Combinator Calculus

CS242

Lecture 2

```
fun innerproduct(a, b, n):
  c := 0
  for i := 1 step 1 until n do
    c := c + a[i] ∗ b[i]
  return c
```

- Statements operate on invisible state
- Computes word-at-a-time by repetition of assignment/modification
- Requires names for arguments, iterator, return value

```
let innerproduct = zip |> (map ∗) |> (reduce +)
```

- Built from composable functions (map, reduce, pipe)
- Operates on whole conceptual units (lists), no repeated steps
- No names for arguments or temporaries

Combinator Calculus

- Calculus:

  "A method of computation or calculation in a special notation"

- Combinator:

  "A primitive function without free variables"

Overview

- A variable-free programming language using only functions
- A simple Turing-complete computational formalism
- A starting point for more involved languages
- And something different!

SKI Calculus

$$I\ x \rightarrow x$$

$$K\ x\ y \rightarrow x$$

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

Definition

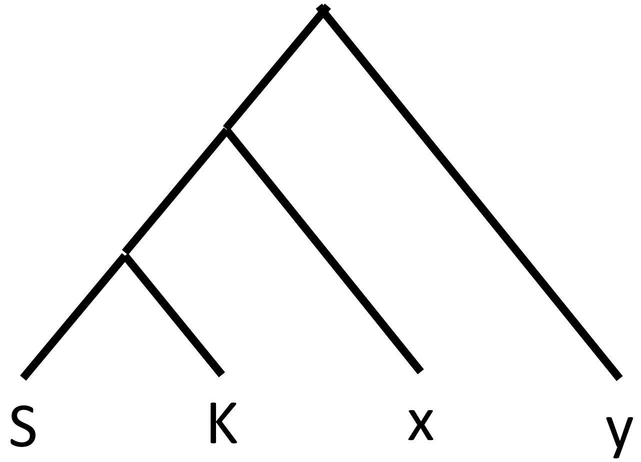
- The *terms* of the SKI calculus are the smallest set such that
  - S, K, and I are terms
  - If x and y are terms, then x y is a term

- Terms are trees, not strings
  - Parentheses show association where necessary
  - In the absence of parentheses, association is to the left
    - i.e., S x y z = (((S x) y) z)

Example

S K x y

(((S K) x) y)



Context Free Grammar

Expr → S

Expr → K

Expr → I

Expr → Expr Expr

Expr → S | K | I | Expr Expr

Rewrite Rules

- The three rules of the SKI calculus are an example of a *rewrite system*
  - Any expression (or subexpression) that matches the left-hand side of a rule can be replaced by the right-hand side

- The symbol $\rightarrow$ stands for a single rewrite
- The symbol $\rightarrow^*$ stands for the reflexive, transitive closure of $\rightarrow$

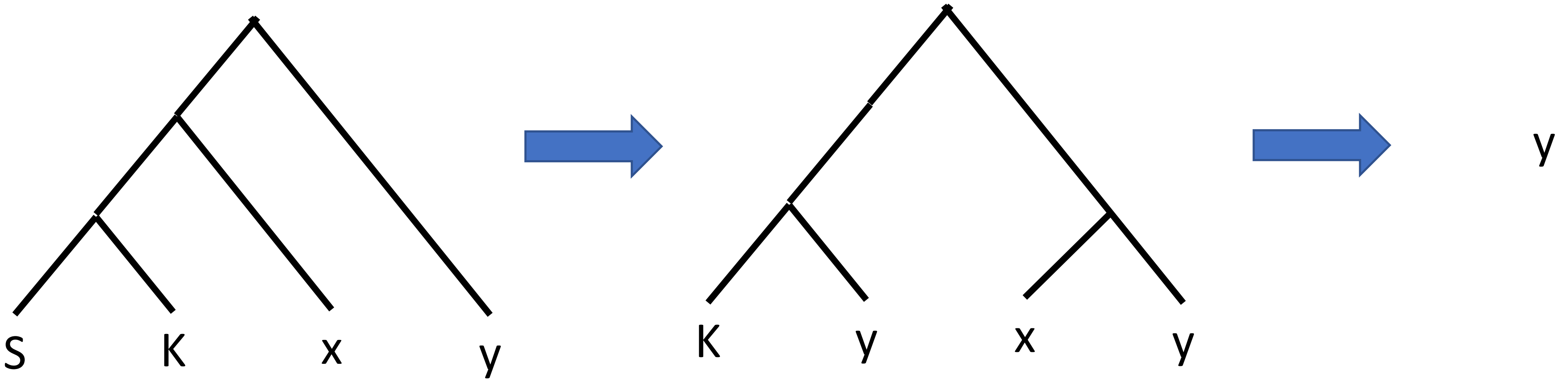
$$I\ x \rightarrow x$$
$$K\ x\ y \rightarrow x$$
$$S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$$

