % 4 == 0 and (year % 100 != 0 or
                              year % 400 == 0)

def _days_before_year(year):
    "year -> number of days before January 1st of year."
    y = year - 1
    return y*365 + y//4 - y//100 + y//400

def _days_in_month(year, month):
    "year, month -> number of days in that month in that year."
    assert 1 <= month <= 12, month
    if month == 2 and _is_leap(year):
        return 29
    return _DAYS_IN_MONTH[month]
```

Visual notations for computer science?

```
dbm = 0
for dim in _DAYS_IN_MONTH[1:]:
    _DAYS_BEFORE_MONTH.append(dbm)
    dbm += dim
del dbm, dim

def _is_leap(year):
    "year -> 1 if leap year, else 0."
    return year % 4 == 0 and (year % 100 != 0 or
                              year % 400 == 0)

def _days_before_year(year):
    "year -> number of days before January 1st of year."
    y = year - 1
    return y*365 + y//4 - y//100 + y//400

def _days_in_month(year, month):
    "year, month -> number of days in that month in that year."
    assert 1 <= month <= 12, month
    if month == 2 and _is_leap(year):
        return 29
    return _DAYS_IN_MONTH[month]
```

→

Visual notations for computer science?

```
dbm = 0
for dim in _DAYS_IN_MONTH[1:]:
    _DAYS_BEFORE_MONTH.append(dbm)
    dbm += dim
del dbm, dim

def _is_leap(year):
    "year -> 1 if leap year, else 0."
    return year % 4 == 0 and (year % 100 != 0 or
                              year % 400 == 0)

def _days_before_year(year):
    "year -> number of days before January 1st of year."
    y = year - 1
    return y*365 + y//4 - y//100 + y//400

def _days_in_month(year, month):
    "year, month -> number of days in that month in that year."
    assert 1 <= month <= 12, month
    if month == 2 and _is_leap(year):
        return 29
    return _DAYS_IN_MONTH[month]
```

→ **What visual equivalent?**

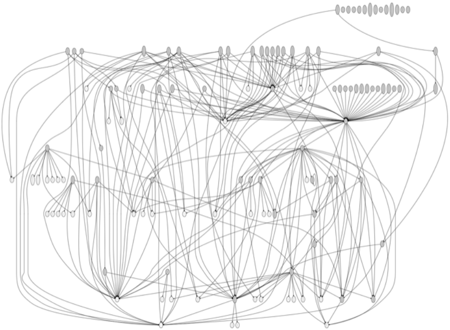
Visual notations for computer science?

Visual notations for computer science?

`snakefood` visualizes dependencies in Python codebases:

Visual notations for computer science?

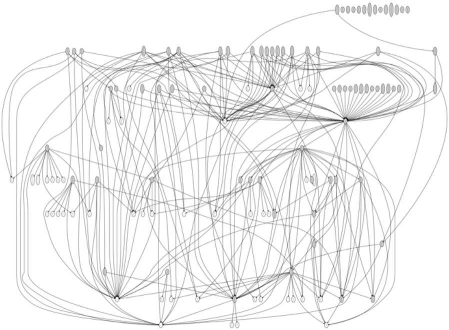
`snakefood` visualizes dependencies in Python codebases:



Flask - http://grokcode.com/864/snakefooding-python-code-for-complexity-visualization/

Visual notations for computer science?

`snakefood` visualizes dependencies in Python codebases:



Flask - http://grokcode.com/864/snakefooding-python-code-for-complexity-visualization/

...but can we have a notation for the *entire language*?

Visual notations

Lots of notations used in CS, math, and science are highly visual.

Visual notations

Lots of notations used in CS, math, and science are highly visual.

But programming languages themselves aren't.

Visual notations

Lots of notations used in CS, math, and science are highly visual.

But programming languages themselves aren't.

- **Claim**: It's possible to create a visual notation for an entire programming language.

Visual notations

Lots of notations used in CS, math, and science are highly visual.

