Combinators II

CS242

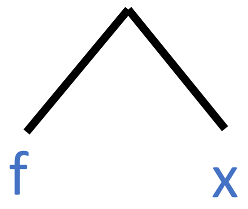
Lecture 3

Review

- Function application written as space/juxtaposition

f x

- Programs as trees



SKI Calculus

$I\ x \rightarrow x$	Identity function
$K\ x\ y \rightarrow x$	Constant functions
$S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$	Generalized function application

Writing Combinators: A Systematic Approach

- Finding a combinator that implements a given function is not trivial
  - Some have nice intuitive definitions (e.g., Booleans)
  - Others are completely non-obvious (e.g., swap)

- There is a systematic way to write combinators
  - Start with a function equation using variables that specifies what we want

    swap x y = y x
  - An *abstraction algorithm* A(…) maps the right-hand side to a combinator
  - The key is to eliminate the variables by replacing them with uses of the combinators S, K, and I

Writing Combinators: A Systematic Approach

- Consider a function equation of one variable: f x = E
  - The equation can use combinators and variables
  - If we apply function f to argument x, the result is E

- We want a combinator A(E,x) that implements f
  - Therefore A(E,x) x = E
  - And A(E,x) doesn't use x
  - We say we *abstract* E with respect to x

- A(x,x) = I
- A(E,x) = K E   if x does not appear in E
- A(E1 E2,x) = S A(E1,x) A(E2,x)

- Note A(…) is not a combinator
  - it is a (recursively defined) mapping from expressions with variables to combinators

Working Through Each Case …

- $A(x,x) = I$

- Consider the equation $f\ x = x$
  - Requires $A(x,x)\ x = x$
  - And $A(x,x)$ does not use x

- What combinator satisfies these two conditions? I!

Working Through Each Case …

- A(E,x) = K E

- Consider the equation f x = E
  - Where E does not use x
  - Again requires A(E,x) x = E
  - And A(E,x) does not use x

- Note that K E does not use x
- Calculate: K E x → E

Working Through Each Case …

- A(E1 E2,x) = S A(E1,x) A(E2,x)
- Consider the equation f x = (E1 E2) x
  - Requires A(E1 E2,x) x = E1 E2
  - And A(E1 E2,x) does not use x

- Notice that S A(E1,x) A(E2,x) does not use x
- Calculate:

  S A(E1,x) A(E2,x) x → (A(E1,x) x) (A(E2,x) x) → E1 (A(E2,x) x) → E1 E2

Improvements

- We can introduce helper combinators to reduce the size of abstracted expressions

- In S x y z, often z is only used in one of x or y
  - We can avoid copying z and just pass it to the one combinator that uses it

- Define
  - c1 x y z = x (y z)   – a version of S where the first argument is constant (doesn't use z)
  - c2 x y z = (x z) y   – a version of S where the second argument is constant (doesn't use z)

- Add new cases for to the abstraction algorithm for applications that use c1 or c2 if possible

  A(E1 E2,x) = c1 E1 A(E2,x) if x does not appear in E1

  A(E1 E2,x) = c2 A(E1,x) E2 if x does not appear in E2

  A(E1 E2,x) = S A(E1,x) A(E2,x) otherwise

Natural Numbers and Factorial

Natural Numbers

n applies its first argument n times to its second argument

$n\ f\ x = f^n(x)$

$0\ f\ x = x$ so $0 = S\ K$

$\text{succ}\ n\ f\ x = f\ (n\ f\ x)$ $\text{succ} = S\ (S\ (K\ S)\ K)$

$\text{succ}\ n\ f\ x \rightarrow S\ (S\ (K\ S)\ K)\ \ n\ f\ x \rightarrow (S\ (K\ S)\ K\ f)\ (n\ f)\ x \rightarrow ((K\ S)\ f)\ (K\ f)\ (n\ f)\ x \rightarrow$
$S\ (K\ f)\ (n\ f)\ x \rightarrow ((K\ f)\ x)\ ((n\ f)\ x) \rightarrow f\ ((n\ f)\ x) = f\ (n\ f\ x)$

Some Useful Functions

one = succ 0

add x y = x succ y

mul x y = x (add y) 0

Abstracting add and mul:

add = c2 (c1 c1 (c2 I succ)) I

mul = c2 (c1 c2 (c2 (c1 c1 I) (c1 add I))) 0

Examples

Shorthand: Write i for $succ^i(0)$

10 (+ 2) 0 → 20

2 (* 2) 1 → 4

Notice how iteration/looping is built-in to the definition of the type.

An example of *primitive recursion*: The number of times we iterate is fixed by the element of the type itself.

Factorial

Standard recursive implementation:

fac n = fac’ 1 1 n

fac’ a i n = if i > n then a else fac’ (a*i) i+1 n

Replace arguments a and i by a pair;

use p.1 and p.2 to select first and second components respectively

fac n = fac’ (pair 1 1) n

fac’ p n = if p.2 > n then p.1 else fac’ (pair (p.2 * p.1) (p.2 + 1)) n

Now define functions (switching from infix to prefix operations):

m p = * (p second) (p first) = mul (p second) (p first)

i2 p = + 1 (p second) = succ (p second)

Abstract the functions into combinators:

m = S (c1 mul (c2 I first)) (c2 I second);

i2 = c1 succ (c2 I second)

Using the combinators:

fac n = (fac’ (pair 1 1) n) first

fac’ p n = if p.2 > n then p.1 else fac’ (pair (m p) (i2 p))

Now use the recursion built into the natural numbers:

fac n = (n fac’ (pair one one)) first

fac’ p = pair (m p) (i2 p)

Abstracting into combinators:

fac = (c2 (c2 I fac’) (pair one one)) first

fac’ = S (c1 pair m) i2

From The Ground Up!

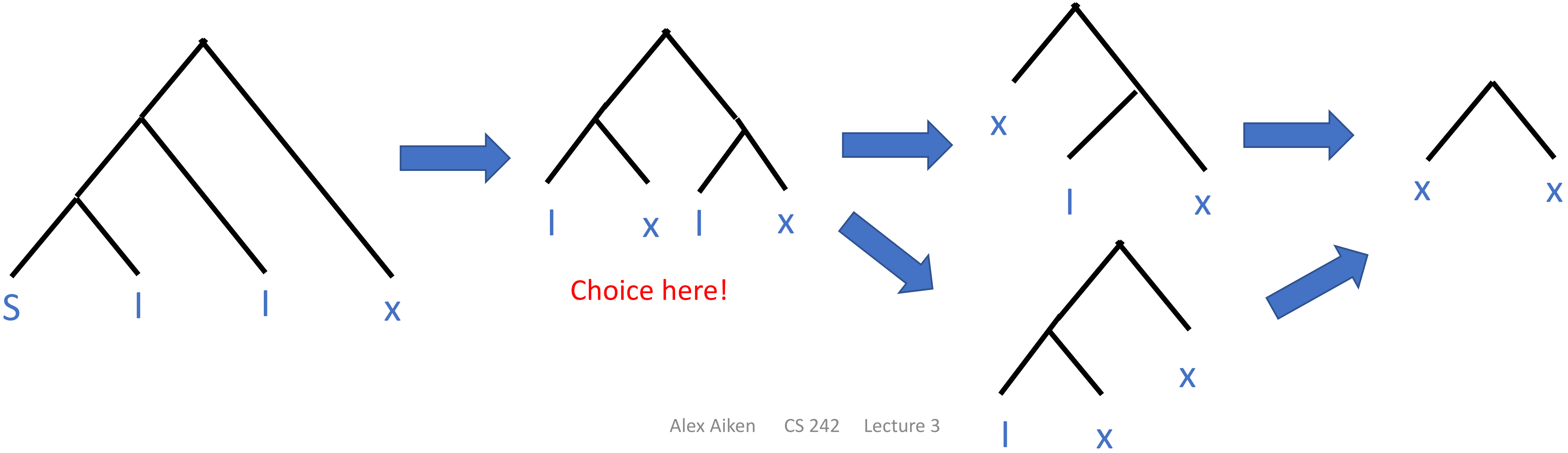
- 14 combinator definitions
- Including
  - Abstraction helpers
  - Control structures
  - Pairs
  - Natural numbers
  - Addition
  - Multiplication

```
# abstraction operators
c1 = S (S (K K) (S (K S) (S (K K) I))) (K (S (S (K S) (S (K K) I)) (K I)))
c2 = S ((c1 S (c1 K (c1 S (S (c1 c1 I) (K I)))))) (K (c1 K I))
# pairs
first = K
second = S K
pair = c2 (c1 c1 (c1 c2 (c1 (c2 I) I))) I
# natural numbers
0 = S K
succ = S (S (K S) K)
one = succ 0
add = c2 (c1 c1 (c2 I succ)) I;
mul = c2 (c1 c2 (c2 (c1 c1 I) (c1 add I))) 0;
# factorial and auxiliary functions
m = S (c1 mul (c2 I first)) (c2 I second);
i2 = c1 succ (c2 I second)
fac' = S (c1 pair m) i2
fac = (c2 (c2 I fac') (pair one one)) first
```

Reduction Order & Confluence

Consider …

S I I x → (I x) (I x) → x (I x) → x x



Order of Evaluation

- In a large expression, many rewrite rules may apply

- Which one should we choose?