Example

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

Example

S K x y → (K y) (x y) → y



Another Example

$$S\ I\ I\ x \rightarrow (I\ x)\ (I\ x) \rightarrow x\ (I\ x) \rightarrow x\ x$$

So ...

$$(S\ I\ I)\ (S\ I\ I) \rightarrow (I\ (S\ I\ I))\ (I\ (S\ I\ I)) \rightarrow (S\ I\ I)\ (I\ (S\ I\ I)) \rightarrow (S\ I\ I)\ (S\ I\ I)$$

What a Strange Language!

- A language of functions
  - Functions are all there is to work with

- Minimalist
  - Typical of languages designed for study
  - Clears away the complexity of ``real'' languages
  - Allows for very direct illustration of key ideas

Programming

- Recursion

- Conditionals

- Data structures

Recursion

- (S I I) (S I I) is a non-terminating expression
  - Can always be rewritten, since it rewrites to itself
  - A form of looping

- Proper recursion is just a little more involved

$$x = S\ (K\ f)\ (S\ I\ I)$$

Then $S\ I\ I\ x \rightarrow^* x\ x = S\ (K\ f)\ (S\ I\ I)\ x \rightarrow^* f\ (x\ x) \rightarrow^* f\ (f\ (x\ x)) \rightarrow^* \ldots$

Conditionals

- To have branching behavior, we need Booleans.

- We use an *encoding.*
  - We choose combinators to represent true, false, or, and, etc.
  - We need to check that our choices are consistent with the expected behavior of Boolean tests and branching

Booleans

- Represent true (false) by a function that given two arguments picks the first (second)
- Combines Boolean with if-then-else

- True $\quad T\ x\ y \rightarrow x$
- False $\quad F\ x\ y \rightarrow y$

- $T = K$
- $F = S\ K$

Boolean Operations

- Let B be a Boolean (T or F)

- Not(B) = B F T

Boolean Operations

- Let B be a Boolean (T or F)

- B1 OR B2 = B1 T B2

Boolean Operations

- Let B be a Boolean (T or F)

- B1 AND B2 = B1 B2 F = B1 B2 (S K)

Example

(NOT F) AND T = (F F T) T F

Abstraction

- Our encodings of Booleans and Boolean operators are *abstractions*
- We require that the Boolean operators produce valid representations of Booleans if they are are applied to Boolean inputs
  - But nothing explicitly enforces this property
  - It is up to use to choose an encoding so that this property holds

Integers

N applies its first argument *N* times to its second argument

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

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

$\mathit{Succ}\ n\ f\ x = f\ (n\ f\ x)$ $\mathit{Succ} = S\ (S\ (K\ S)\ K)$

$S\ (S\ (K\ S)\ K)\ \ n\ f\ x \rightarrow\ (S\ (K\ S)\ K\ f)\ (n\ f)\ x \rightarrow ((K\ S)\ f)\ (K\ f)\ (n\ f)\ x \rightarrow$
$S\ (K\ f)\ (n\ f)\ x \rightarrow ((K\ f)\ x)\ ((n\ f)\ x) \rightarrow f\ ((n\ f)\ x) = f\ (n\ f\ x)$

Order of Evaluation

- More than one rule may apply to an expression
- 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 any program transformation (e.g., for optimization) is based on changing the order in which computation happens

Order of Evaluation

- What is a good reduction strategy?
  - There are multiple sensible choices

- In general, applying rules as close to the root as possible will terminate if any reduction strategy terminates
  - By avoiding work on expressions that are thrown away (e.g., by K)
  - And reducing the potential for parallelism

Confluence

- But could different choices of evaluation order change the result of the program?

- The answer is no!

