Combinators II

CS242

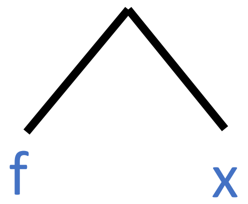
Lecture 3

Review

- Function application written as space/juxtaposition

f x

- Programs as trees



SKI Calculus

$I\ x \to x$ Identity function

$K\ x\ y \to x$ Constant functions

$S\ x\ y\ z \to (x\ z)\ (y\ z)$ Generalized function application

Factorial

- 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)
```

The Abstraction Algorithm

Transform a function definition with variables f x = E

Into a combinator f = A(E,x)

- Where A(E,x) x = E
- And A(E,x) doesn't use x

Allows us to define only the fully applied function case, and then calculate the combinator

A(x,x) = I

A(E,x) = K E   if x does not appear in E

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

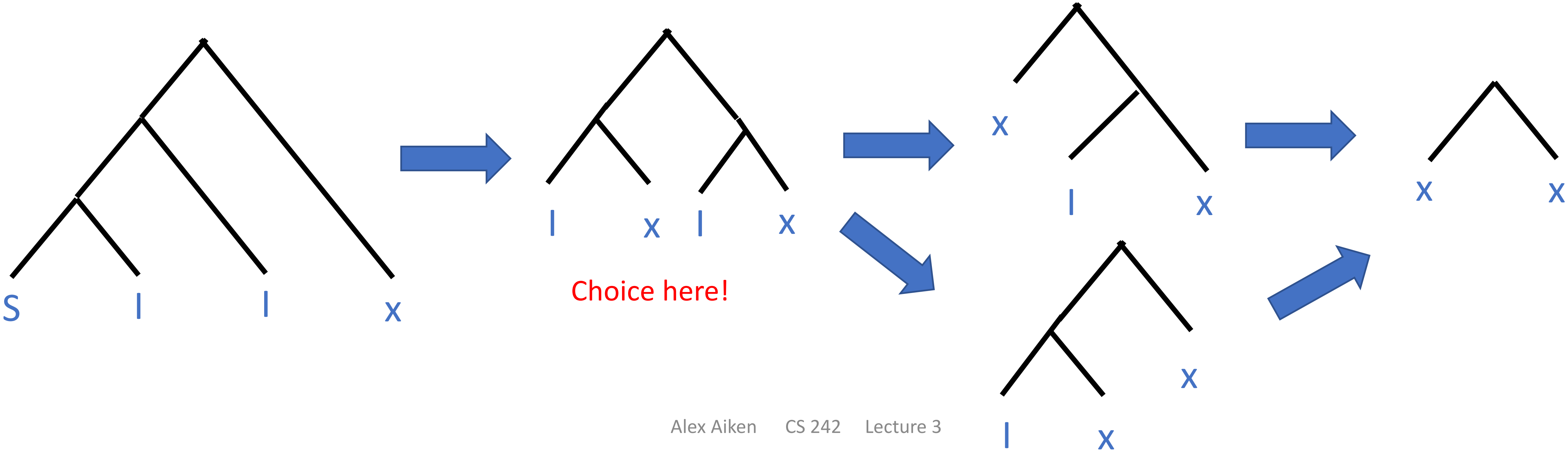
A(E1 E2,x) = c2 A(E1,x) E2   if x does not appear in E2

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

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

Order of Evaluation

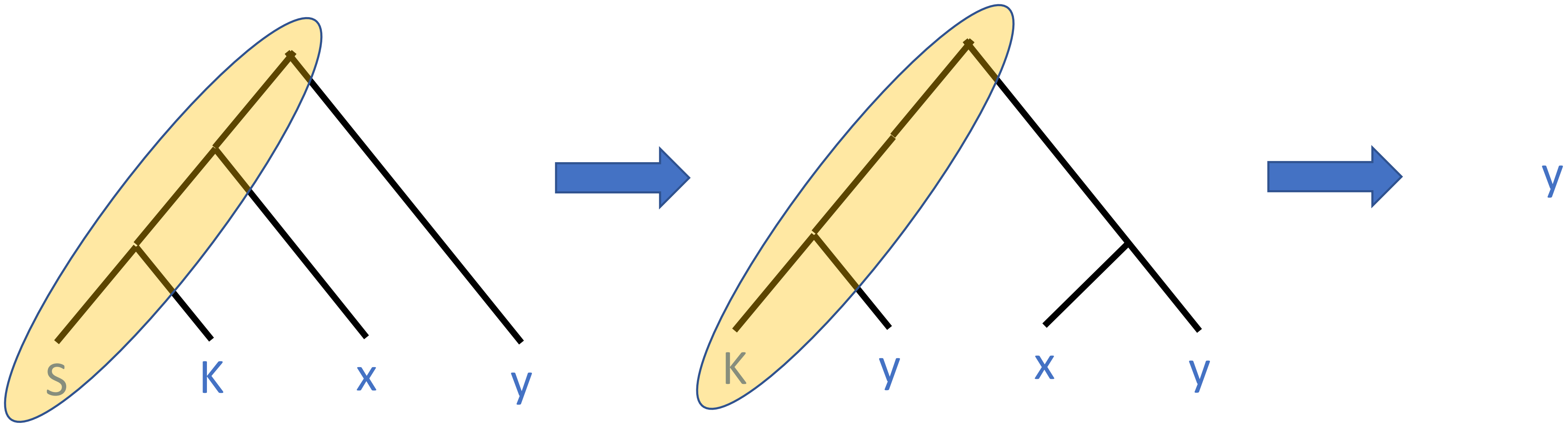
What is a good reduction strategy?

A Standard Choice

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

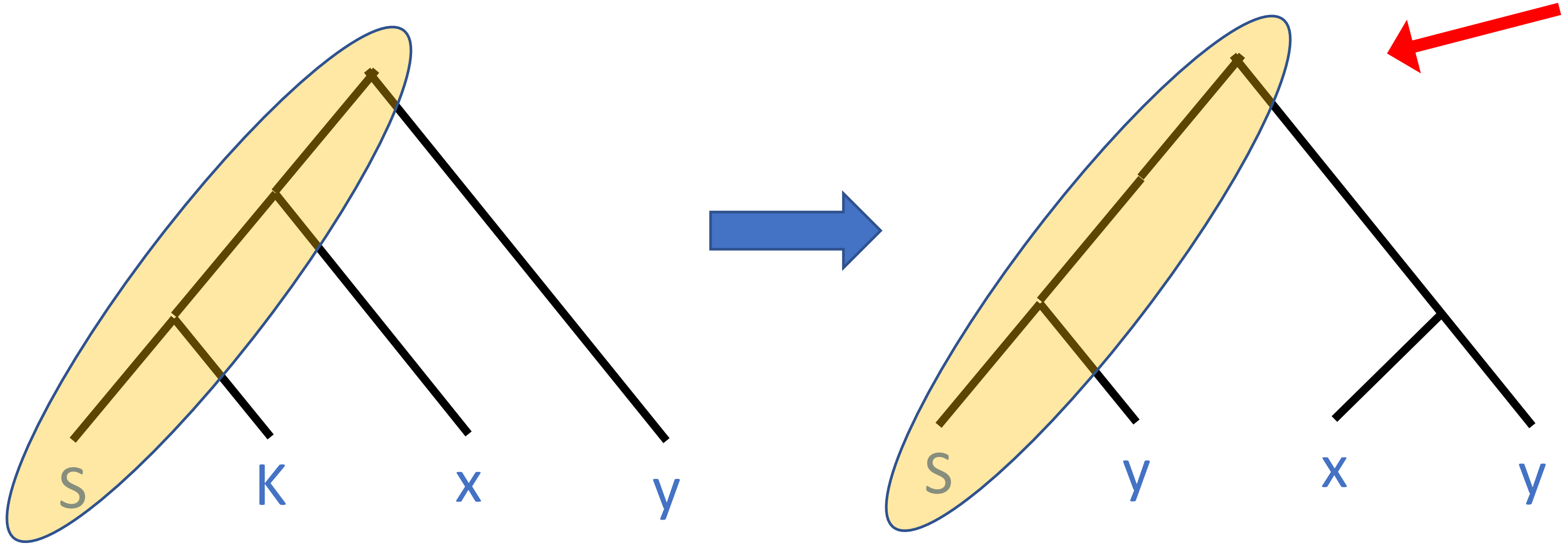
Example

S K x y → (K y) (x y) → y



Example

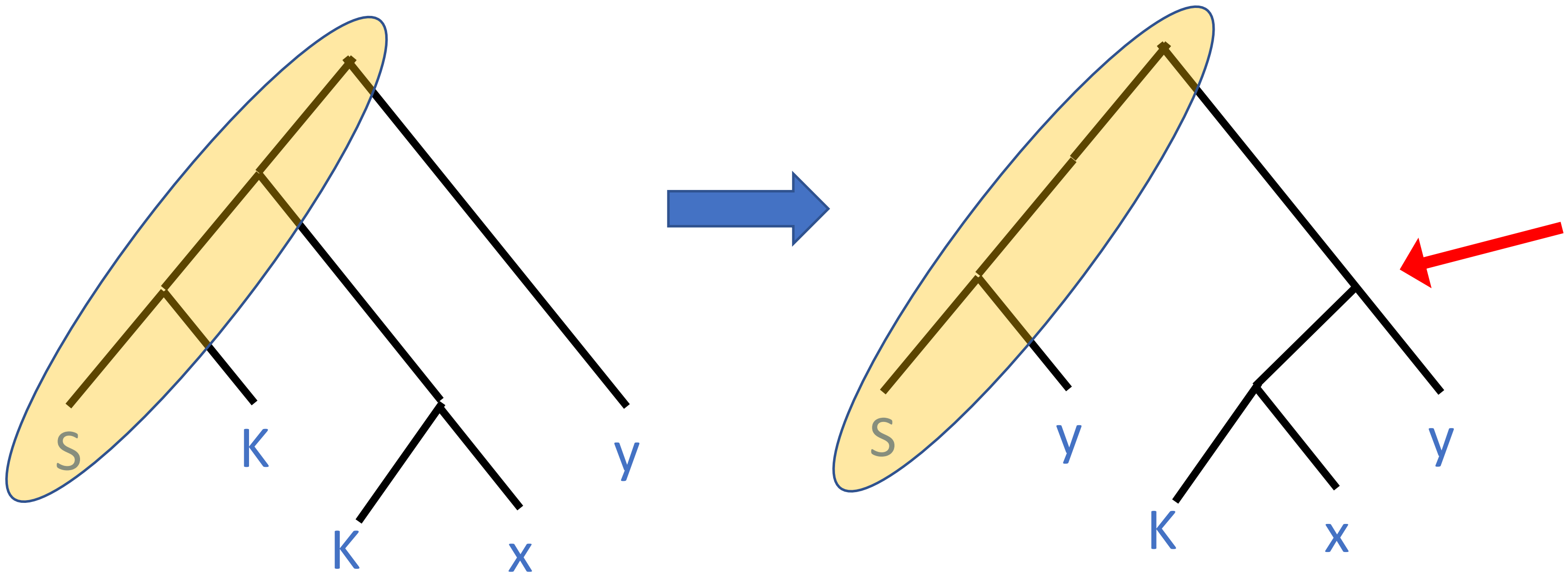
$$S\ S\ x\ y \rightarrow (S\ y)\ (x\ y)$$