Order of Evaluation

- A process for choosing where to apply the rules is a *reduction strategy*
  - Each rule application is one reduction

- Most languages have a fixed reduction/evaluation order
  - So people forget that there might be more than one choice
  - But concurrent/parallel languages do provide multiple choices

Order of Evaluation

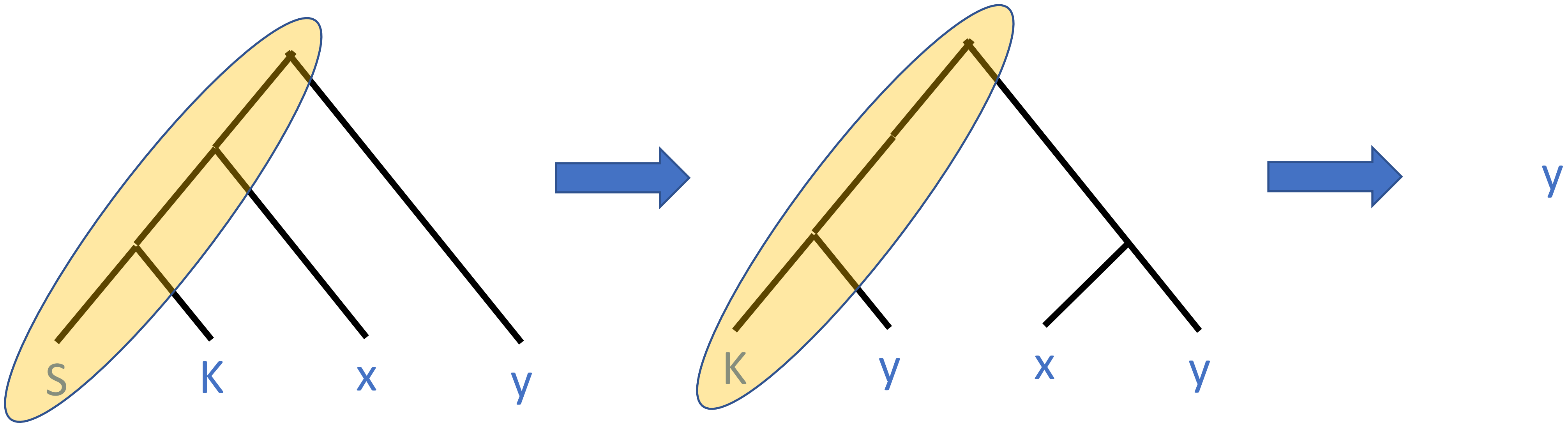
What is a good reduction strategy?

A Standard Choice

- Normal order
  - Traverse the leftmost spine of the expression tree from the root to the leaf combinator
  - If a rewrite rule applies, apply it, and repeat
  - Otherwise halt

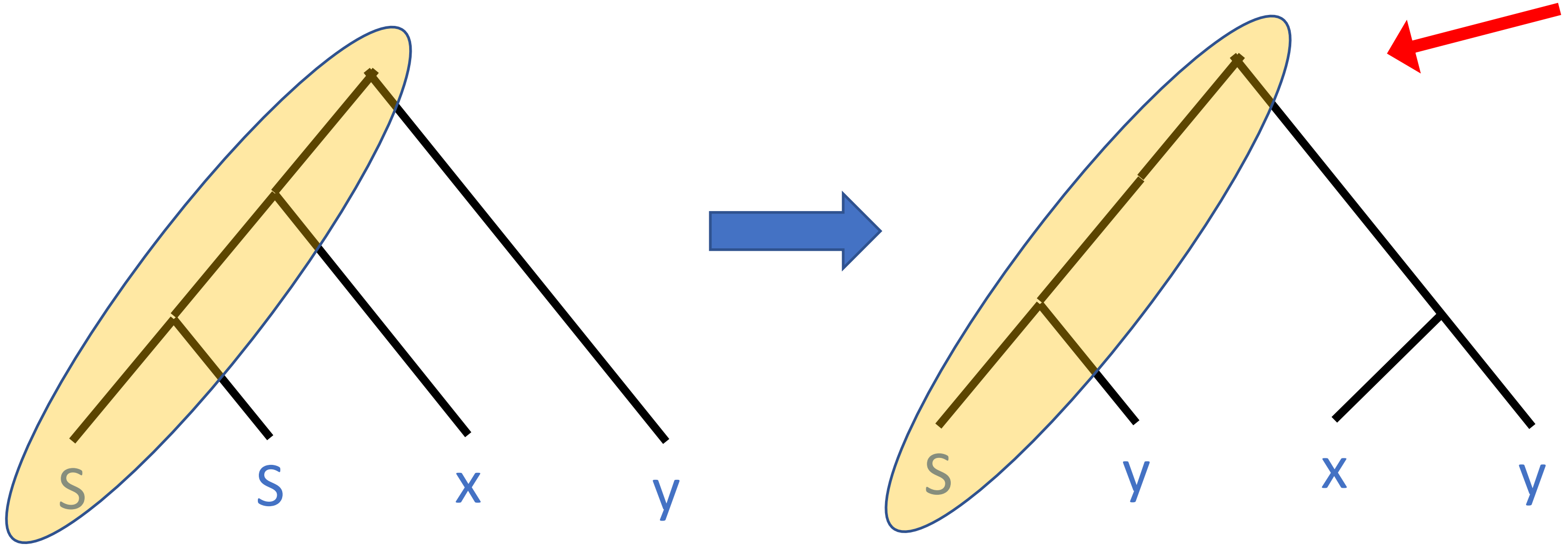
Example

S K x y → (K y) (x y) → y



Example

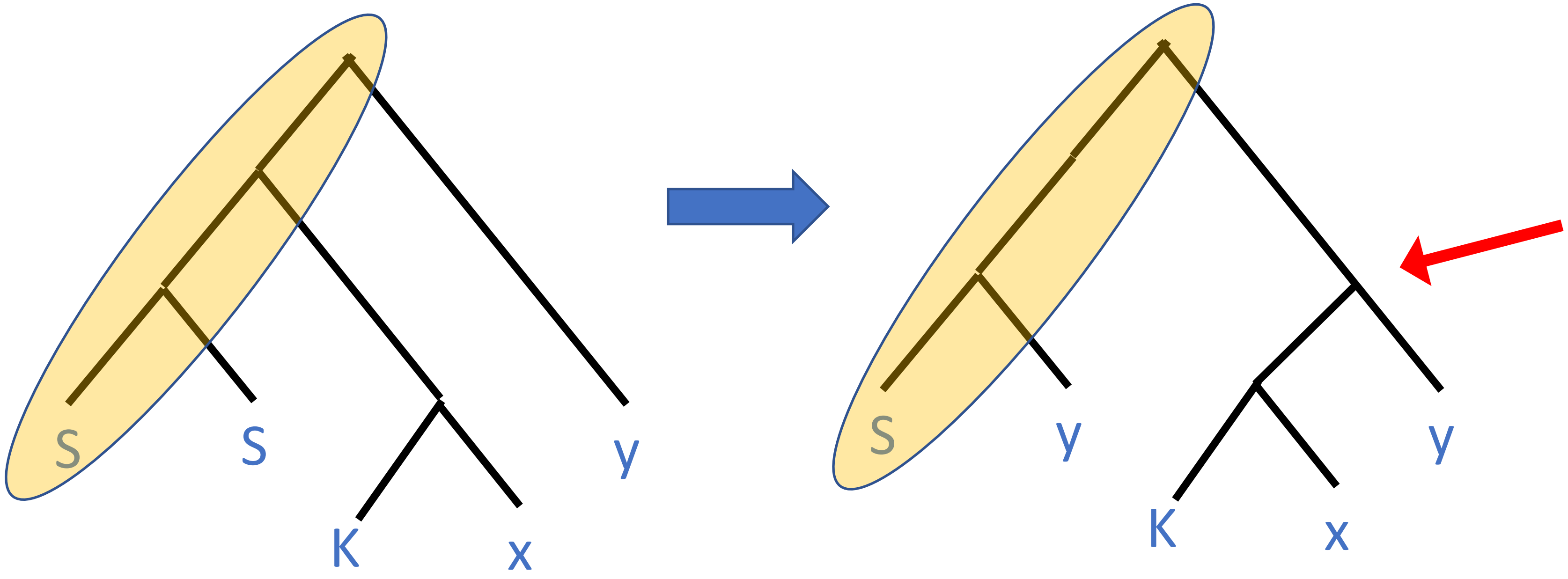
S S x y → (S y) (x y)