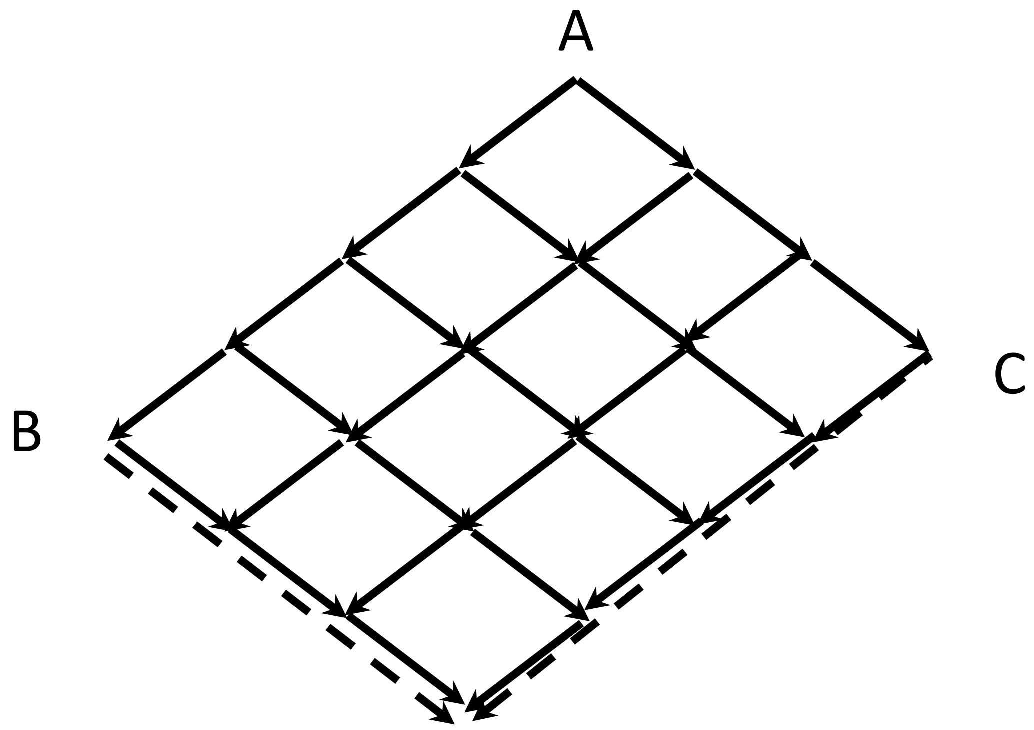
- A set of rewrite rules is *confluent* if for any expression E_0, if $E_0 \rightarrow^* E_1$ and $E_0 \rightarrow^* E_2$, then there exists E_3 such that $E_1 \rightarrow^* E_3$ and $E_2 \rightarrow^* E_3$.

Proving Confluence

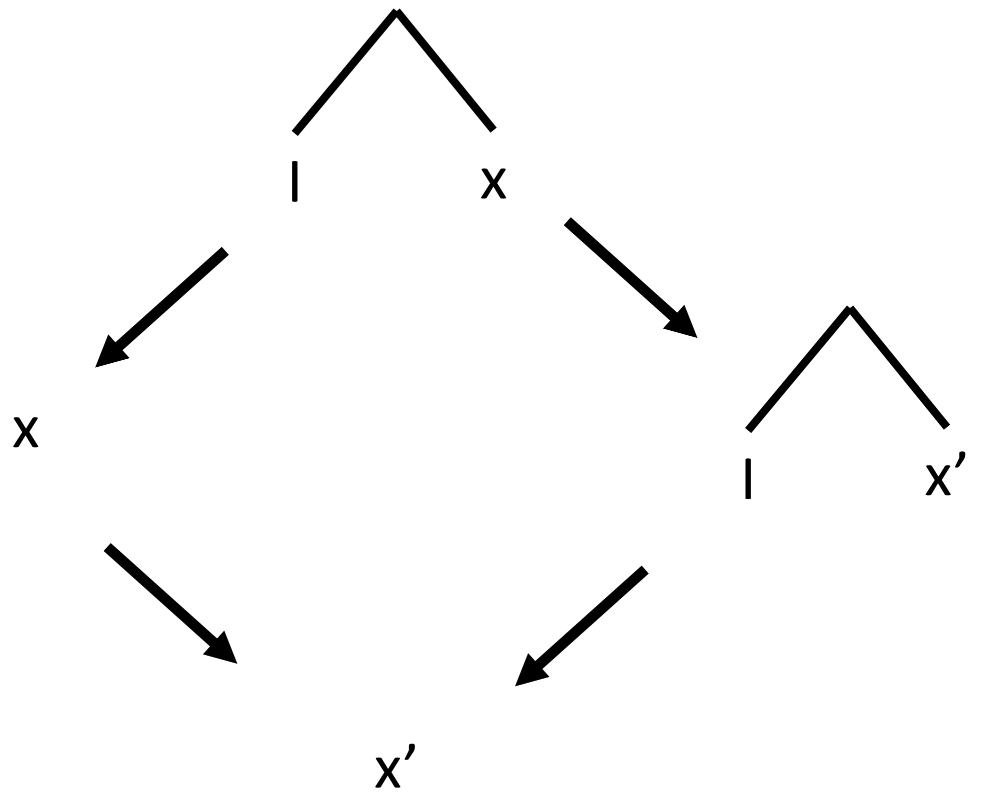
Thm: If for all A, $A \rightarrow B$ & $A \rightarrow C$ implies there exists a D such that $B \rightarrow D$ and $C \rightarrow D$ (in one step), then $\rightarrow$ is confluent.