But programming languages themselves aren't.

- **Claim**: It's possible to create a visual notation for an entire programming language.
- **Proof**:

Visual notations

Lots of notations used in CS, math, and science are highly visual.

But programming languages themselves aren't.

- **Claim**: It's possible to create a visual notation for an entire programming language.
- **Proof**: by lambda calculus.

Example: circuitry for lambda calculus

Example: circuitry for lambda calculus

Lambda calculus is:

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) =$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$
- $((\lambda x.\ \lambda y.\ x + y)\ 2\ 3) =$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$
- $((\lambda x.\ \lambda y.\ x + y)\ 2\ 3) = 5$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$
- $((\lambda x.\ \lambda y.\ x + y)\ 2\ 3) = 5$
- $((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) =$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$
- $((\lambda x.\ \lambda y.\ x + y)\ 2\ 3) = 5$
- $((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) = ((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) =$

Example: circuitry for lambda calculus

Lambda calculus is:

- A formalization of computation, devised by Alonzo Church around 1935
- Consists of expressions made only of *functions* and their arguments
- For example, $(\lambda x.\ x)$ is the identity function.
- $((\lambda x.\ x)\ 2) = 2$
- $((\lambda x.\ \lambda y.\ x + y)\ 2\ 3) = 5$
- $((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) = ((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) =$ ...loops forever.

Example: circuitry for lambda calculus

Basic design: inputs flow to outputs, passing through functions

Example: circuitry for lambda calculus

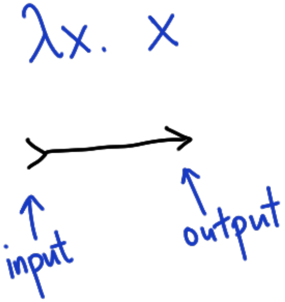
Basic design: inputs flow to outputs, passing through functions

The identity function:

Example: circuitry for lambda calculus

Basic design: inputs flow to outputs, passing through functions

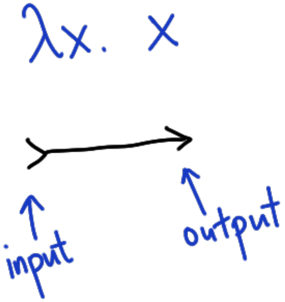
The identity function:



Example: circuitry for lambda calculus

Basic design: inputs flow to outputs, passing through functions

The identity function:

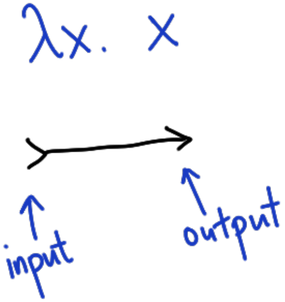


Multiple arguments, and application:

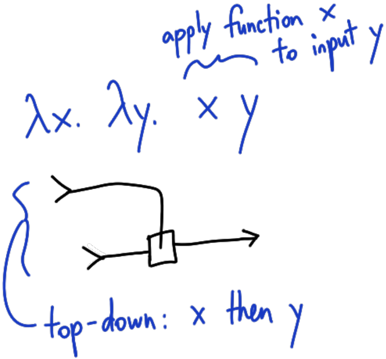
Example: circuitry for lambda calculus

Basic design: inputs flow to outputs, passing through functions

The identity function:



Multiple arguments, and application:

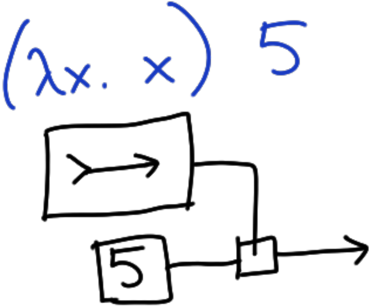


Example: circuitry for lambda calculus

Example execution:

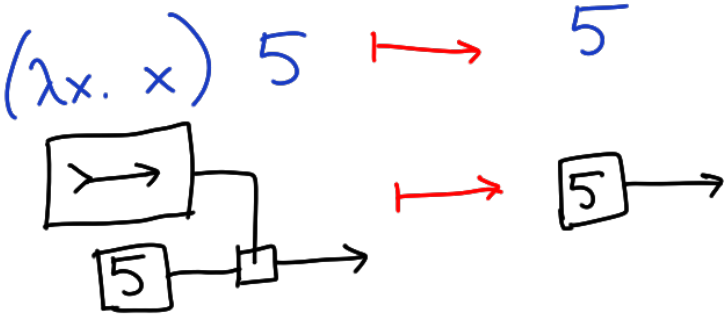
Example: circuitry for lambda calculus

Example execution:



Example: circuitry for lambda calculus

Example execution:



Example: circuitry for lambda calculus

Building Boolean logic:

Example: circuitry for lambda calculus

Building Boolean logic:

"true" $\lambda x.\ \lambda y.\ x$

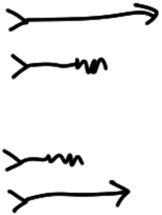
"false" $\lambda x.\ \lambda y.\ y$