No rule applies because S doesn't have enough arguments, so we stop here.

Example

S S (K x) y → (S y) (K x y)

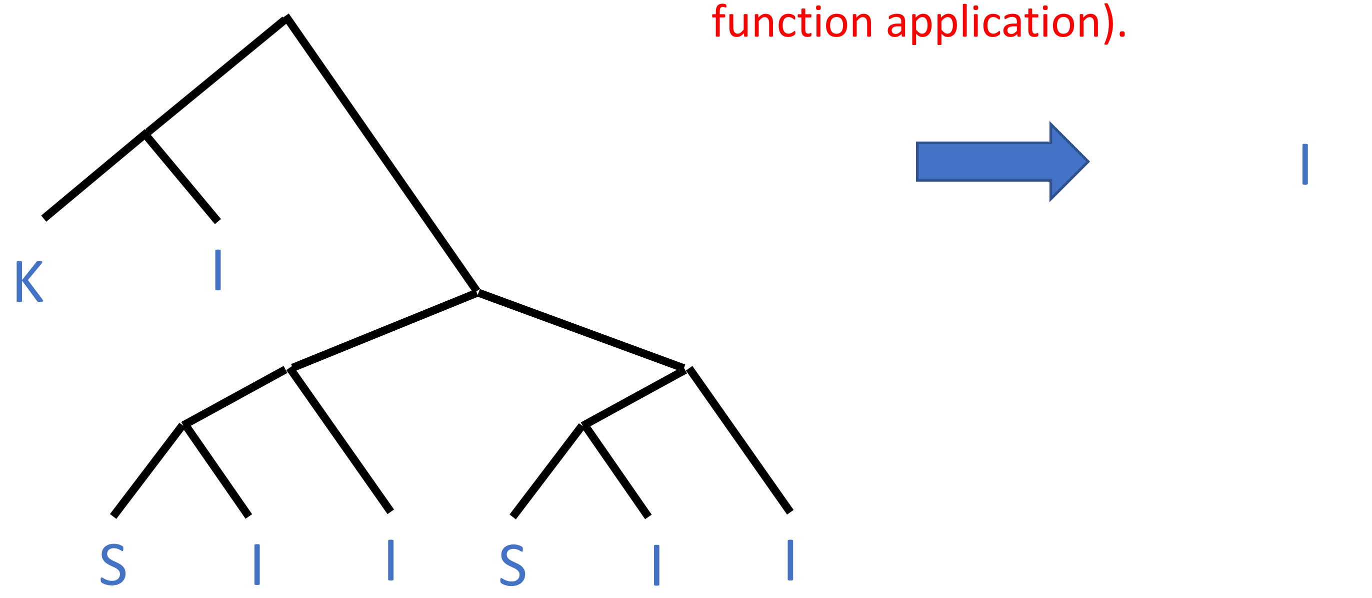


We don't rewrite here!

Why? In general, rewriting anywhere other than along the left spine may do unnecessary work or even fail to terminate.

And Another Example

Doing any reductions other than normal order may waste computation or loop forever (if we never rewrite the top-level function application).



Summary: Normal Order

- If any reduction order terminates, normal order will terminate

- Also called *lazy evaluation*
  - Only evaluate what is absolutely necessary to get an answer (if one exists)
  - In practice *call-by-value* is more popular
  - But more on that in a later lecture …

- One of the arguments for using combinator languages is parallelism
  - Doing more than one reduction at a time
  - So *not* normal order …
  - Could anything, besides non-termination, go wrong?

Confluence

- Could different choices of evaluation order change the (terminating) result of the program?

- The answer is no!

- A set of rewrite rules is *confluent* if for any expression E_0, if $E_0 \rightarrow^* E_1$ and $E_0 \rightarrow^* E_2$, then there exists E_3 such that $E_1 \rightarrow^* E_3$ and $E_2 \rightarrow^* E_3$.

Proving Confluence

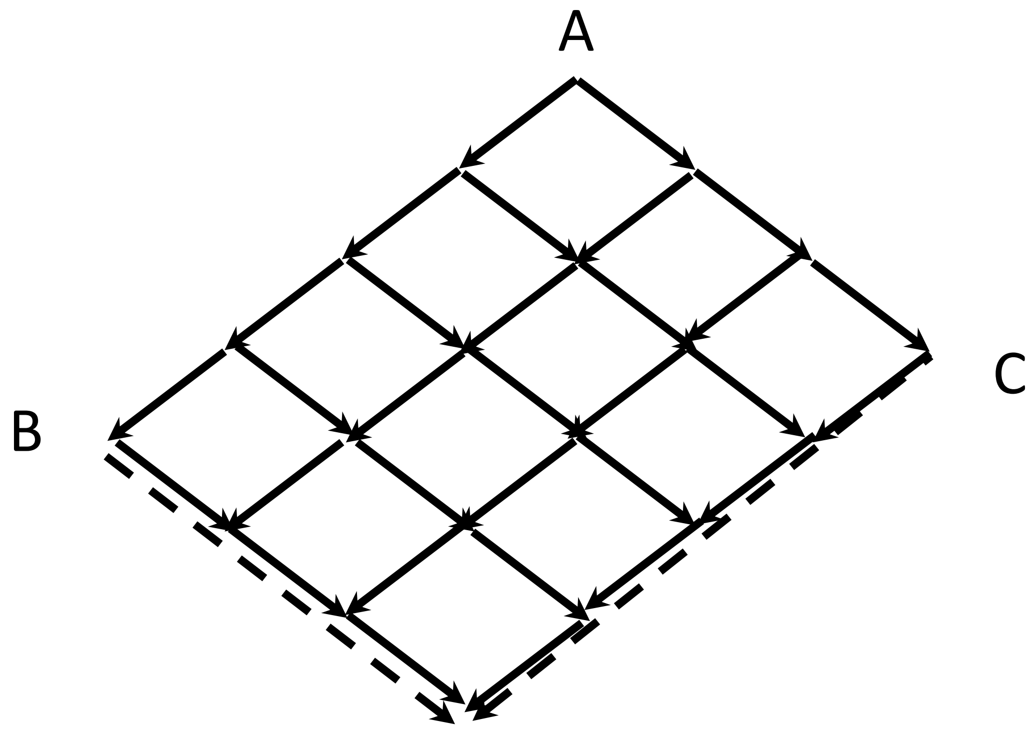
Definition:

If for all A, $A \rightarrow B$ & $A \rightarrow C$ implies there exists a D such that $B \rightarrow D$ and $C \rightarrow D$, then $\rightarrow$ has the *one step diamond property*.

Thm: If $\rightarrow$ has the one step diamond property, then $\rightarrow$ is confluent.

Proof: Assume $A \rightarrow^* X$ & $A \rightarrow^* Y$. The proof is by induction on the length of the derivations.