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

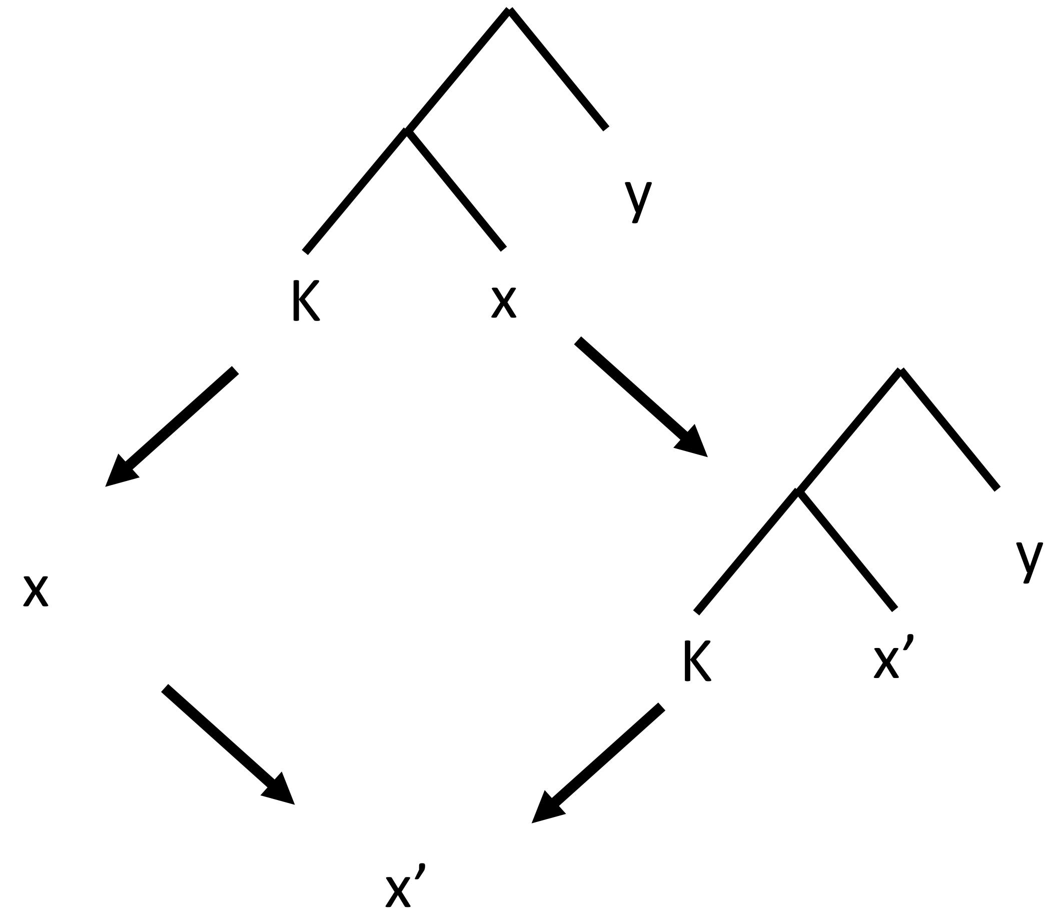
Diagram