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

Example

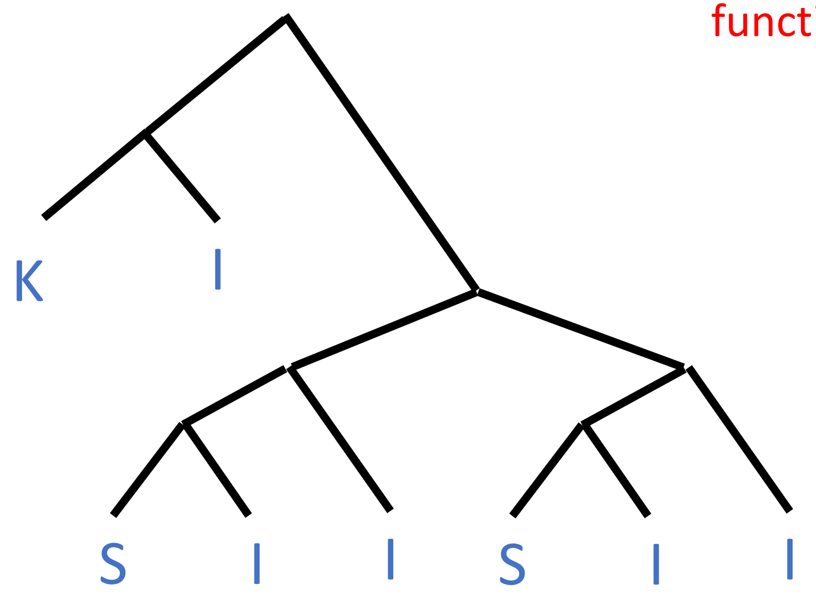
S S (K x) y → (S y) (K x y)



We don't rewrite here!

Why? In general, rewriting anywhere other than the leftmost function 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).

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

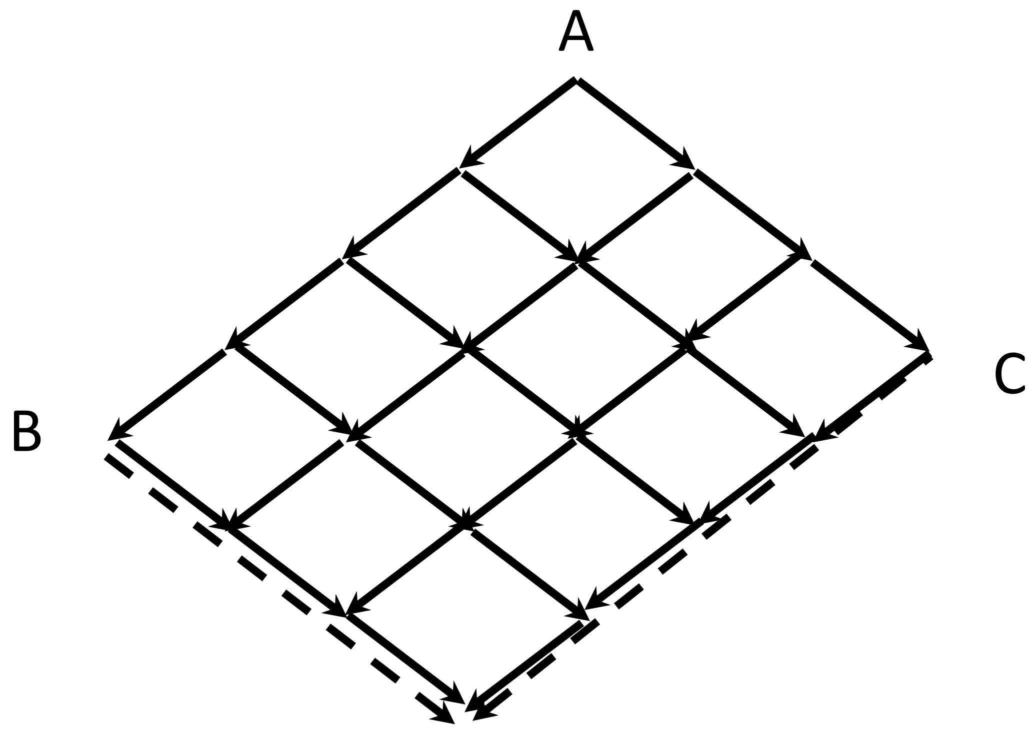
Definition:

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

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

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

Diagram

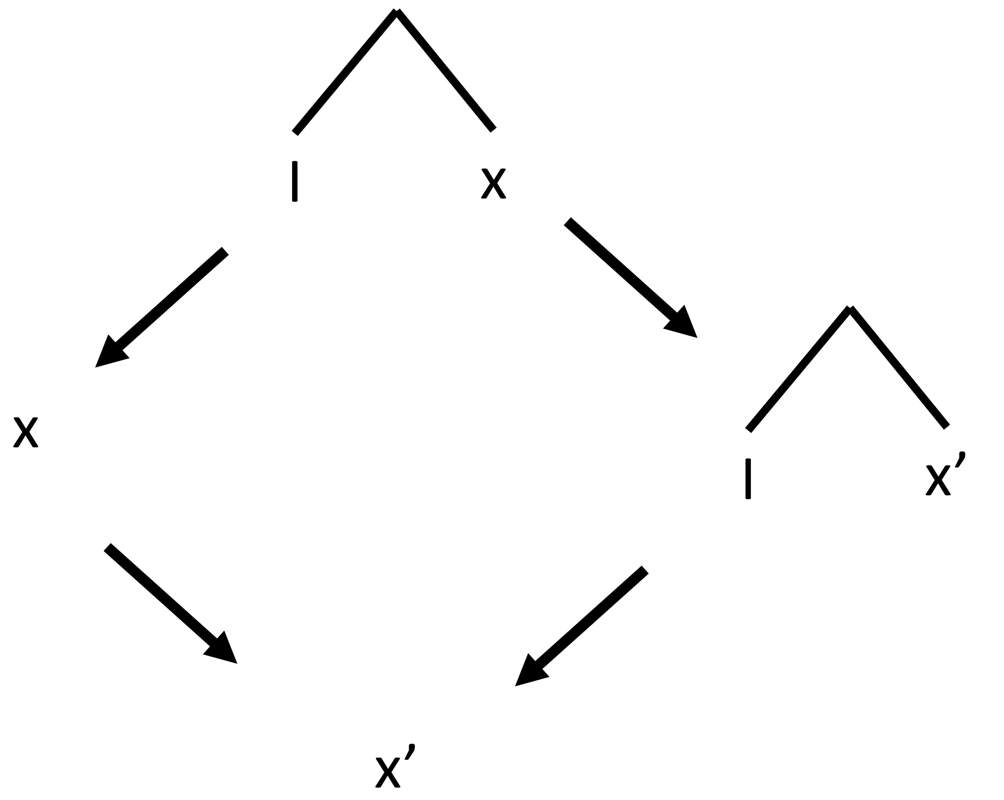


Confluence of SKI