Diagram

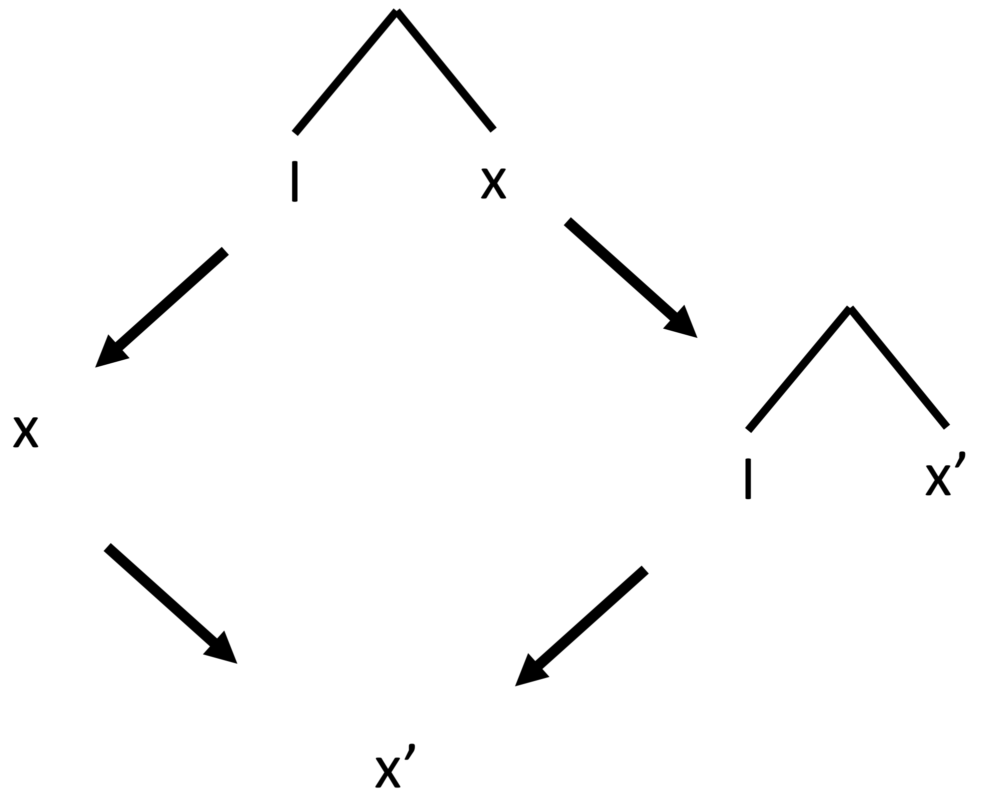


Confluence of SKI

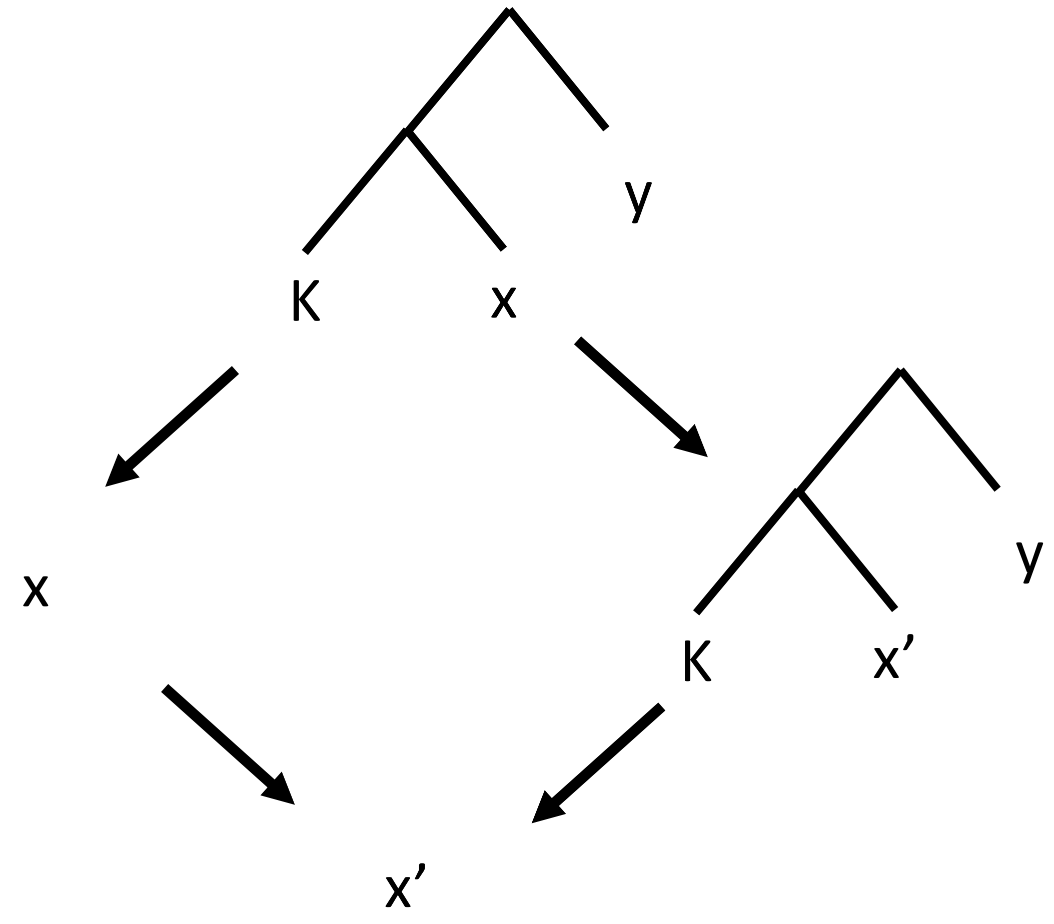
- So to show that SKI is confluent, it suffices to show it has the one step diamond property

- Note: The one step diamond property is sufficient, but not necessary, to prove confluence. But it is a very common proof method for showing the confluence of rewrite systems.