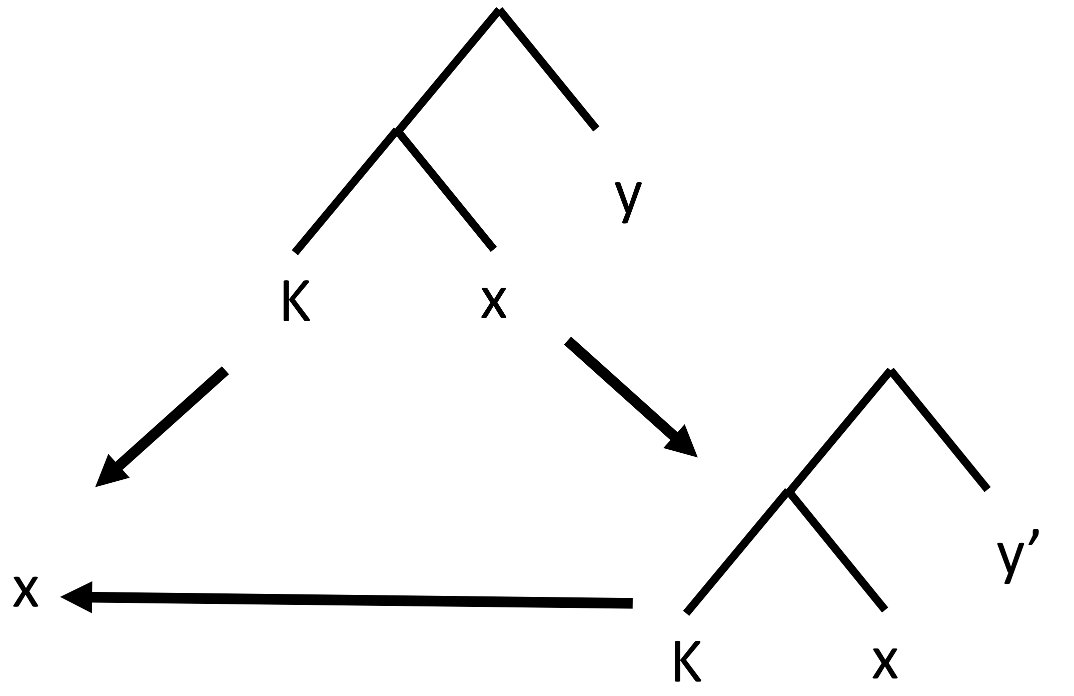
Confluence of SKI: Case I x



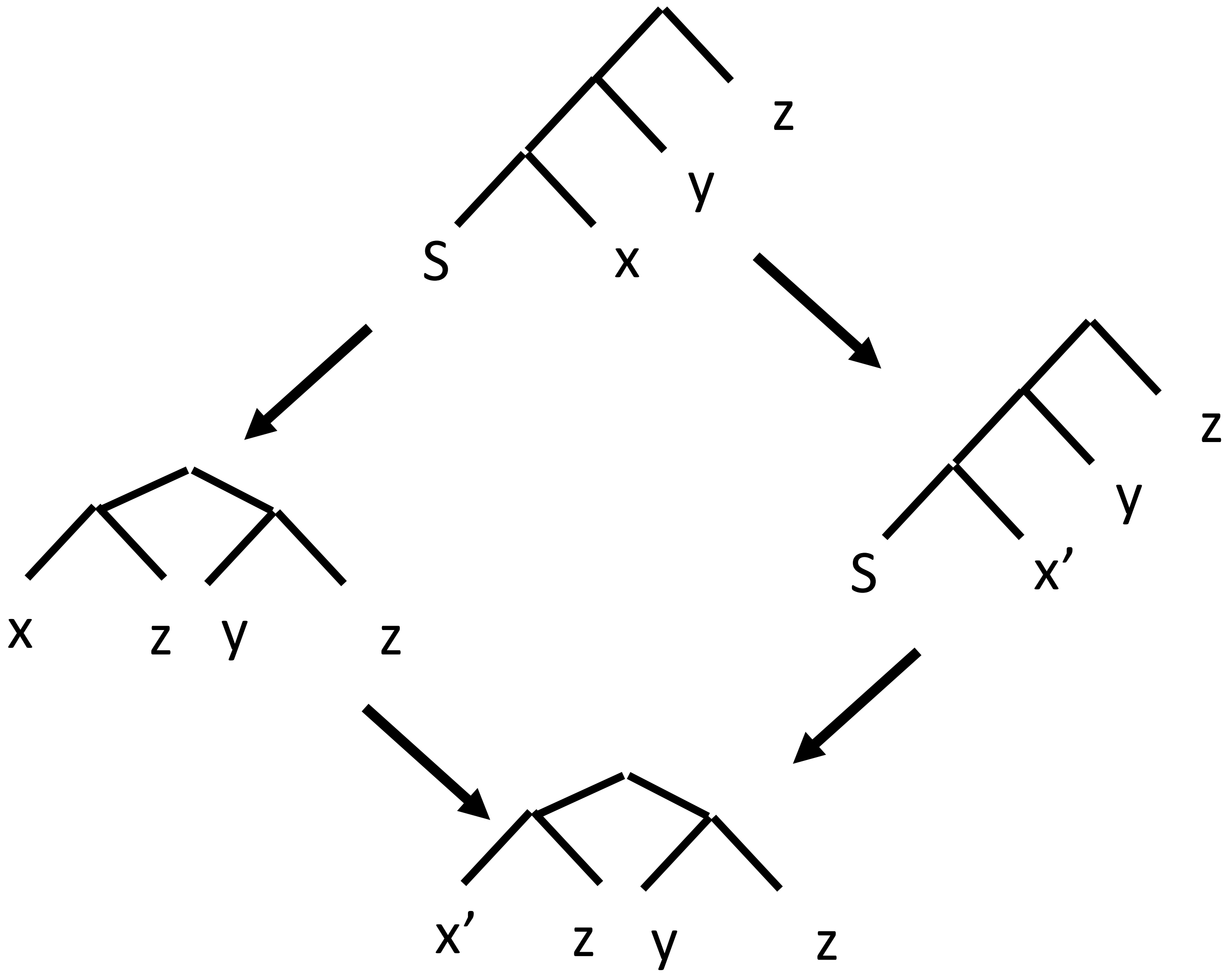
Case K x y   (1 of 2)