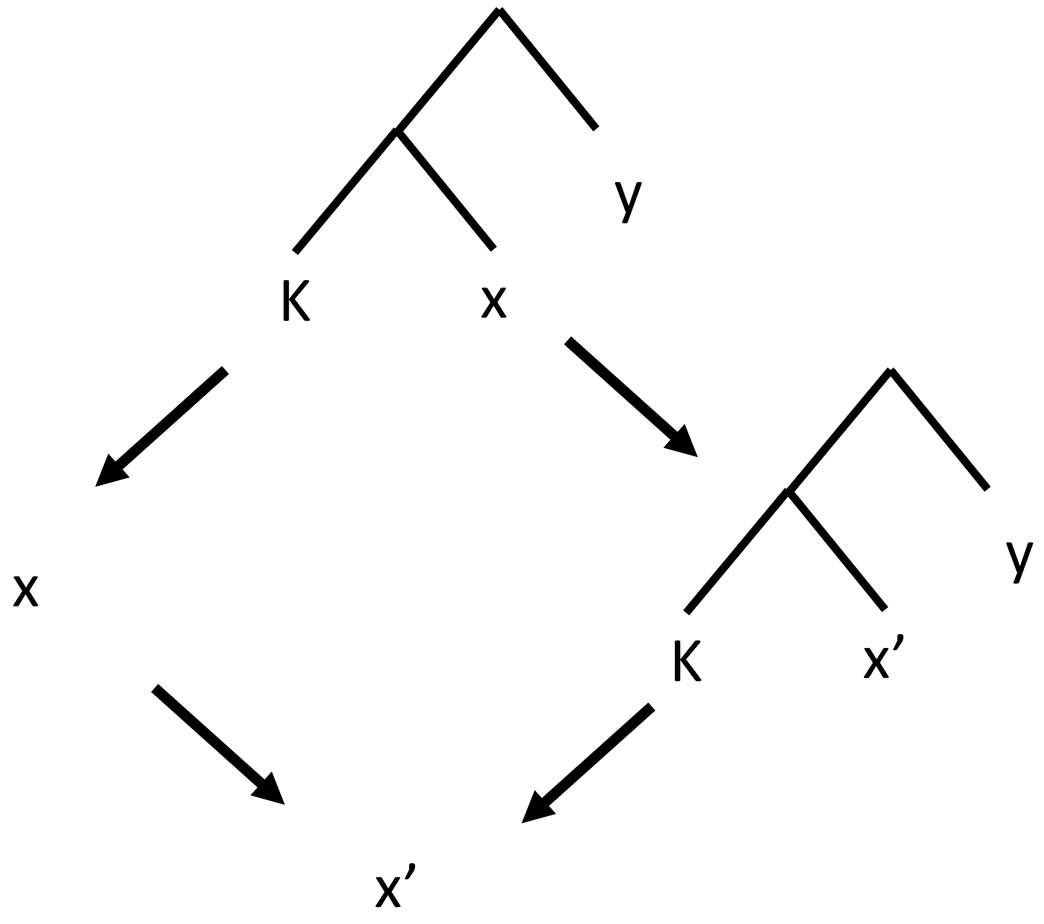
- So to show that SKI is confluent, it suffices to show it has the one step diamond property

- 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.

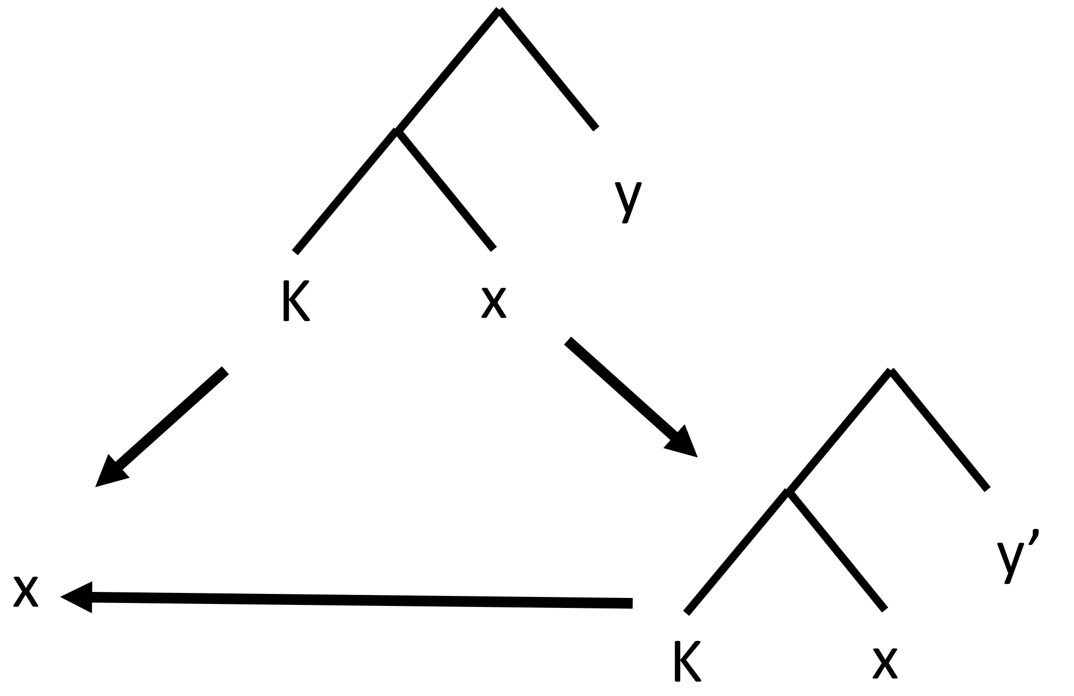
Confluence of SKI: Case I x



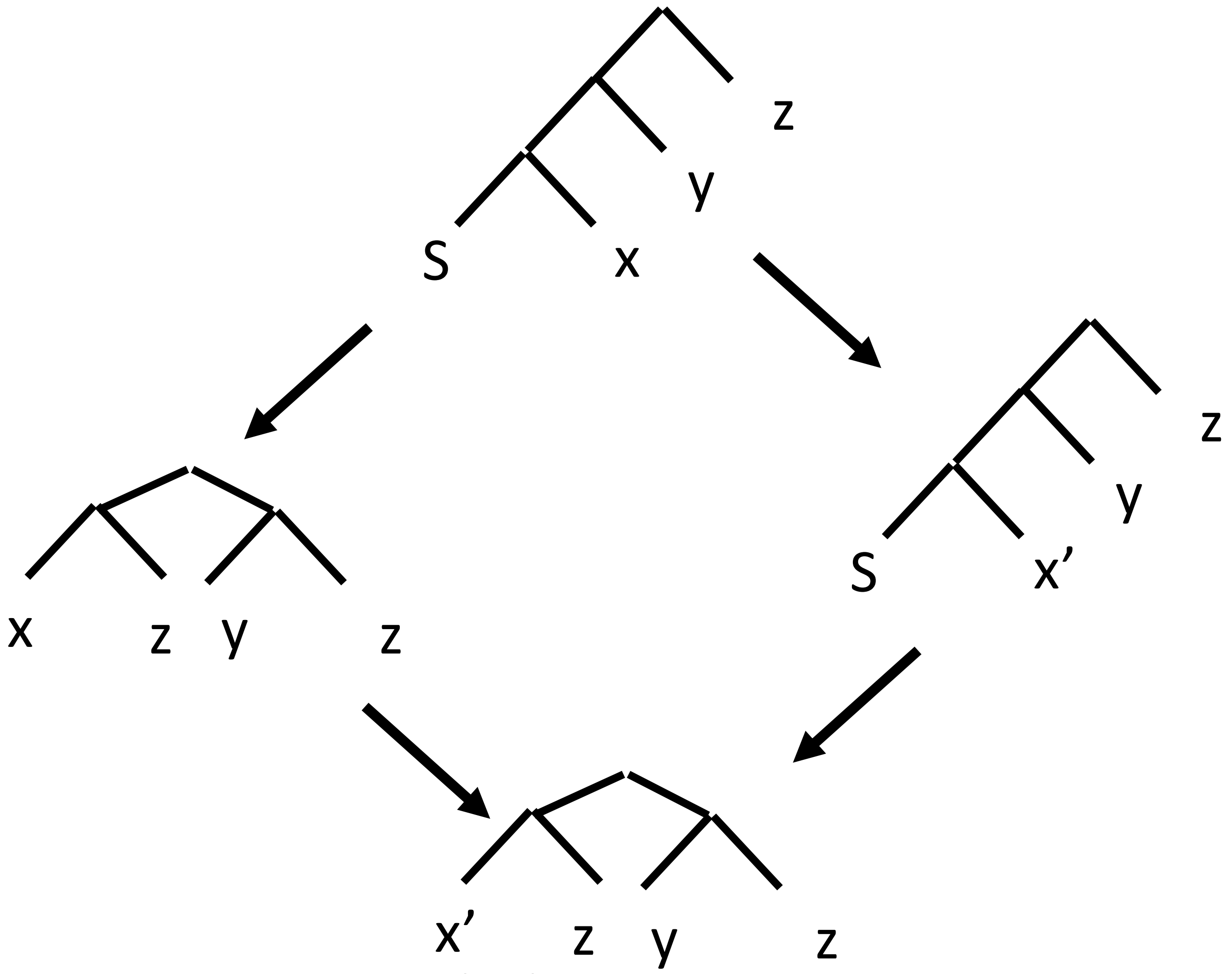
Case K x y   (1 of 2)



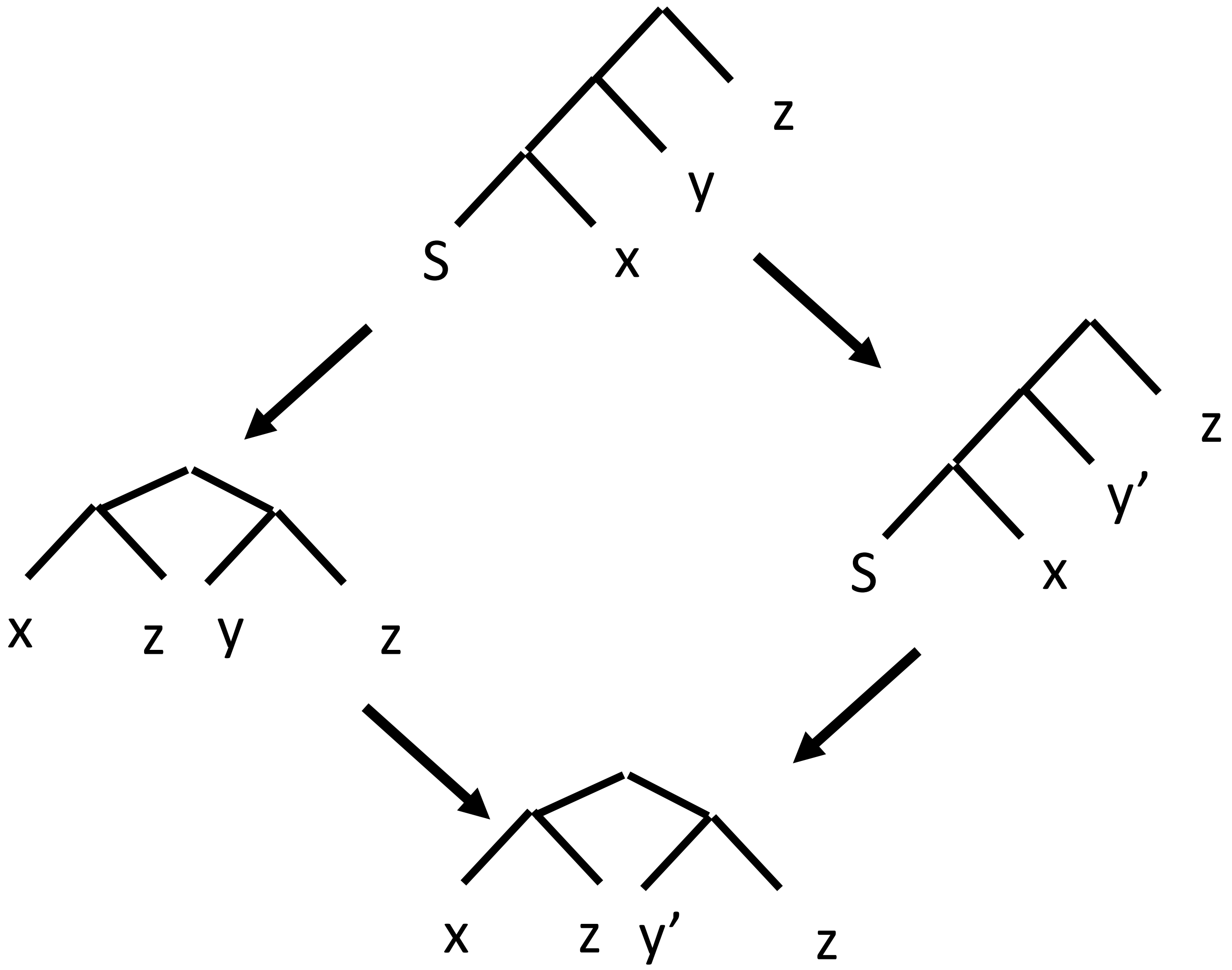
Case K x y (2 of 2)



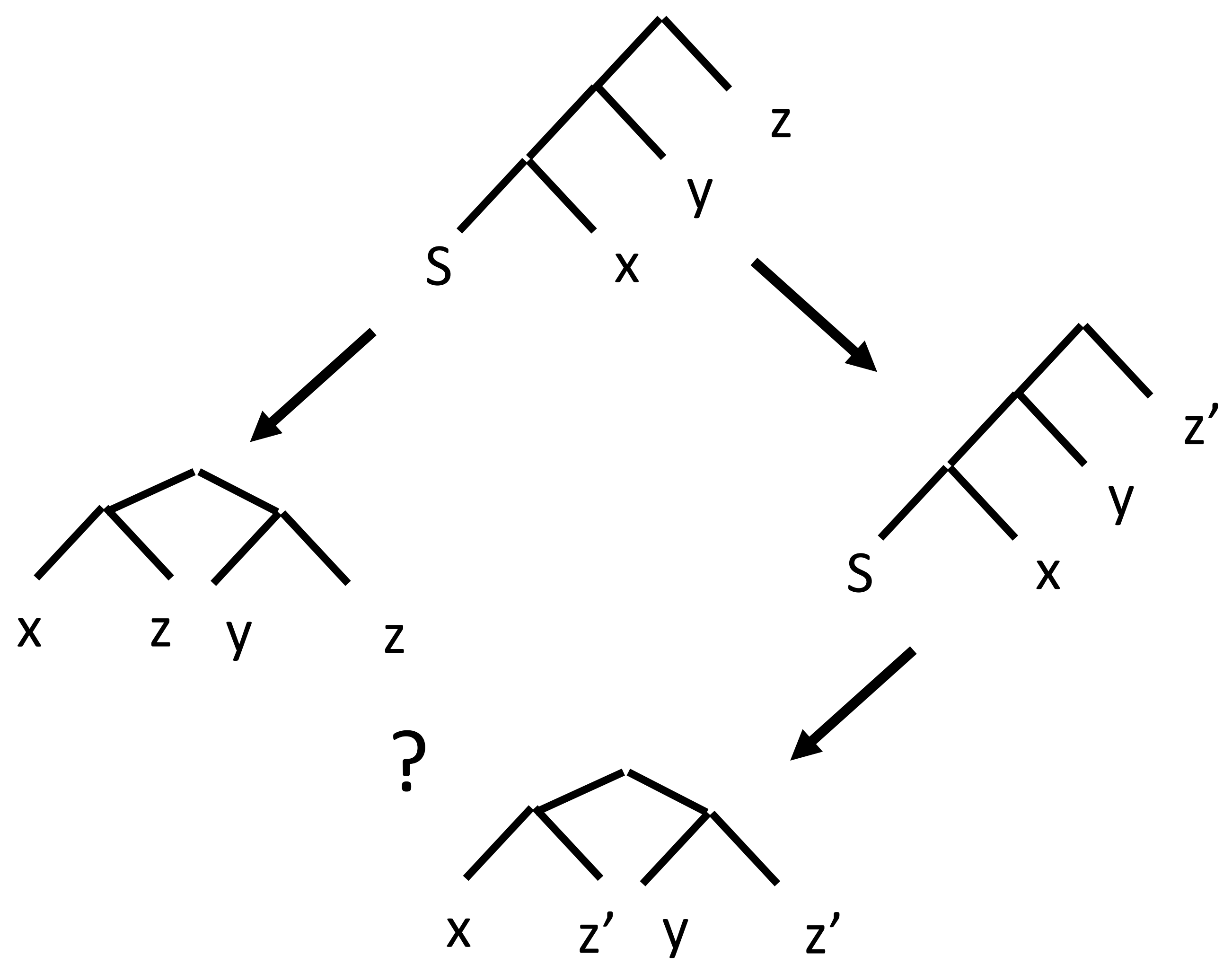
Case S x y z (1 of 3)



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

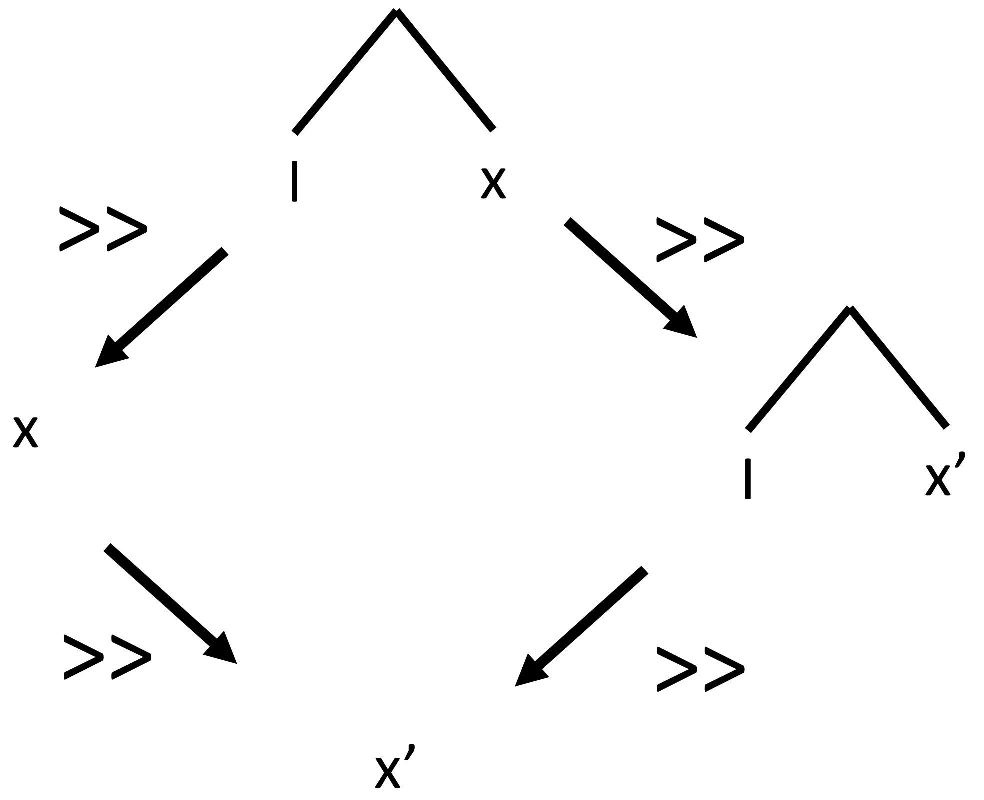
- But all is not lost!