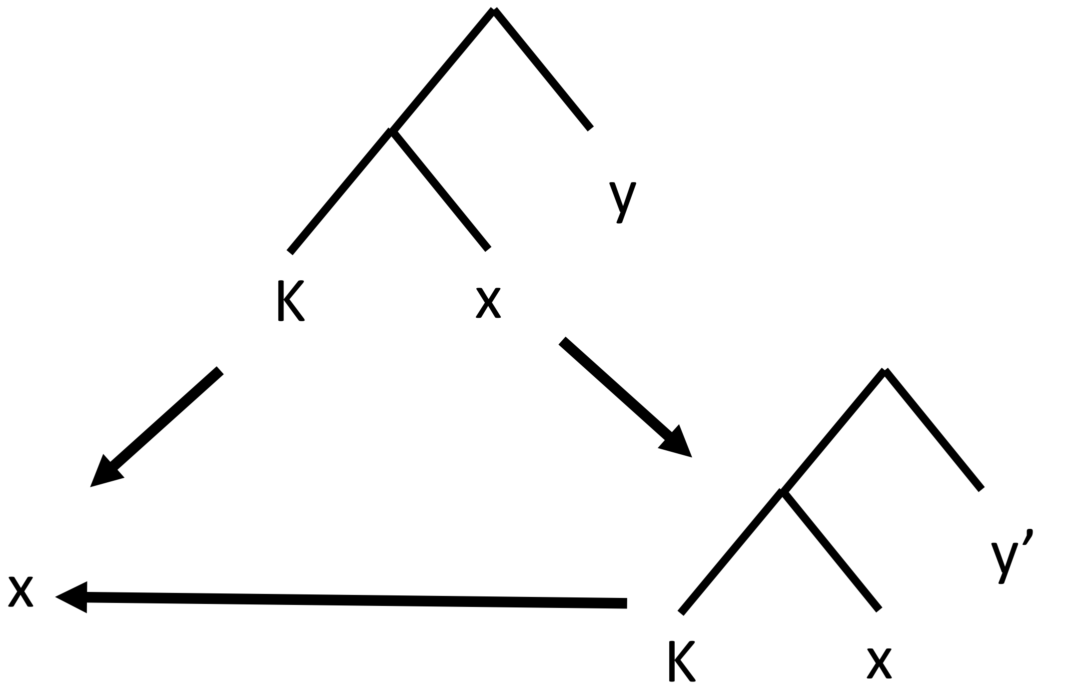
Confluence of SKI: Case I x



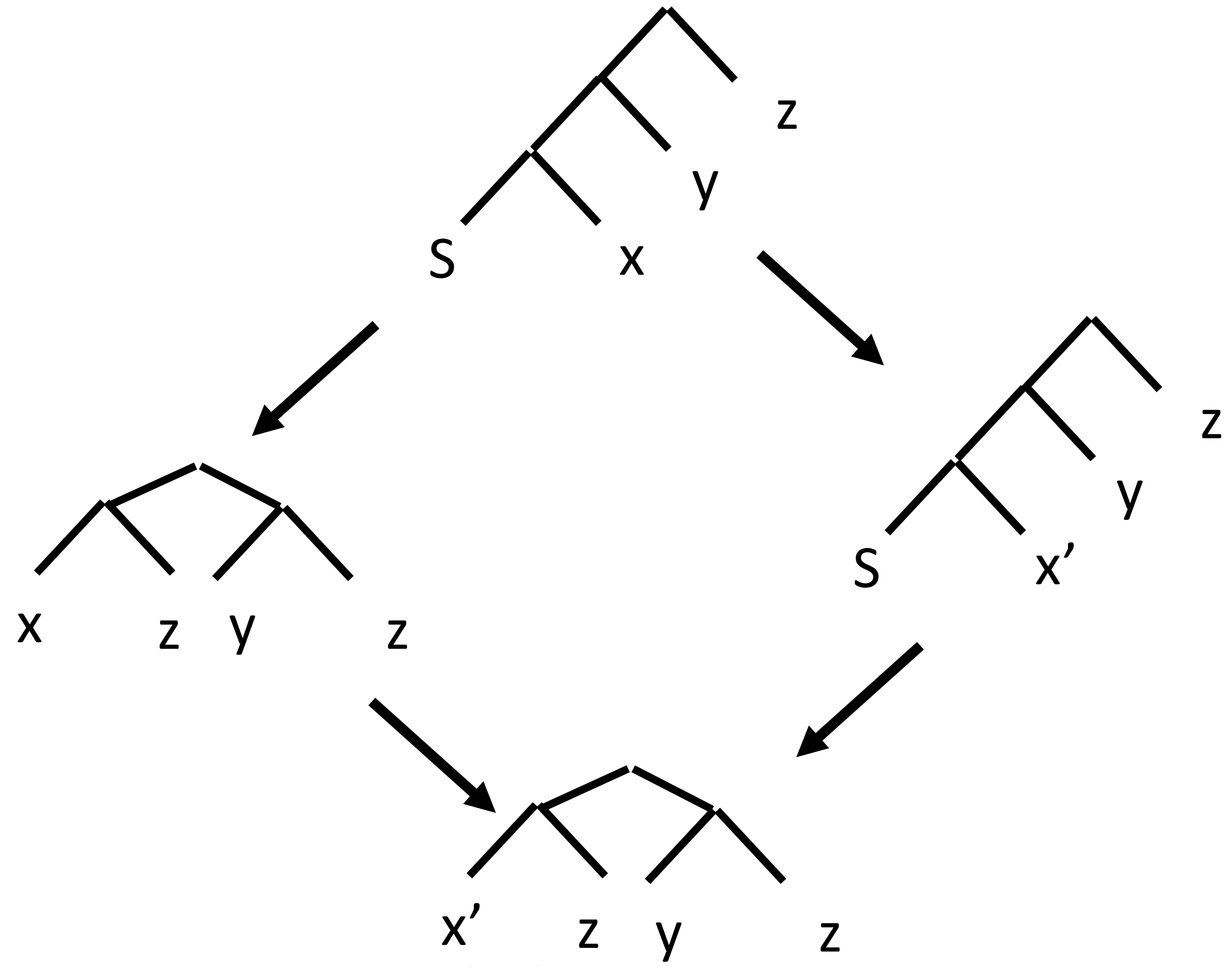
Case K x y   (1 of 2)



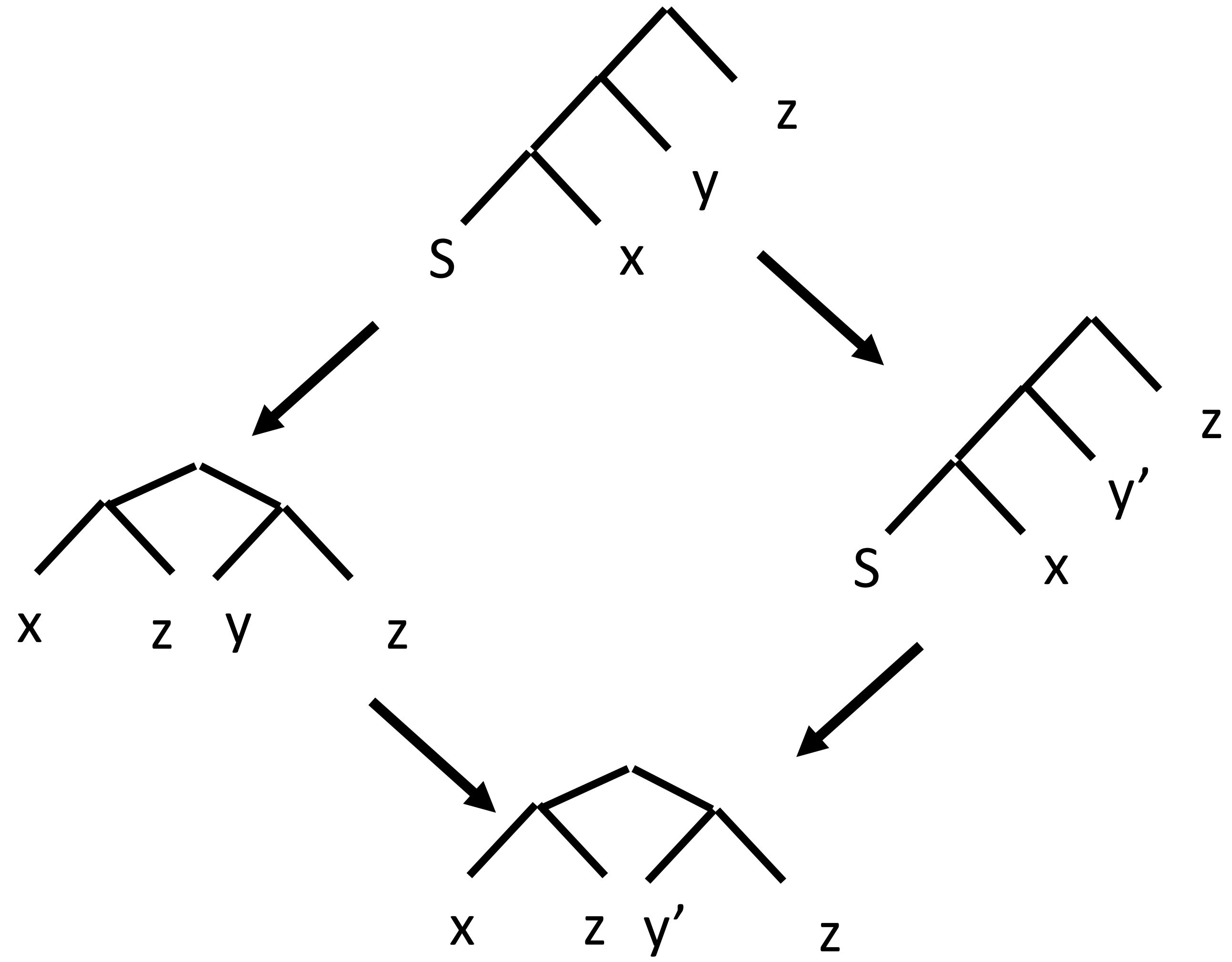
Case K x y (2 of 2)