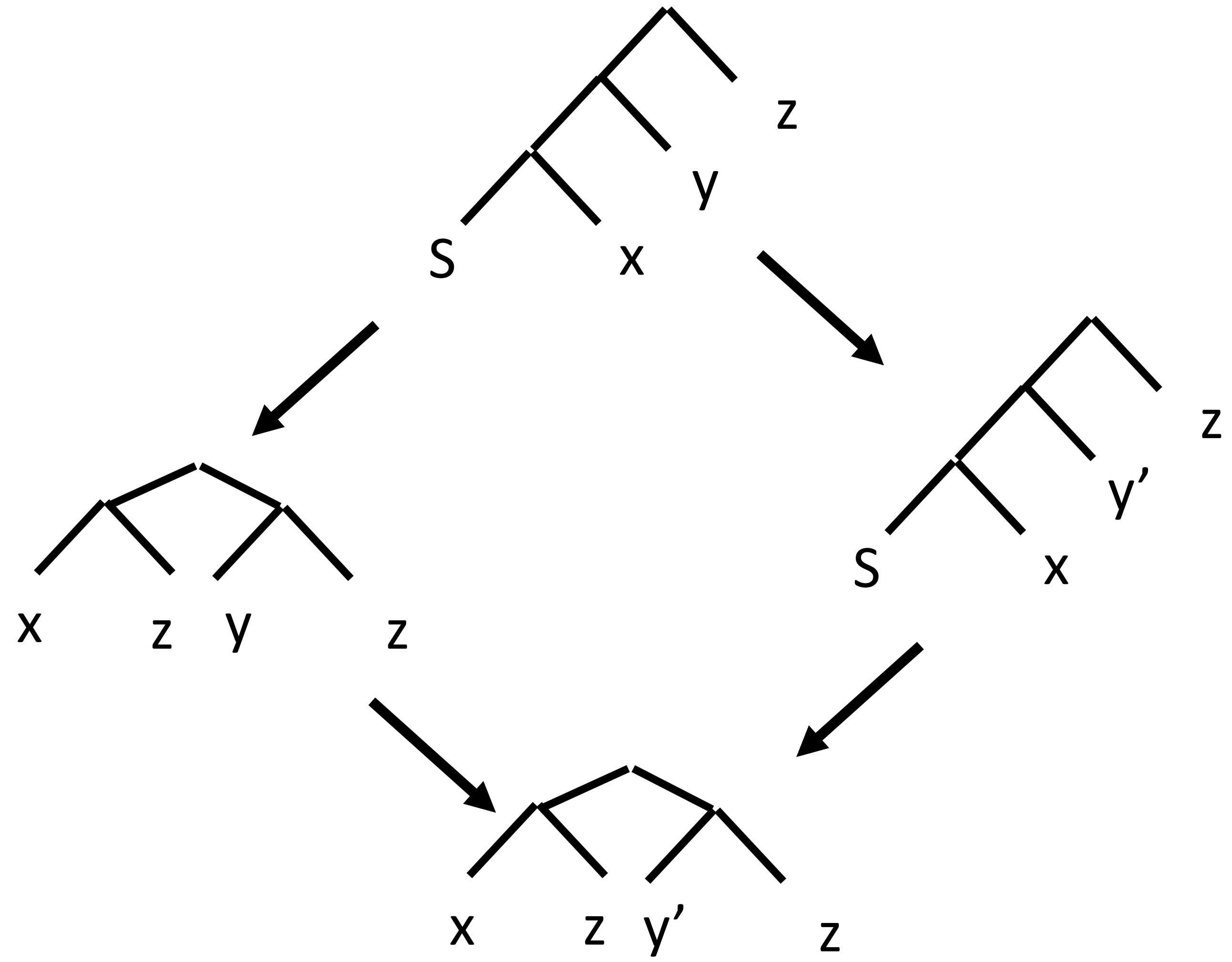
Case K x y   (2 of 2)