  - If we can find another rewrite relation that is equivalent to $\rightarrow$ and has the one step diamond property, then that will show that $\rightarrow$ is confluent

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

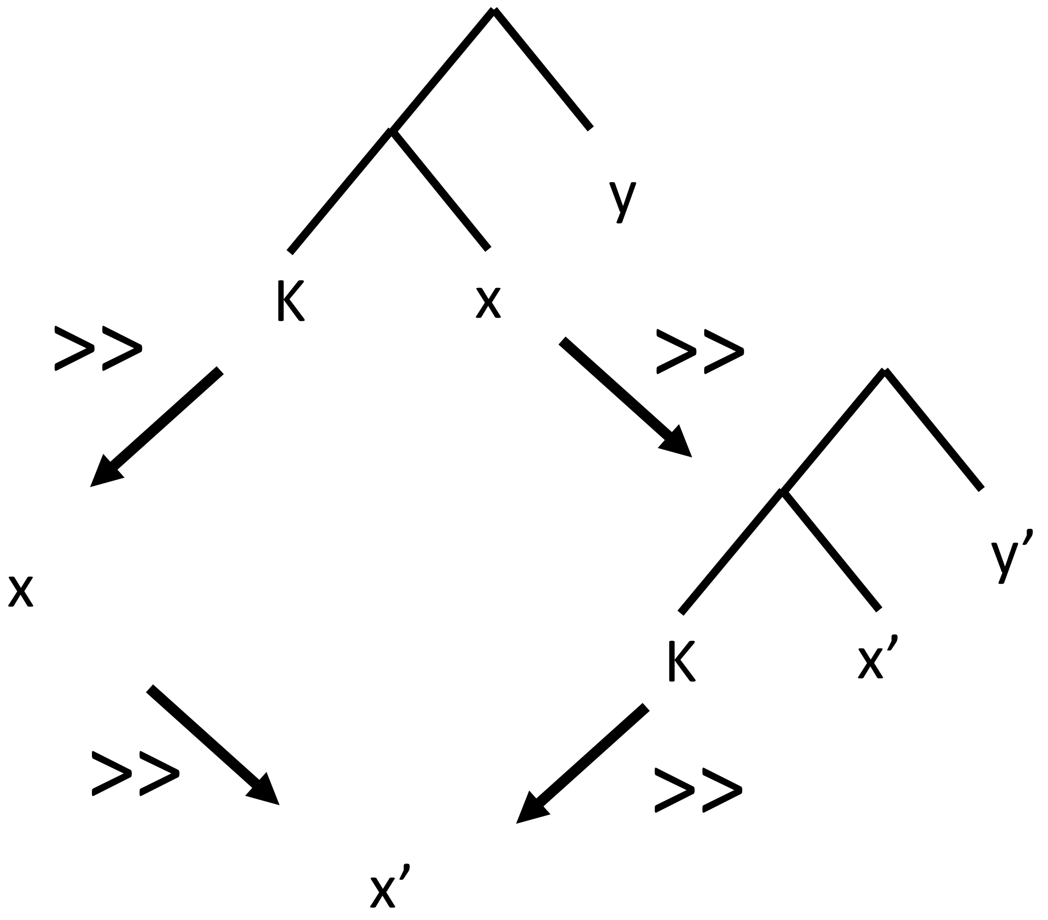
- Clearly $A >>^* B$ iff $A \rightarrow^* B$

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

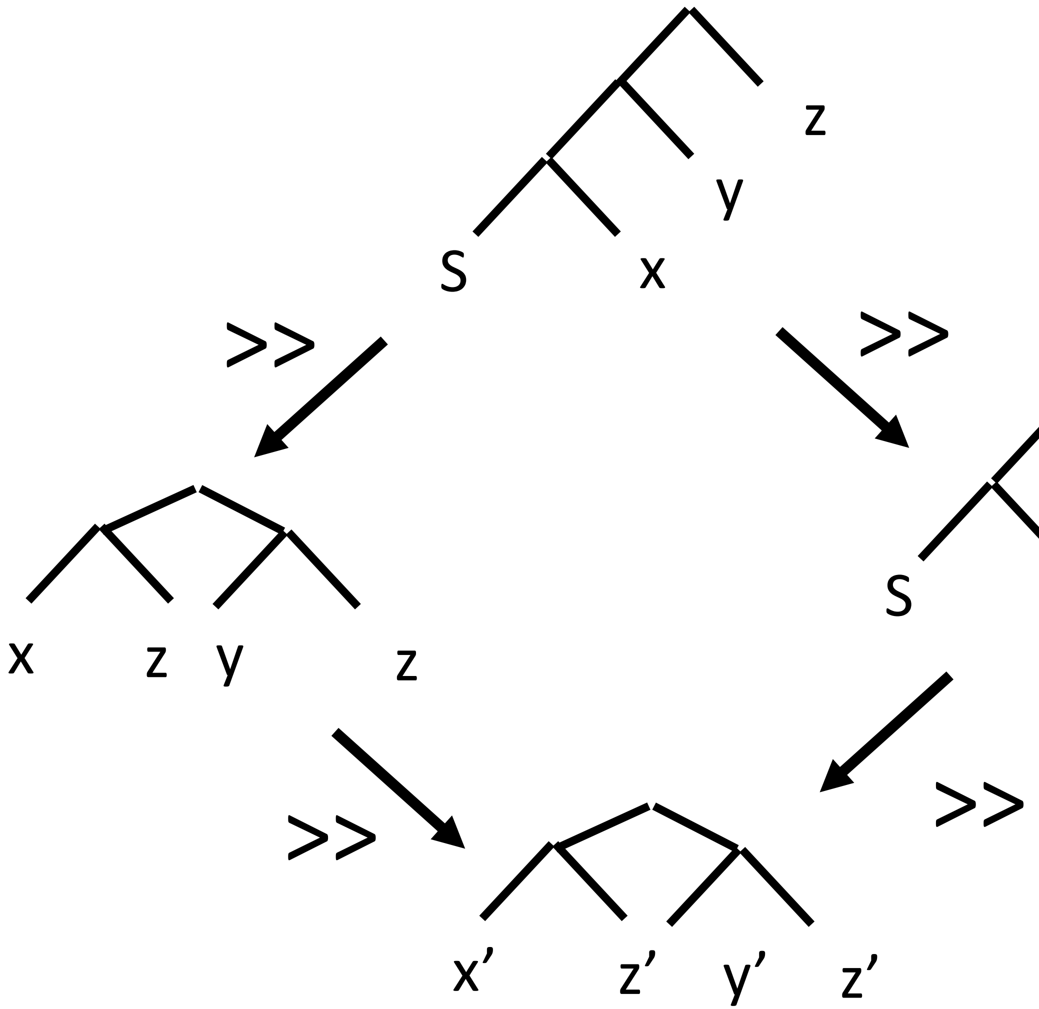
Case I x



Case K x y



Case S x y z



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: $S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$
- 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 ...

NumPy
Array Programming with Combinators

Overview

- In practice, combinator programming is used most with collections
  - And particularly arrays
- Benefits
  - Conciseness: Bulk operations over the entire collection
    - Iteration/recursion is "baked in" to the operations
  - Performance: Leave the details of the implementation the underlying system
    - Might be very different for different hardware, e.g., CPUs or GPUs
- The most popular of these interfaces today is NumPy
  - But note, python has imperative features
  - So programs tend to be a mix of styles, including using variables, state, etc.

A Brief NumPy Tutorial

A short overview of NumPy arrays

- Defining
- Shape
- Views
- Filters

Using NumPy