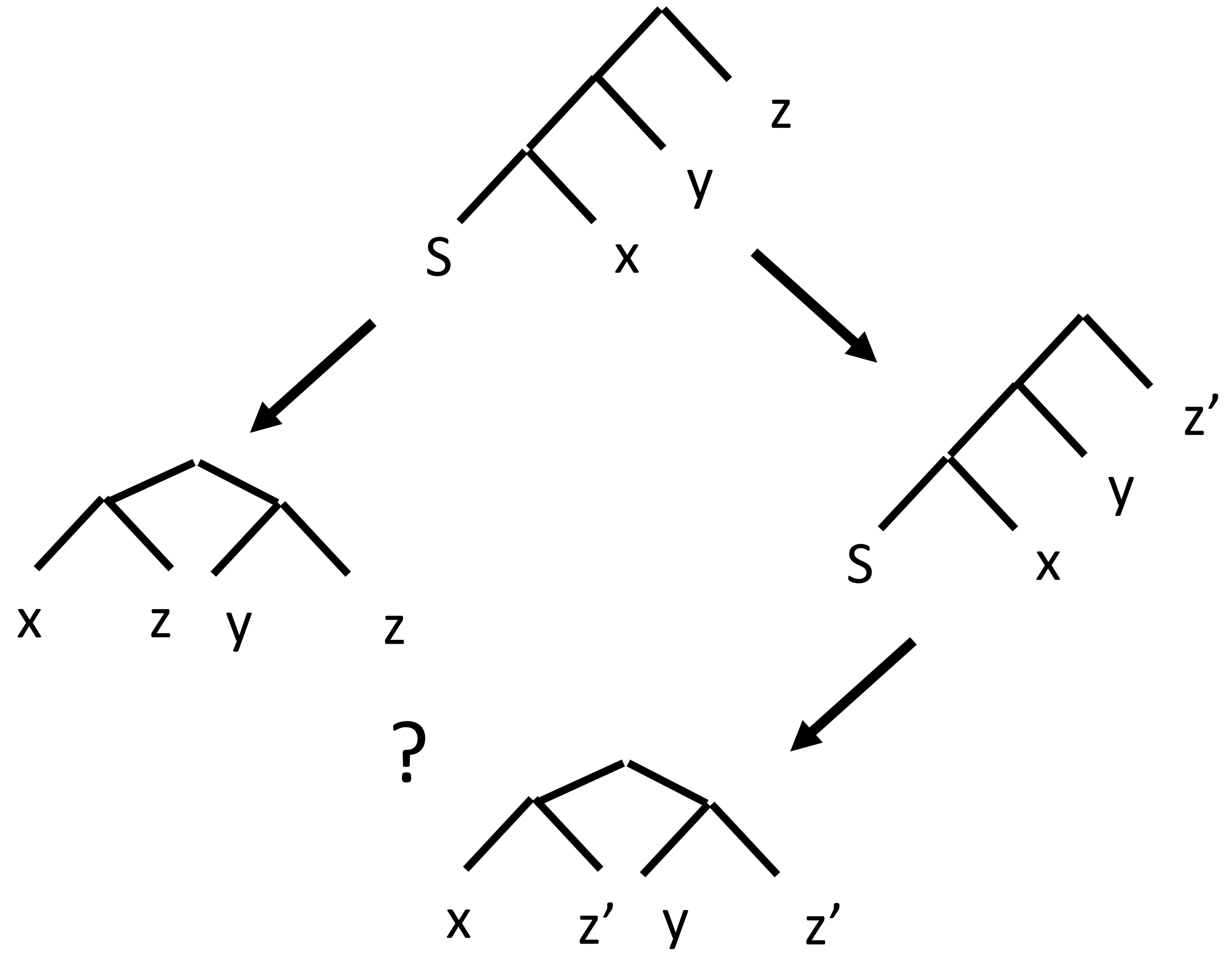
Case S x y z (1 of 3)



Case S x y z (2 of 3)



Case S x y z   (3 of 3)



A New Relation

- $\rightarrow$ doesn't have the one step diamond property!
  - Because S copies its third argument

- But all is not lost!
  - If we can find another rewrite relation that is equivalent to $\rightarrow$ and has the one step diamond property, then that will show that $\rightarrow$ is confluent

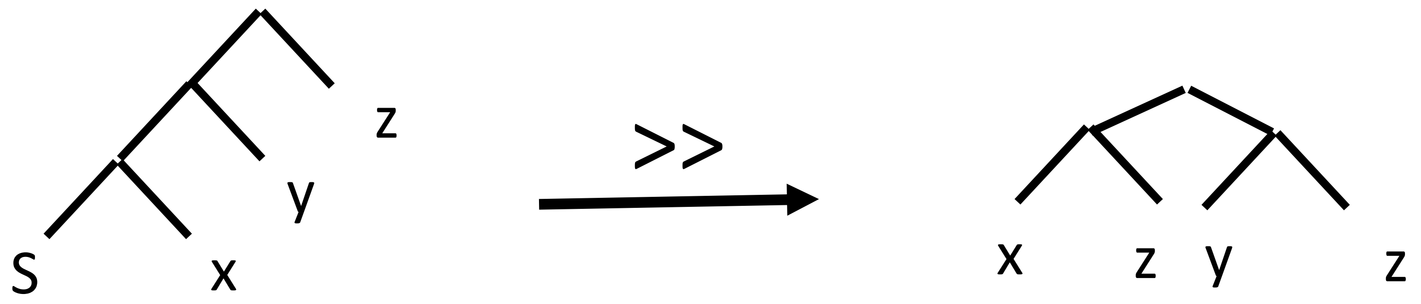
- Define $X >> Y$ if
  - $X \rightarrow Y$ via a rewrite at the root node
  - $X = A\ B$, $Y = A'\ B'$ and $A >> A'$ and $B >> B'$

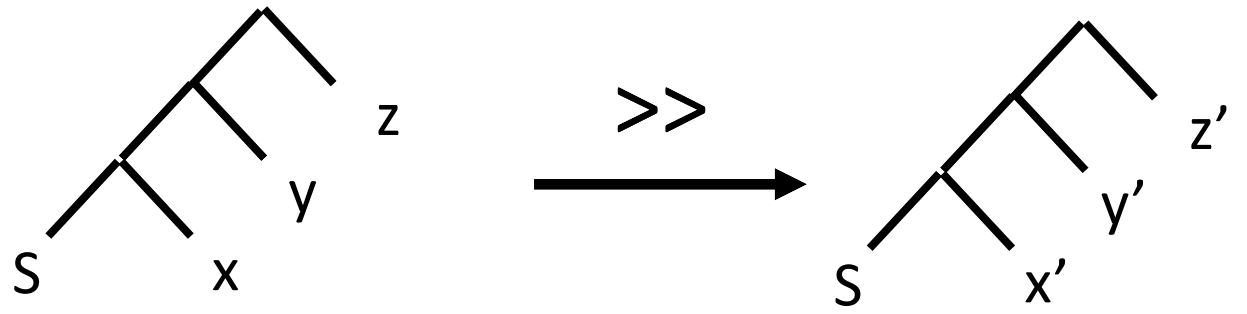
- Easy to see that $A >>^* B$ iff $A \rightarrow^* B$

- Thm: $>>$ has the one step diamond property.

First, What Does >> Do?