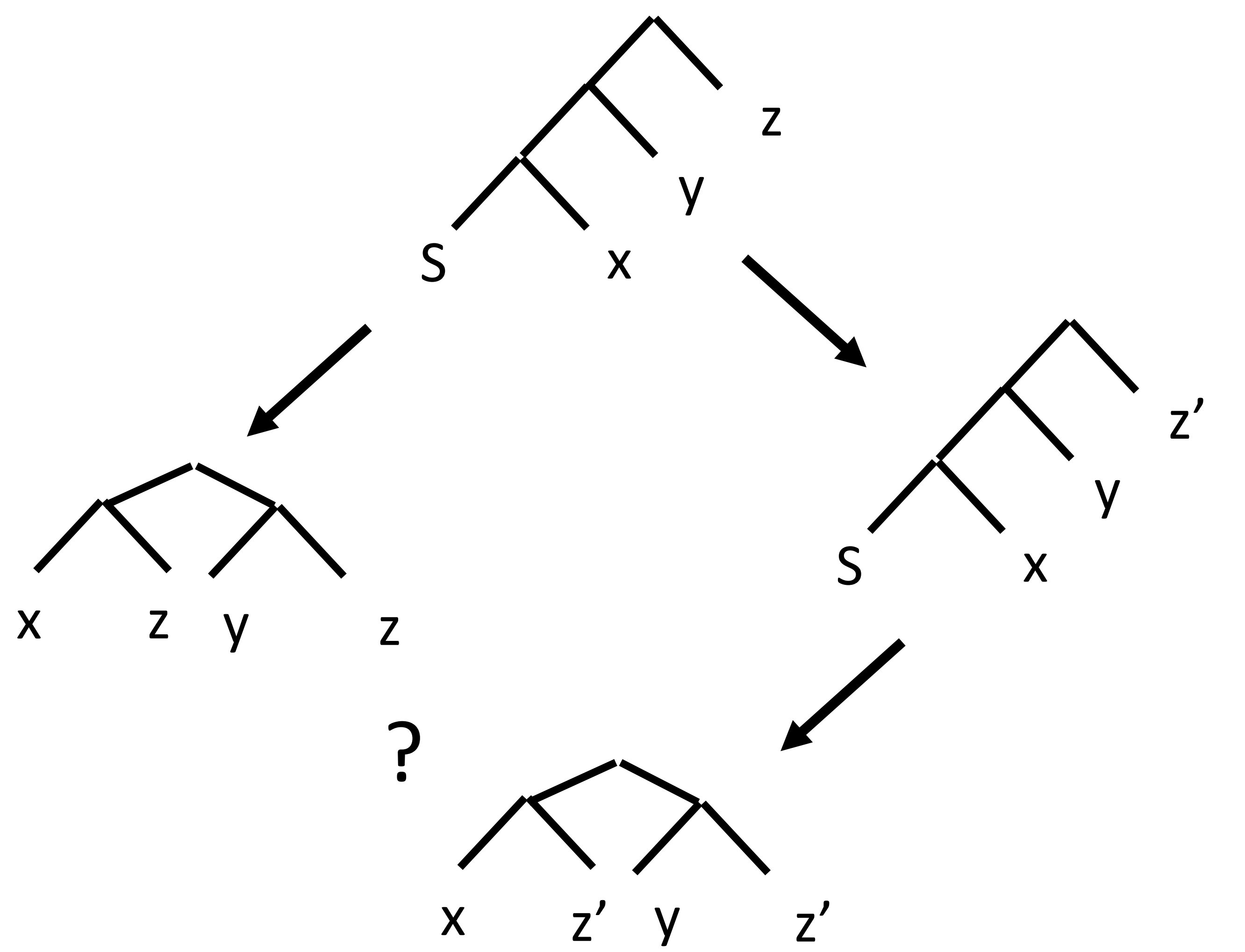
Case S x y z   (1 of 3)



Case S x y z   (2 of 3)