```
# This line will always appear in a NumPy program
import numpy as np
```

Defining an Array

```
import numpy as np

# initialize an array A of 10 elements with the integers 0..9
A = np.arange(0,10)
```

Example: Adding Arrays

```
import numpy as np
A = np.arange(0,10)

# addition is pointwise if the dimensions match
np.add(A,A)
```

Reshaping

```
import numpy as np
A = np.arange(0,10)

# Reshaping is a general operation that changes array dimensions.
# Normally defines a view: creates a new way of naming the array but does
# not make a copy.

# view the elements of A as a 2x5 array
A.reshape(2,5)

# view the elements of A as a 10x1 (column) array
A.reshape(10,1)
```

Example: Outer Product

```
import numpy as np
A = np.arange(0,10)

# We can use a combination of reshape and broadcast to define a
# concise outer product.

np.multiply(A,A.reshape(10,1))
```

Slicing

```
import numpy as np
A = np.arange(0,10)

# slicing defines views of subsets of an array
A[3:]     # slice of 4th element to the end of the array
A[:-3]    # slice up to the 4th element from the end of the array
A[1:-1]  # slice of all but the first and last elements of the array
A.reshape(2,5)[:,1:3]     # slicing in multiple dimensions
A.reshape(2,5)[0:2,1:3] # same slice written a different way
```

Example: Moving Average

```
import numpy as np
A = np.arange(0,10)

# cumulative sum is one of many NumPy built-in array functions
B = np.cumsum(A)

# moving average of A with a window of size 3
(B[3:] – B[:-3] )  / 3.0
```

Masks

```
import numpy as np
A = np.arange(0,10)

# Using an array in a predicate returns an array of Boolean results
# Here broadcasting promotes 5 to a 1D array of 5's
A > 5
A <= 5
(2 * A) == (A ** 2)
```

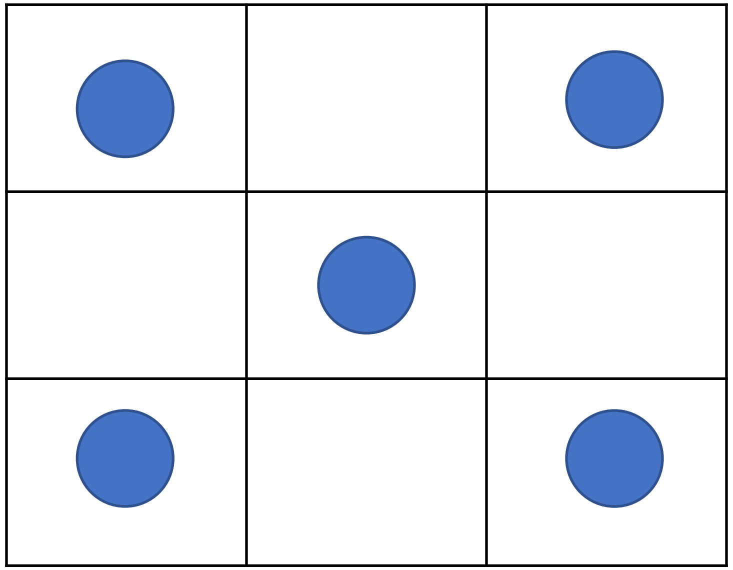
Filters

```
import numpy as np
A = np.arange(0,10)