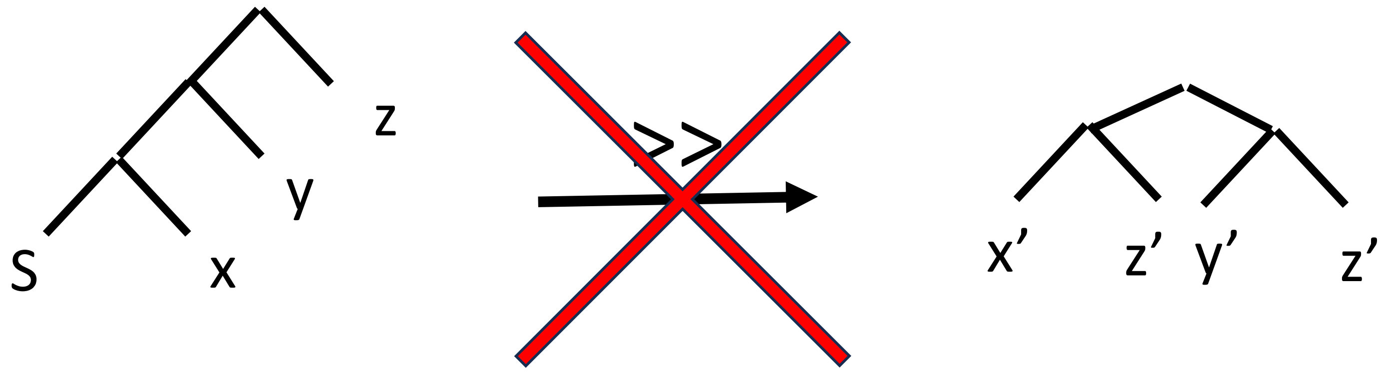
- Allows multiple rewrites as long as they are in *independent subtrees*



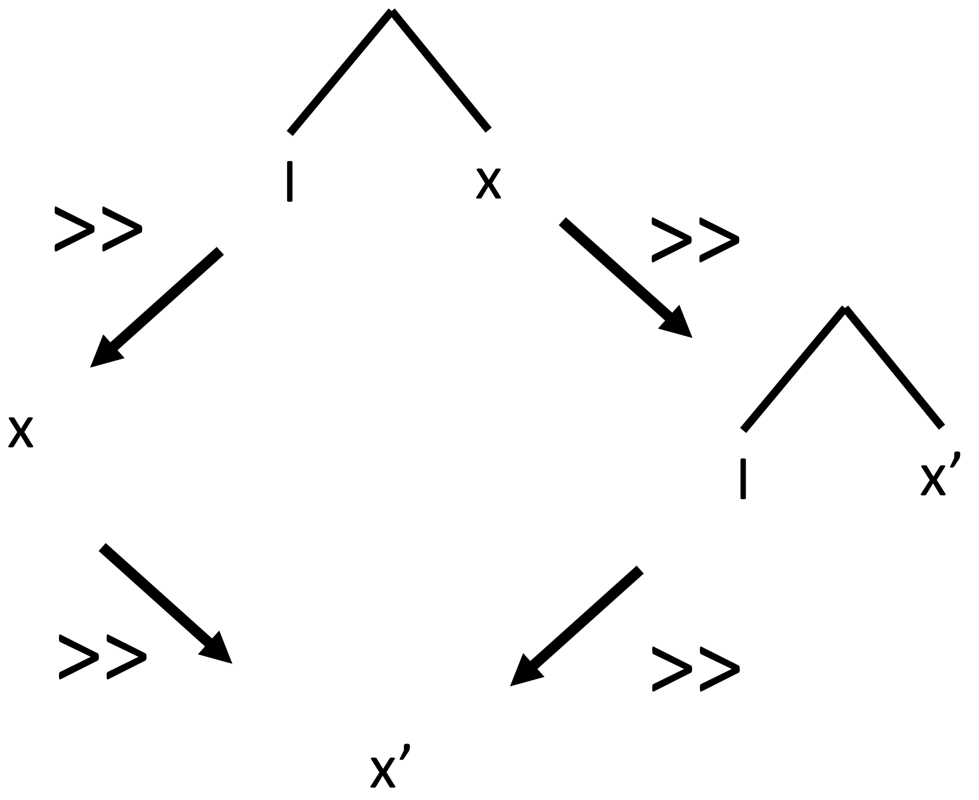


What Does >> Not Do?

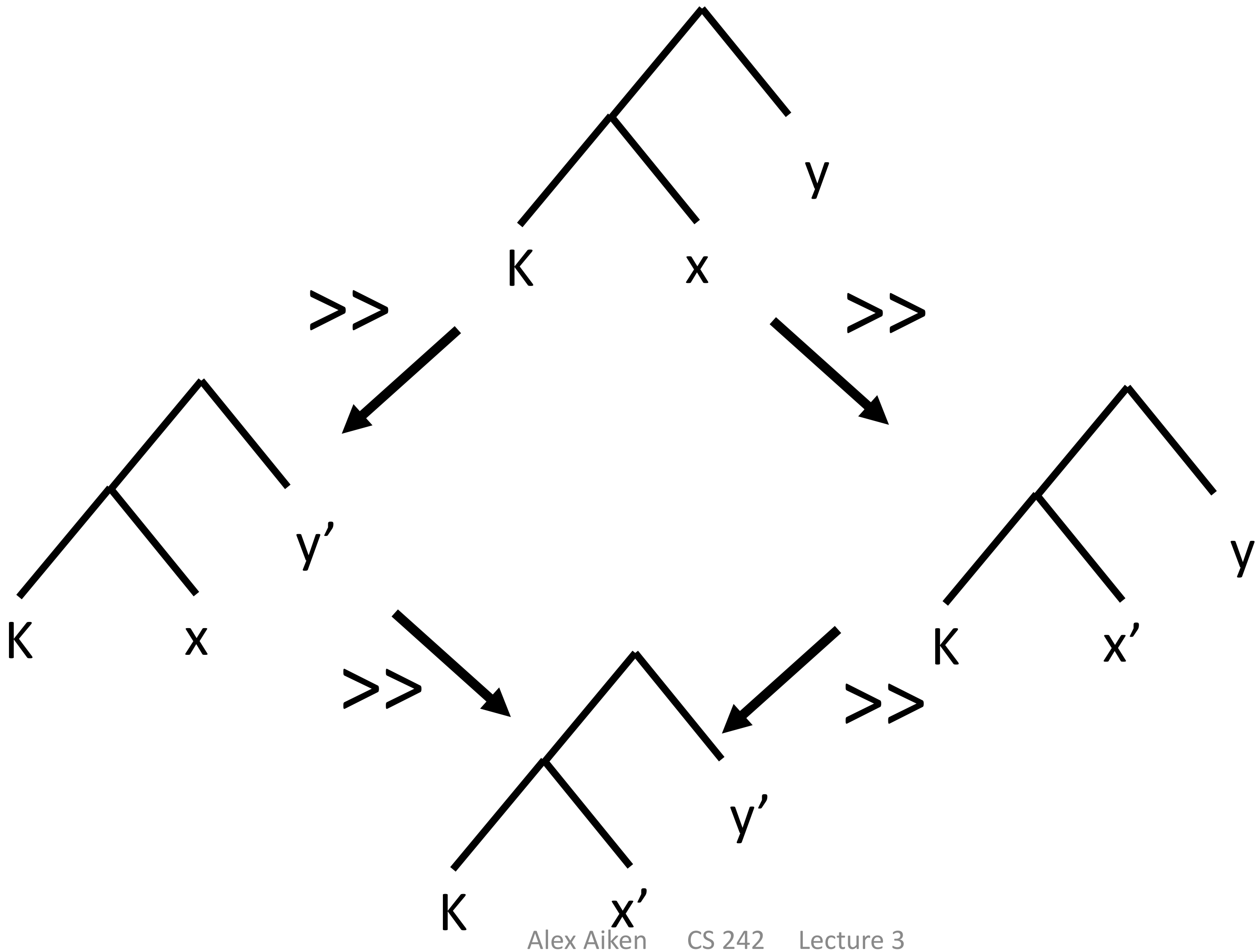
- Multiple rewrites must be in *independent subtrees*