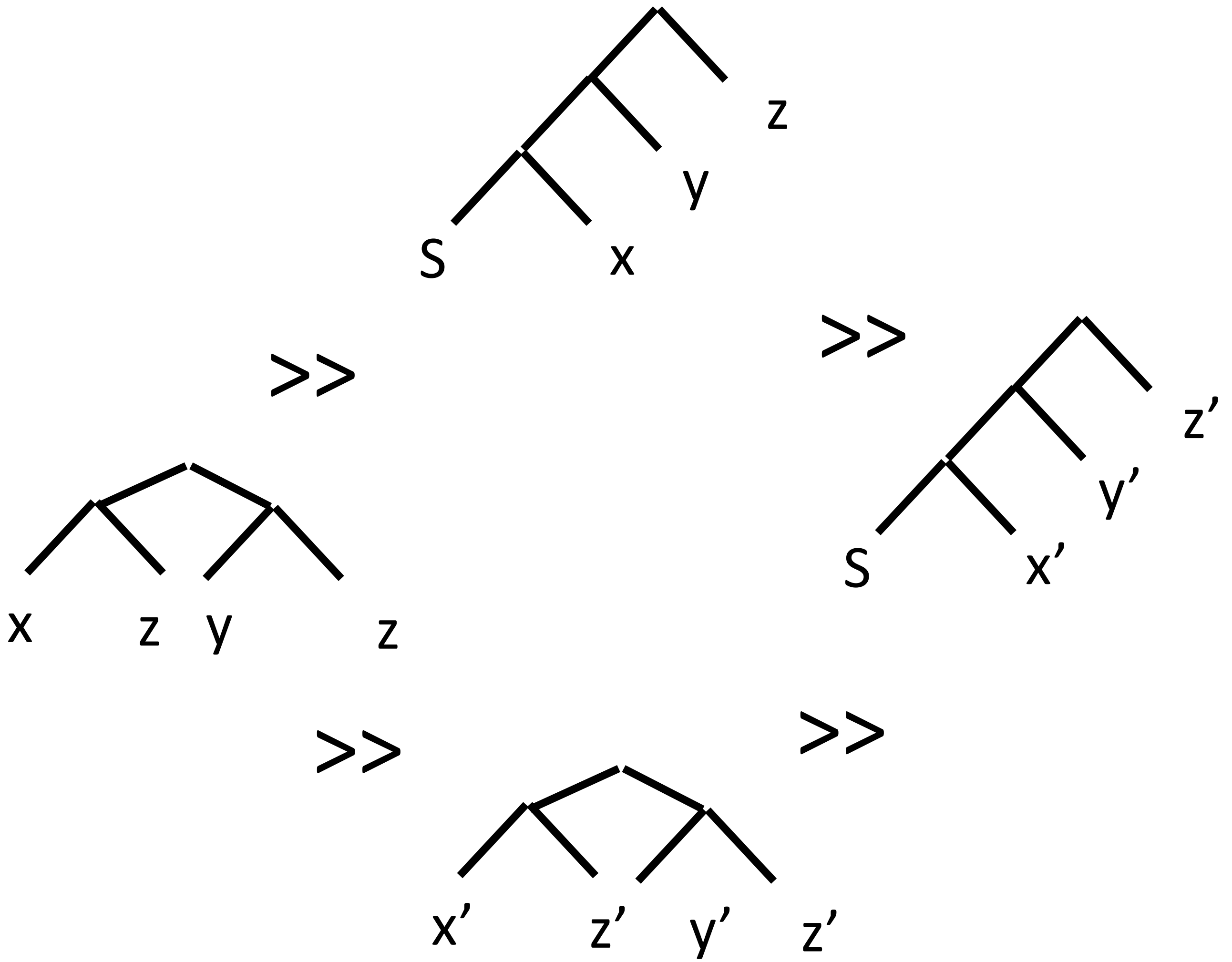
Case S x y z   (3 of 3)



A New Relation

- Define X >> Y if
  - X $\rightarrow$ Y via a rewrite at the root node
  - X = A B and A >> A' and B >> B'
- Clearly A $>>^*$ B iff A $\rightarrow^*$ B
- Thm: >> has the one step diamond property.

Case S x y z   (3 of 3)



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?
  - Combinator 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

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 ...

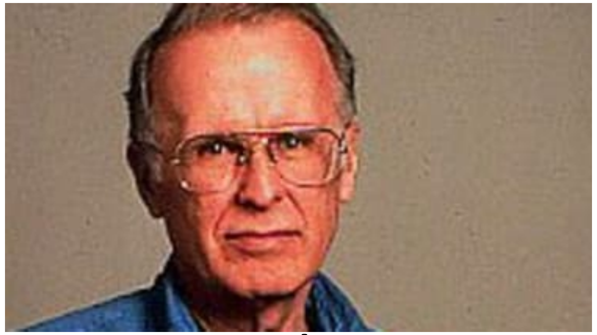
History

- 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), successors to APL, MatLab, Map-Reduce

FP



- John Backus's Turing Award lecture brought new attention to combinator languages
- Backus developed FORTRAN, the first successful high-level language
- But late in his career he advocated for functional programming
  - Encouraging thinking at the function level
  - And getting away from a word-at-a-time model of computation
- Proposed FP, a combinator-based functional language
  - And emphasized an "algebra of programs"

FP's Algebra

- FP's basic data structure is the vector (or list)
- map f [x, y, z] = [f x, f y, f z]
- Combinators enable simple and powerful program transformations:

$$(\text{map } f) \circ (\text{map } g) = \text{map } (f \circ g)$$

- Think about how you could describe this transformation in C!

Summary

- Combinator calculi are among the simplest formal computation systems
- Also important in practice for large scale distributed/parallel programming
  - Where thinking in terms of bulk operations is beneficial
- 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