Example: circuitry for lambda calculus

Building Boolean logic:

"true" $\lambda x.\ \lambda y.\ x$

"false" $\lambda x.\ \lambda y.\ y$

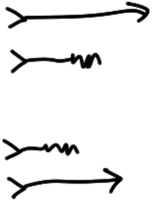


Example: circuitry for lambda calculus

Building Boolean logic:

"true" $\lambda x.\ \lambda y.\ x$

"false" $\lambda x.\ \lambda y.\ y$

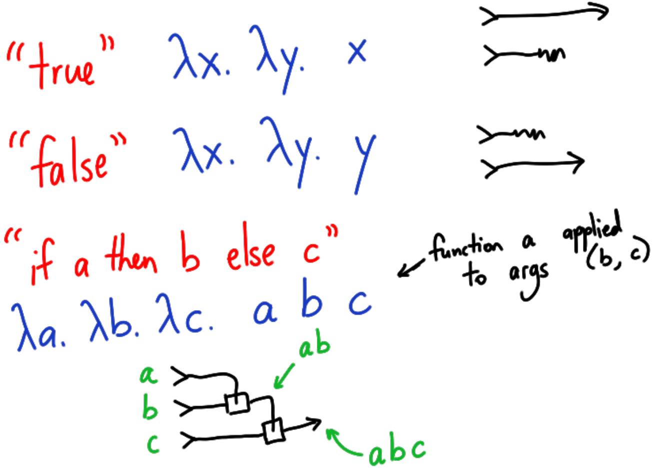


"if a then b else c"

$\lambda a.\ \lambda b.\ \lambda c.\ a\ b\ c$ ← function a applied to args (b, c)

Example: circuitry for lambda calculus

Building Boolean logic:



Example: circuitry for lambda calculus

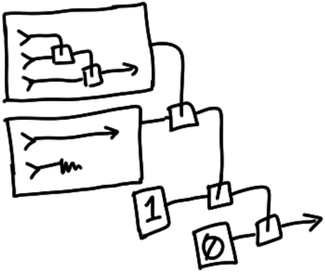
"if true then 1 else 0"

$(\lambda a. \lambda b. \lambda c.\ a\,b\,c)\ (\lambda x. \lambda y.\ x)\ 1\ 0$

Example: circuitry for lambda calculus

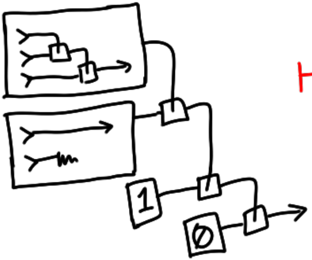
"if true then 1 else 0"

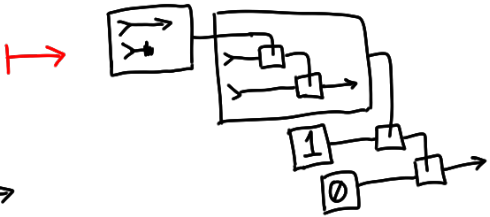
$(\lambda a.\ \lambda b.\ \lambda c.\ a\ b\ c)\ (\lambda x.\ \lambda y.\ x)\ 1\ 0$



Example: circuitry for lambda calculus

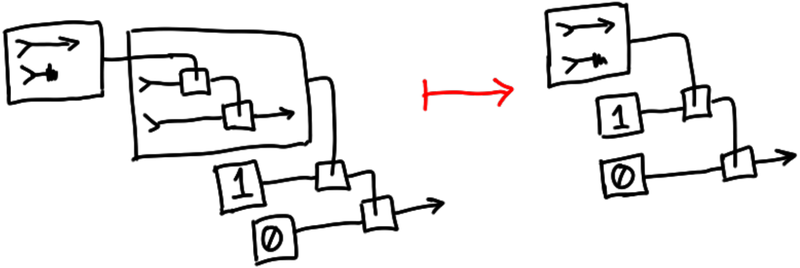
"if true then 1 else 0"





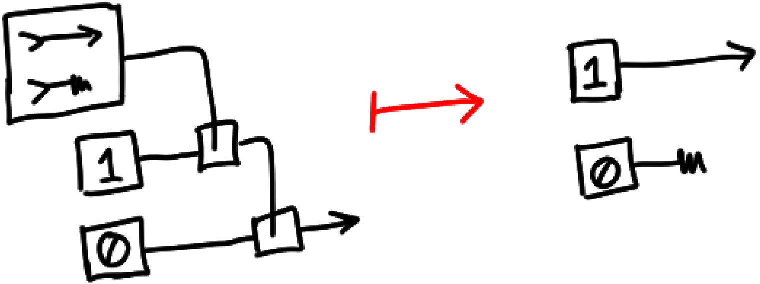
Example: circuitry for lambda calculus

"if true then 1 else 0"



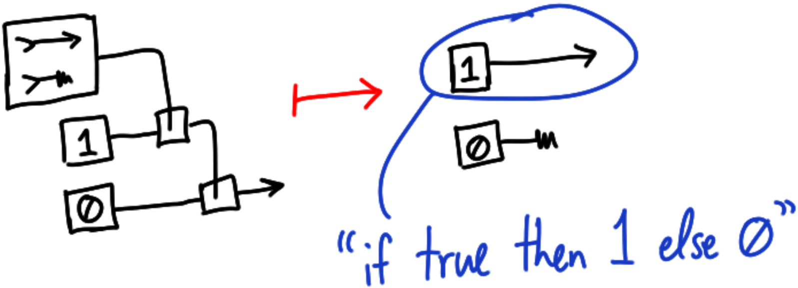
Example: circuitry for lambda calculus

"if true then 1 else 0"



Example: circuitry for lambda calculus

"if true then 1 else 0"



Example: circuitry for lambda calculus

Linked lists

first =

second =

pair =

Example: circuitry for lambda calculus

Linked lists

first = λp. p true

second = λp. p false