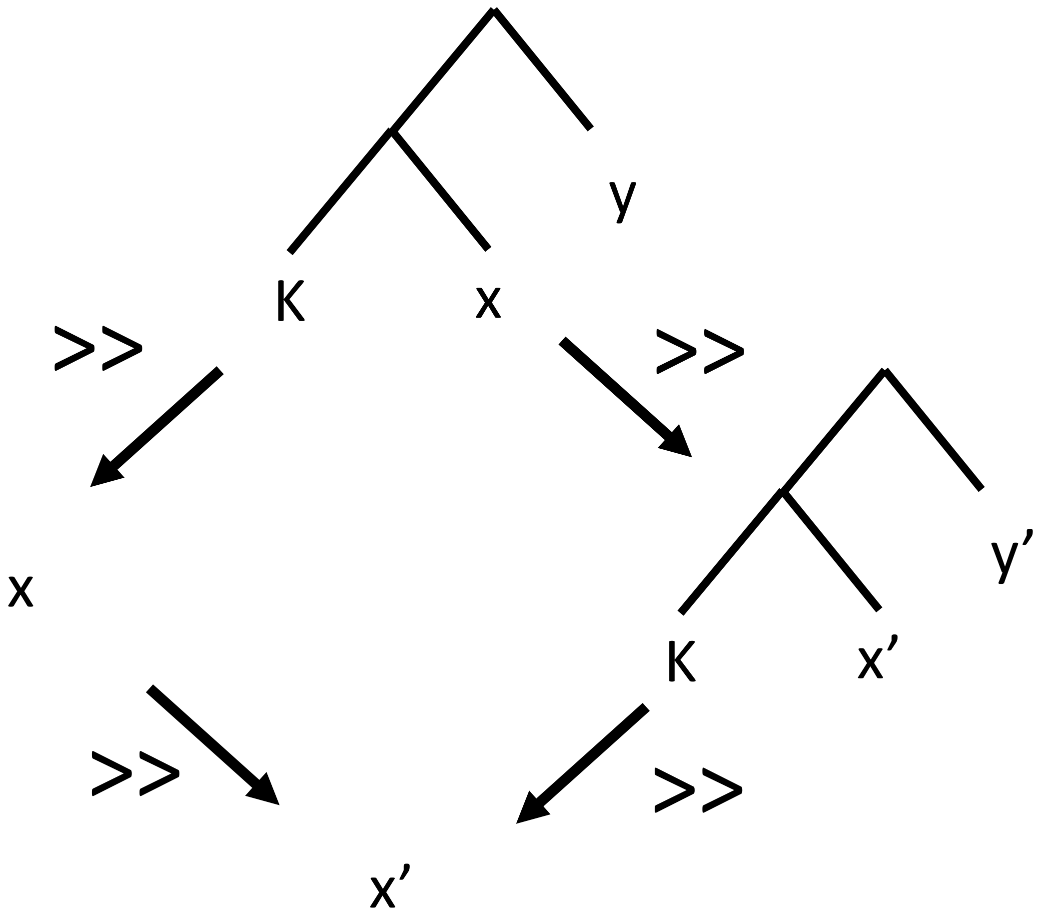
Case l x



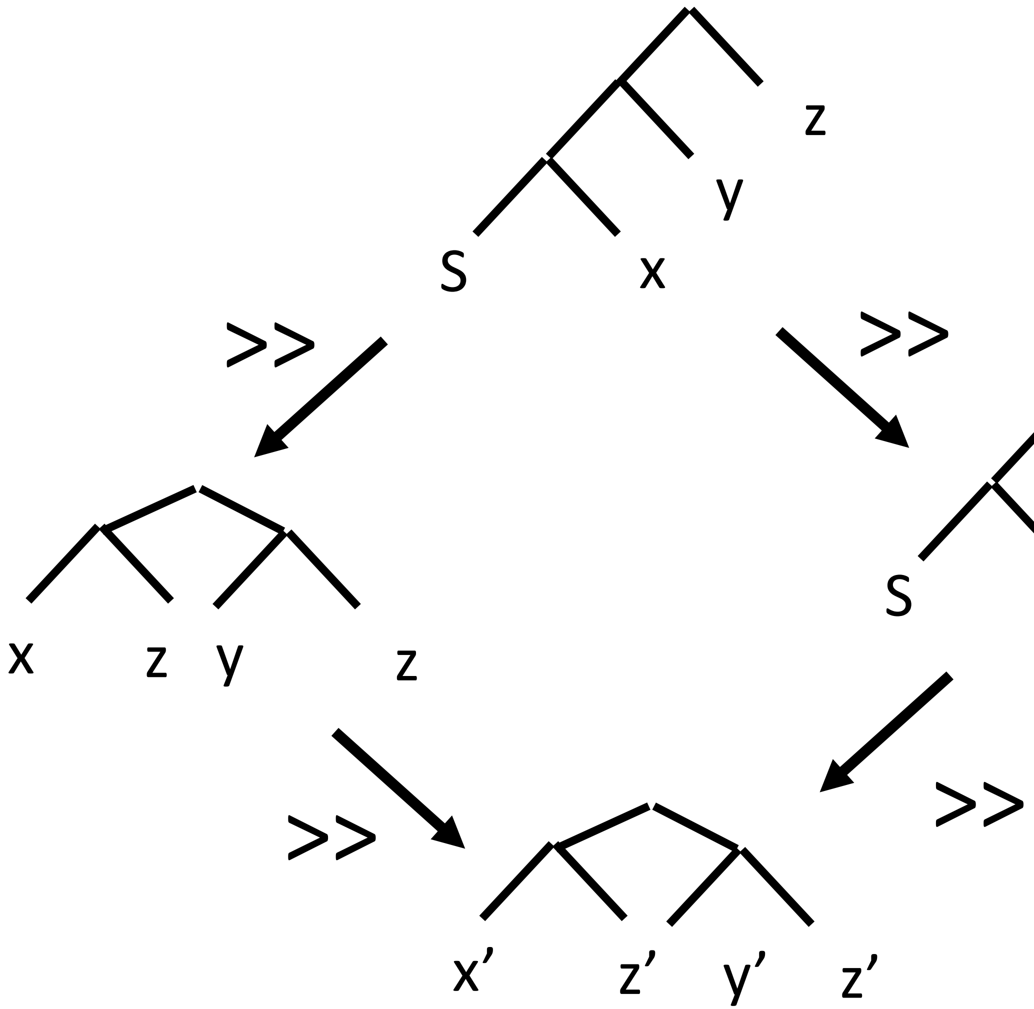
Case K x y (Boring Case)



Case K x y (Interesting Case)



Case S x y z (Interesting Case Only …)



Discussion

- Combinator calculus has the advantage of having no variables
  - Compositional!

- All computations are local rewrite rules
  - Compute by pattern matching on the shape and contents of a tree
  - All operations are local and there are few cases
  - No need to worry about variables, scope, renaming ...

- Many proofs of properties are easier in combinator systems
  - E.g., confluence

Discussion

- Combinator calculus has the disadvantage of having no variables
- Consider the S combinator again: $S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$
- Note how z is ``passed'' to both x and y before the final application
- In a combinator calculus, this is the *only* way to pass information
  - In a language with variables, we would simply stash z in a variable and use it in x and y as needed
  - In a combinator-based language, z must be explicitly passed down to all parts of the subtree that need it

Discussion

- Thus, what can be done in one step with a variable requires many steps (in general) in a pure combinator system

- Why does this matter?
  - SKI calculus is not a direct match to the way we build machines
    - Our machines have memory locations and can store things in them
    - Languages with variables take advantage of this fact

Discussion

- Another advantage of combinators is working at the function level
  - Avoid reasoning about individual data accesses

- A natural fit for parallel and distributed bulk operations on data
  - Map a function over all elements of a dataset
  - Reduce a dataset to a single value using an associative operator
  - Transpose a matrix
  - Convolve an image
  - ...

- Note that in parallel/distributed operations, variables can be a problem ...

Summing UP: SKI and Beyond

History

- SKI calculus was developed by Schoenfinkel in the 1920’s
  - One of Hilbert’s students
- Rediscovered by Haskell Curry in the 1930’s
- The properties of SKI were known before any computers were built ...

History

- First combinator-based programming language was APL
  - Designed by Ken Iverson in the 1960's
- Designed for expressing pipelines of operations on bulk data
  - Array programming
  - Basic data type is the multidimensional array
- Trivia: Special APL keyboards accommodated the many 1 character combinators
  - APL programs tend to be extremely concise
- Highly influential
  - On functional programming (several languages)
  - And array programming (Matlab, R, NumPy)

```
{(+⌿ω)÷≢ω}
```

Summary

- Combinator calculi are among the simplest formal computation systems
- Also important in practice for array/collection programming
  - Where thinking in terms of bulk operations with built-in iteration is useful
- Not used as a model for sequential computation
  - Where we often want to take advantage of temporary storage/variables
- Combinators are also important in program transformations
  - Much easier to design combinator-based transformation systems
  - Some compilers (Haskell's GHC) even translate into an intermediate combinator-based form for some optimizations

Next Time

- Another primitive calculus
- The lambda calculus
  - The basis of functional programming languages
  - And much of modern type systems