# Boolean arrays can be used as array indices to filter arrays
A[A > 5]                 # elements of A that are > 5
A[A <= 5]                # elements of A that are <= 5
A[(2 * A) == (A ** 2)]   # elements x of A where 2*x == x ** 2
```

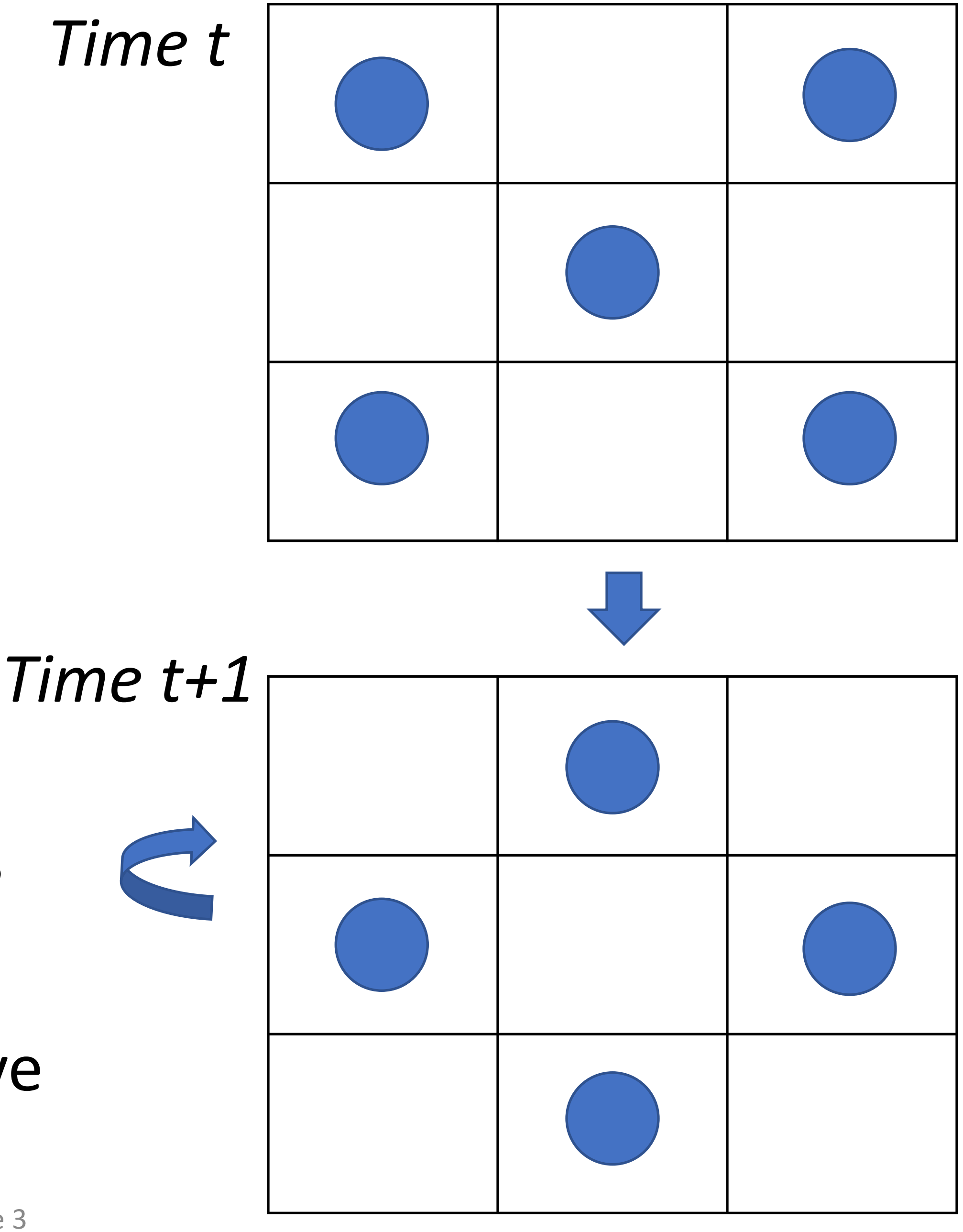
A Bigger Example: The Game of Life

- The Game of Life is played on 2D grid in time steps
- Grid cells are either *live* or *dead*
- A cell is live or dead at time *t+1* based on its neighbors at time *t*
  - Cells at the world's edge are always dead
- Defined by George Conway in 1969
  - An early example of cellular automata



Rules

- A live cell with < 2 neighbors dies
  - From loneliness
- A live cell with > 3 neighbors dies
  - From overcrowding
- A live cell with 2 or 3 neighbors survives
- A dead cell with 3 neighbors becomes live



The Game of Life

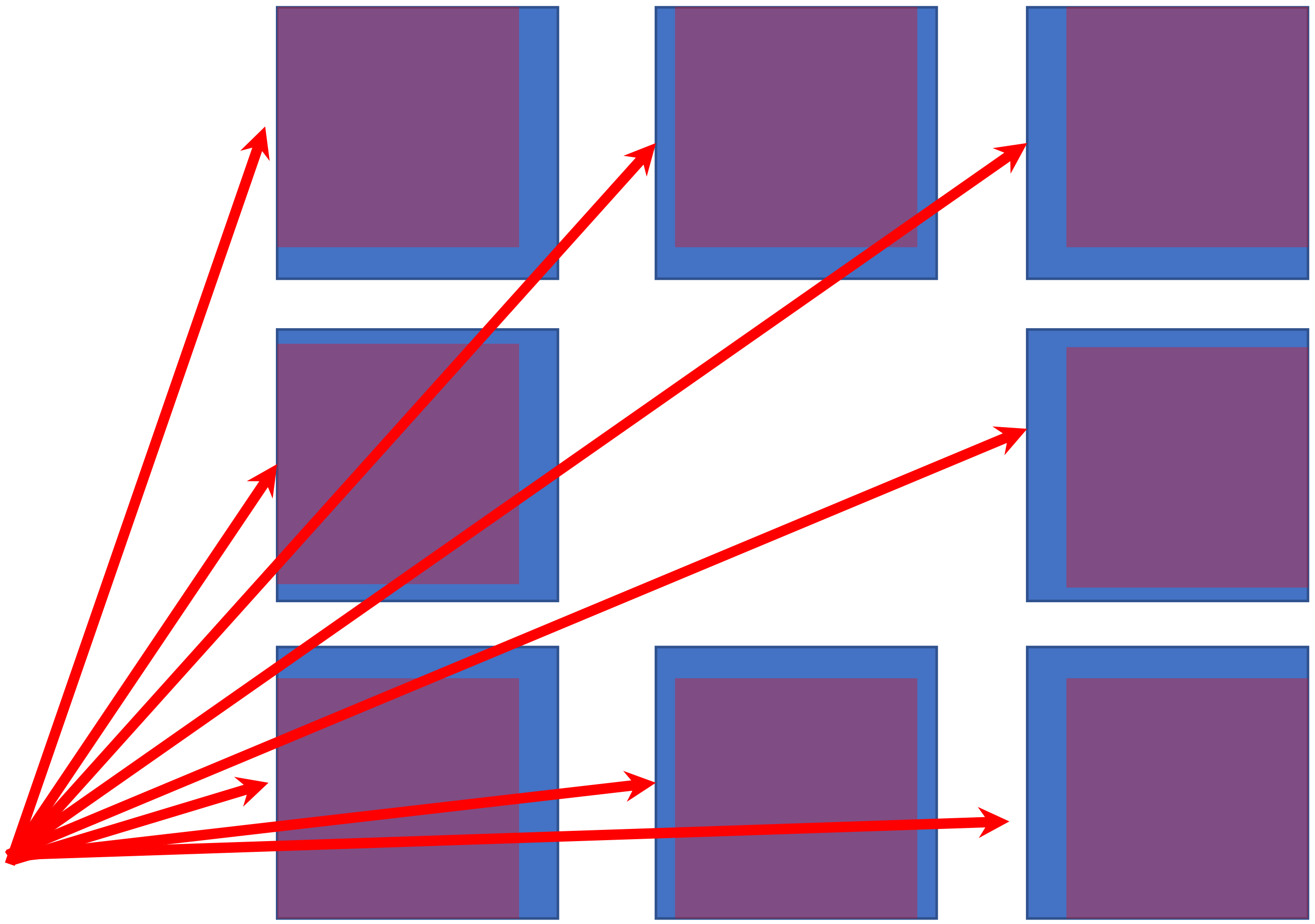
```
import numpy as np
Z = np.zeros((300, 600))
Z[1:-1,1:-1] = np.random.randint(0,2,np.shape(Z[1:-1,1:-1]))       # 0 is dead, 1 is live

while True:
    N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
           Z[1:-1, 0:-2]                          + Z[1:-1, 2:] +
           Z[2:   , 0:-2]  + Z[2:   , 1:-1]  + Z[2:   , 2:])
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
    Z[:,:] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
```

Picture

```
N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
     Z[1:-1, 0:-2]                  + Z[1:-1, 2:] +
     Z[2:  , 0:-2]  + Z[2:  , 1:-1]  + Z[2:  , 2:])
```

Summing these 8 subarrays computes the number of live neighbors for each cell in the interior of the space.



Explanation

…

```
# N is a 2D array of the number of neighbors of each cell
# birth is a 2D Boolean array; a cell is true if it is has 3 neighbors and is dead
birth = (N == 3) & (Z[1:-1, 1:-1] == 0)

# survive is a 2D Boolean array; a cell is true if it is has 2 or 3 neighbors and is live
survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)

# create a new generation
# the interior cells of Z are live if they are born or survive the previous time step
Z[:,:] = 0
Z[1:-1, 1:-1][birth | survive] = 1
```

The Game of Life

```
import numpy as np
Z = np.zeros((300, 600))
Z[1:-1,1:-1] = np.random.randint(0,2,np.shape(Z[1:-1,1:-1]))       # 0 is dead, 1 is live

while True:
    N = (Z[0:-2, 0:-2] + Z[0:-2, 1:-1] + Z[0:-2, 2:] +
         Z[1:-1, 0:-2]                 + Z[1:-1, 2:] +
         Z[2:   , 0:-2]  + Z[2:   , 1:-1]  + Z[2:   , 2:])
    birth = (N == 3) & (Z[1:-1, 1:-1] == 0)
    survive = ((N == 2) | (N == 3)) & (Z[1:-1, 1:-1] == 1)
    Z[:,:] = 0
    Z[1:-1, 1:-1][birth | survive] = 1
```

History

- SKI calculus was developed by Schoenfinkel in the 1920's
  - One of Hilbert's students
- Rediscovered by 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
  - Basic data type is the multidimensional array
- Trivia: Special APL keyboards accommodated the many 1 character combinators
  - APL programs can be unreadable strings of Greek letters
- Highly influential
  - On functional programming (several languages)
  - And array programming (Matlab, R, NumPy)

```
{(+/⍵)÷≢⍵}
```

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