pair = λa. λb. λx. if x then a else b

Example: circuitry for lambda calculus

Linked lists

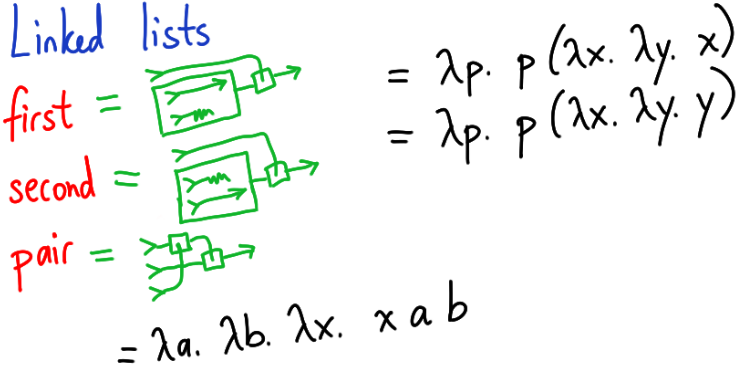
$$\text{first} = \lambda p.\ p\ \text{true} = \lambda p.\ p\ (\lambda x.\ \lambda y.\ x)$$

$$\text{second} = \lambda p.\ p\ \text{false} = \lambda p.\ p\ (\lambda x.\ \lambda y.\ y)$$

$$\text{pair} = \lambda a.\ \lambda b.\ \lambda x.\ \text{if } x \text{ then } a \text{ else } b$$

$$= \lambda a.\ \lambda b.\ \lambda x.\ x\ a\ b$$

Example: circuitry for lambda calculus



Example: circuitry for lambda calculus

Omega combinator – applies input to itself:

Example: circuitry for lambda calculus

Omega combinator – applies input to itself:

$$\omega = \lambda x.\ (x\ x)$$

Example: circuitry for lambda calculus

Omega combinator – applies input to itself:

$$\omega = \lambda x.\ (x\ x)$$

$$\omega\omega = (\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x)) = \ldots$$

Example: circuitry for lambda calculus

Omega combinator – applies input to itself:

$$\omega = \lambda x.\ (x\ x)$$

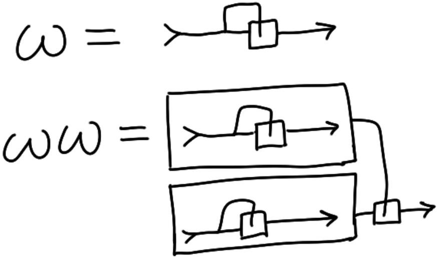
$$\omega\omega = (\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x)) = \ldots$$

Example: circuitry for lambda calculus

Omega combinator – applies input to itself:

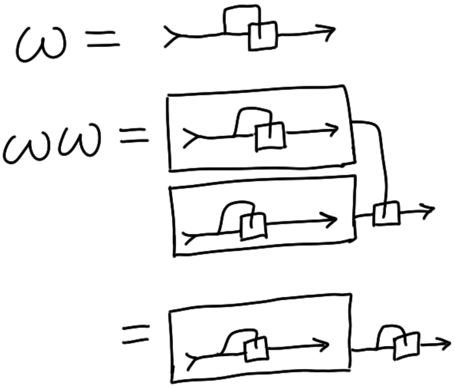
$$\omega = \lambda x.\,(x\;x)$$

$$\omega\omega = (\lambda x.\,(x\;x))\;(\lambda x.\,(x\;x)) = \ldots$$



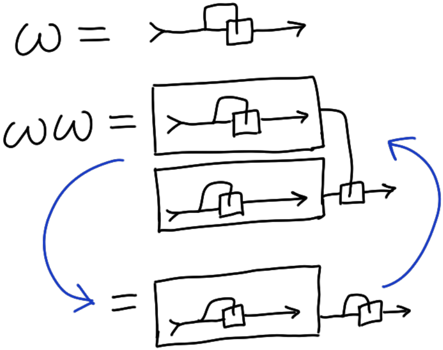
Example: circuitry for lambda calculus

Omega combinator – applies input to itself:



Example: circuitry for lambda calculus

Omega combinator – applies input to itself:



Example: circuitry for lambda calculus

Fixed point combinator:

Example: circuitry for lambda calculus

Fixed point combinator:

$$Y\ f = f\ Y\ f$$

Example: circuitry for lambda calculus

Fixed point combinator:

$$Y\ f = f\ Y\ f$$

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

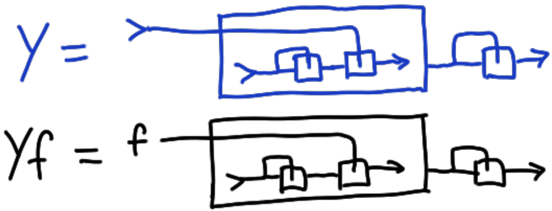
Example: circuitry for lambda calculus

Fixed point combinator:

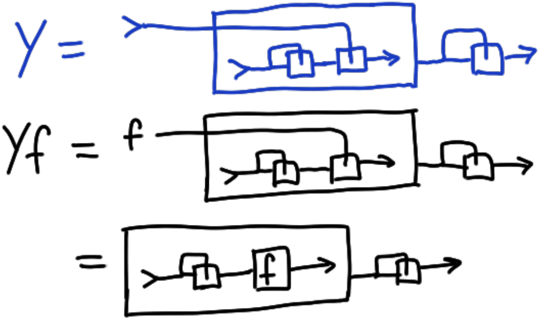
$$Y\ f = f\ Y\ f$$

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

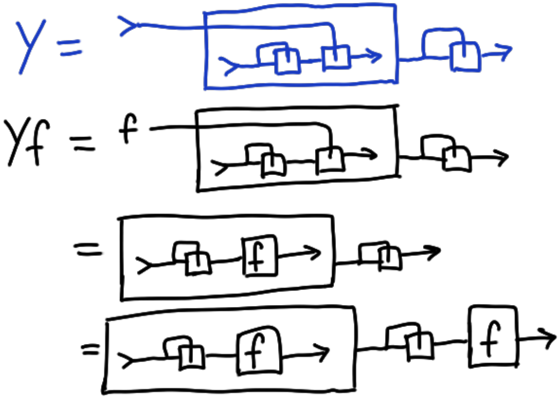
Example: circuitry for lambda calculus



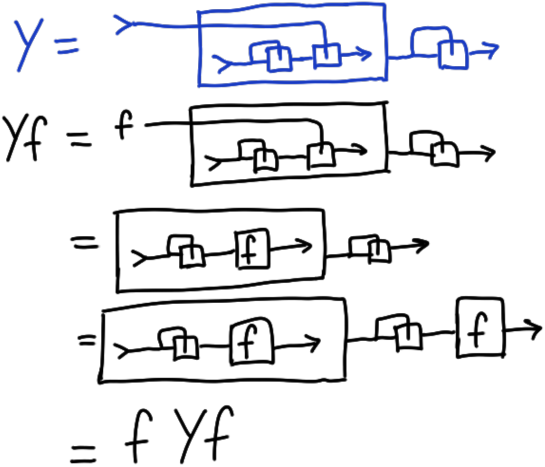
Example: circuitry for lambda calculus



Example: circuitry for lambda calculus



Example: circuitry for lambda calculus



Suggested further exercises

Suggested further exercises

- Implement numbers: addition, subtraction, multiplication, exponentiation

Suggested further exercises

- Implement numbers: addition, subtraction, multiplication, exponentiation
- map, reduce, filter for the list implementation

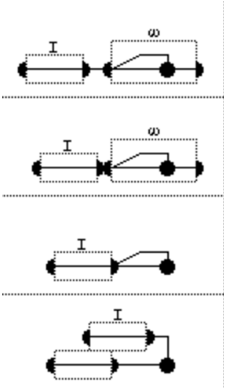
Suggested further exercises

- Implement numbers: addition, subtraction, multiplication, exponentiation
- map, reduce, filter for the list implementation
- combinator puzzles in *To Mock a Mockingbird*

Similar previous work

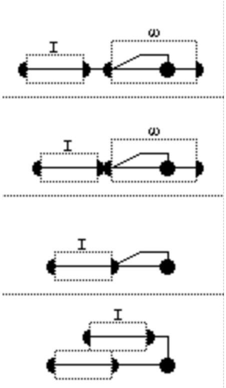
Similar previous work

To Dissect a Mockingbird

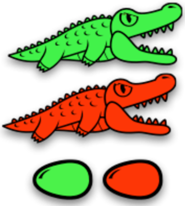


Similar previous work

To Dissect a Mockingbird

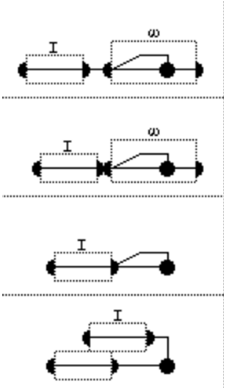


Alligator Eggs game

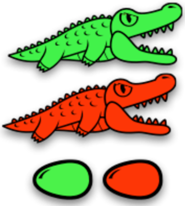


Similar previous work

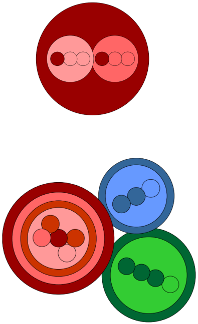
To Dissect a Mockingbird



Alligator Eggs game

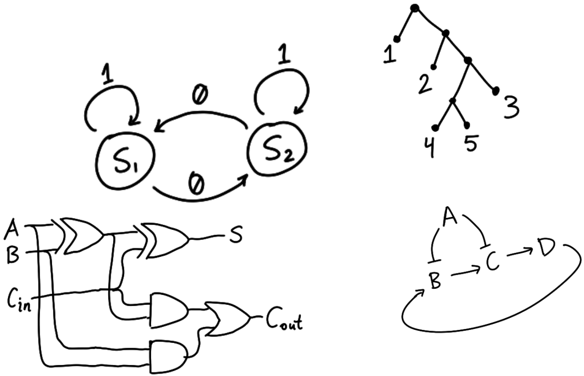


Visual Lambda Calculus, bubble notation and GUI

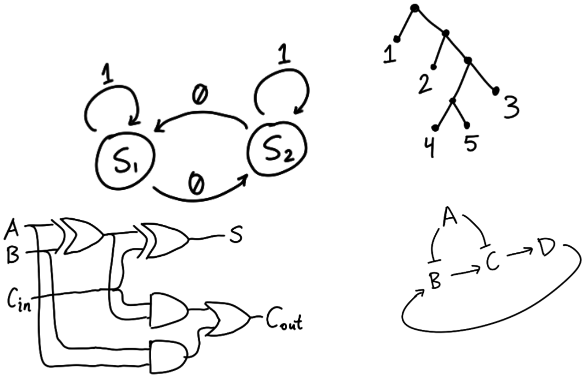


Notations as abstractions

Notations as abstractions

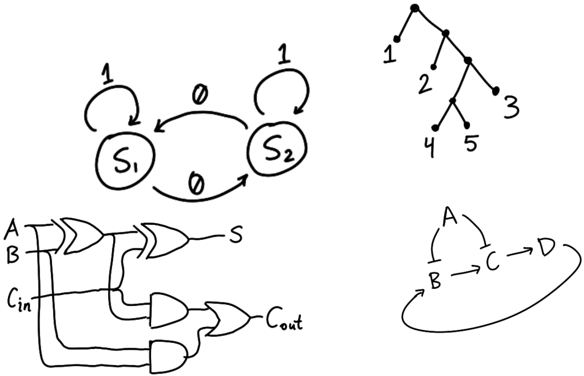


Notations as abstractions



Notations can only exist on top of **abstractions**.

Notations as abstractions



Notations can only exist on top of **abstractions**.

Abstractions trade **freedom** for **specificity**.

Keep making abstractions!

Keep making abstractions!

- Abstractions that limit allowable code to be more correct

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks
  - programming languages for other abstraction levels?

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks
  - programming languages for other abstraction levels?
- New programming models

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks
  - programming languages for other abstraction levels?
- New programming models
  - Pict: concurrent programming

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks
  - programming languages for other abstraction levels?
- New programming models
  - Pict: concurrent programming
    - π-calculus

Keep making abstractions!

- Abstractions that limit allowable code to be more correct
  - Static type checking
  - Dependent types for code correctness
    - Dafny: `ensures`, `requires`
- *Executable biology* – programs that simulate biological processes
  - Kappa: rule-based protein interaction networks
  - programming languages for other abstraction levels?
- New programming models
  - Pict: concurrent programming
    - π-calculus
    - both **sequential composition** and **parallel composition** of code

One final similarity between notations and programming languages...

One final similarity between notations and programming languages...

sometimes you get into wars about which ones are right!

Notation wars

Vectors vs. quaternions.

Notation wars

Vectors vs. quaternions.

Oliver Heaviside, in *Electromagnetic Theory*, 1893:

Notation wars

Vectors vs. quaternions.

Oliver Heaviside, in *Electromagnetic Theory*, 1893:

"A vector is considered by Hamilton and Tait to be a quaternion… It is *really* a vector. It is as unfair to call a vector a quaternion as to call a man a quadruped."

Notation wars

Vectors vs. quaternions.

Oliver Heaviside, in *Electromagnetic Theory*, 1893:

"A vector is considered by Hamilton and Tait to be a quaternion... It is *really* a vector. It is as unfair to call a vector a quaternion as to call a man a quadruped."

"Students who had found quaternions quite hopeless could understand my vectors very well."

Notation wars

Standards proliferated.

Notation wars

Standards proliferated.

Florian Cajori, in *A History of Mathematical Notations*, 1928:

Notation wars

Standards proliferated.

Florian Cajori, in *A History of Mathematical Notations*, 1928:

"...the mark '$V\nabla a$,' used by Tait, is Gibb's '$\nabla \times a$,' Heaviside's 'curl a,' Wiechert's 'Quirl a,' Lorentz' 'Rot a,' Voigt's 'Vort a,' Abraham and Langevin's 'Rot $\tilde{a}$.' "

Notation wars

Standards proliferated.

Florian Cajori, in *A History of Mathematical Notations*, 1928:

"...the mark '$V\nabla a$,' used by Tait, is Gibb's '$\nabla \times a$,' Heaviside's 'curl a,' Wiechert's 'Quirl a,' Lorentz' 'Rot a,' Voigt's 'Vort a,' Abraham and Langevin's 'Rot $\tilde{a}$.' "

- committee appointed by Felix Klein in 1903: couldn't decide
- special commission of the International Congress of Mathematicians in 1908: couldn't decide

Notation wars

Standards proliferated.

Florian Cajori, in *A History of Mathematical Notations*, 1928:

"...the mark '$V\nabla a$,' used by Tait, is Gibb's '$\nabla \times a$,' Heaviside's 'curl a,' Wiechert's 'Quirl a,' Lorentz' 'Rot a,' Voigt's 'Vort a,' Abraham and Langevin's 'Rot $\tilde{a}$.' "

- committee appointed by Felix Klein in 1903: couldn't decide
- special commission of the International Congress of Mathematicians in 1908: couldn't decide

Maybe having many standards is the necessary price of **innovation**.

What programming languages can learn from notations:

What programming languages can learn from notations:

- Be visual

What programming languages can learn from notations:

- Be visual
- Build abstractions

What programming languages can learn from notations:

- Be visual
- Build abstractions
- Standardization wars happen sometimes

What programming languages can learn from notations:

- Be visual
- Build abstractions
- Standardization wars happen sometimes

Further reading:

- *Drawing Theories Apart*, David Kaiser: the history of Feynman diagrams
- *To Mock a Mockingbird*, Raymond Smullyan: combinator puzzles
- *A History of Mathematical Notations*, Florian Cajori

What programming languages can learn from notations:

- Be visual
- Build abstractions
- Standardization wars happen sometimes

Further reading:

- *Drawing Theories Apart*, David Kaiser: the history of Feynman diagrams
- *To Mock a Mockingbird*, Raymond Smullyan: combinator puzzles
- *A History of Mathematical Notations*, Florian Cajori

Thanks!

`@csvoss`