Combinator Calculus

CS242

Lecture 2

Combinator:

A function without free variables

Calculus:

A method of computation or calculation in a special notation

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

A function call is written by juxtaposing two expressions

The function

The argument

$I\ x \rightarrow x$

The result

The arrow indicates a step of computation

Identity function

$K\ x\ y \rightarrow x$

Constant functions

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

Generalized function application

Multiple Arguments

K — A function by itself is a well-formed program. No rules apply.

K x — No rules apply to K with one argument

K x y → x — K only ``executes" when it has two arguments

K x y z → x z — K only uses the first two arguments

What is S?

Creates a function application.

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

Duplicates z

For a general functional language:

Need a way to program function calls (applications).

Need to reuse values (make copies).

S combines both.

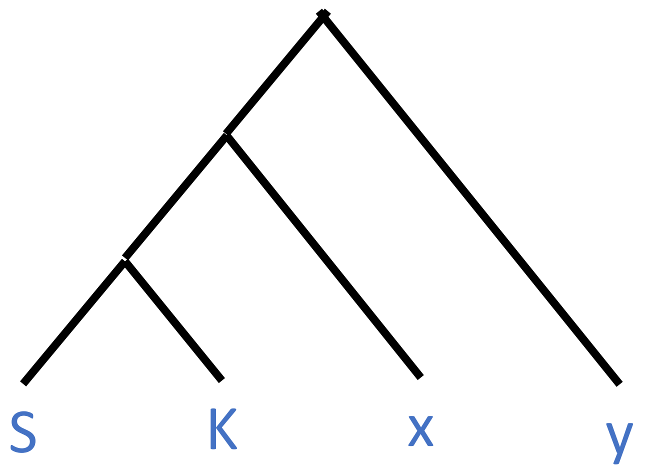
Definition

- 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

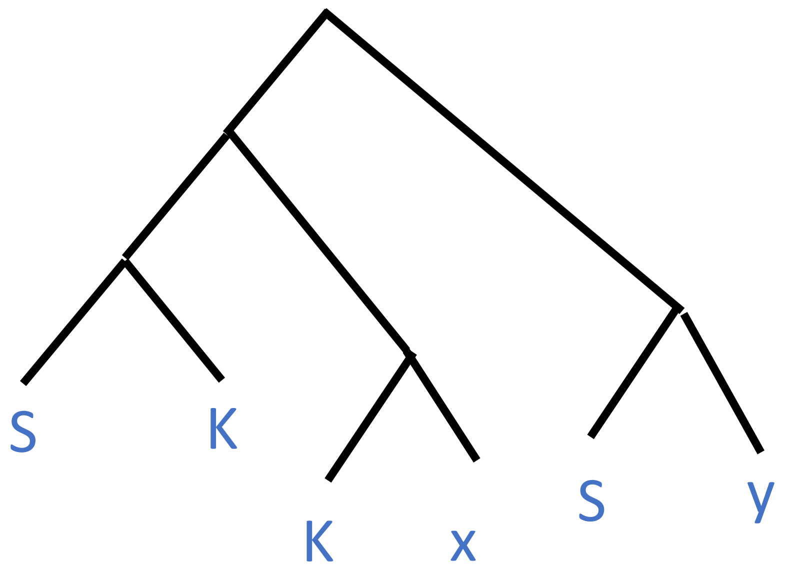
S K x y

(((S K) x) y)



Example

(((S K) (K x)) (S y))



Context Free Grammar

Expr → S

Expr → K

Expr → I

Expr → Expr Expr

Expr → (Expr)

Expr → S | K | I | Expr 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.e., zero or more rewrites

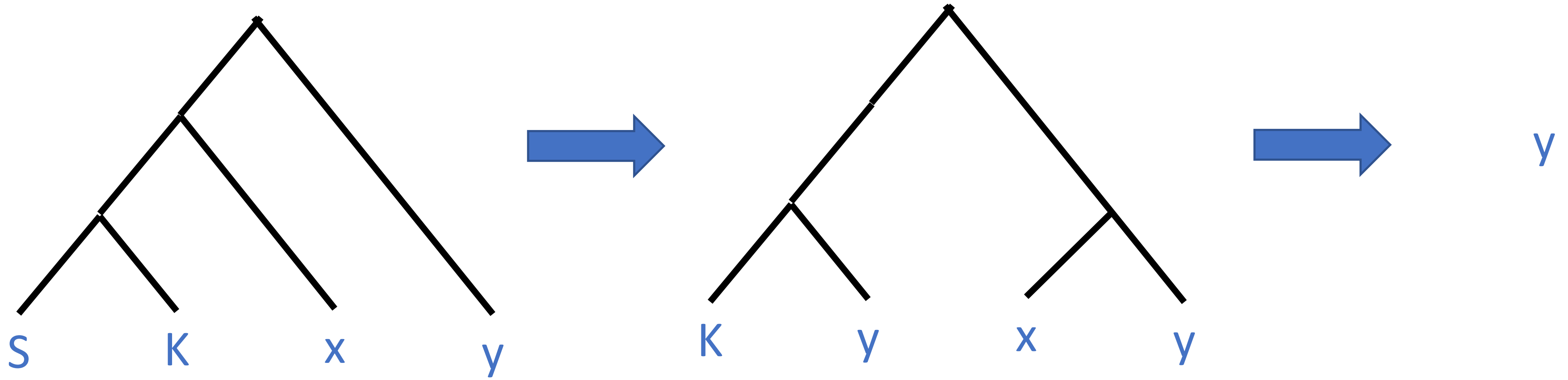
$$I\ x \rightarrow x$$
$$K\ x\ y \rightarrow x$$
$$S\ x\ y\ z \rightarrow (x\ z)\ (y\ z)$$

Example

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

Example

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



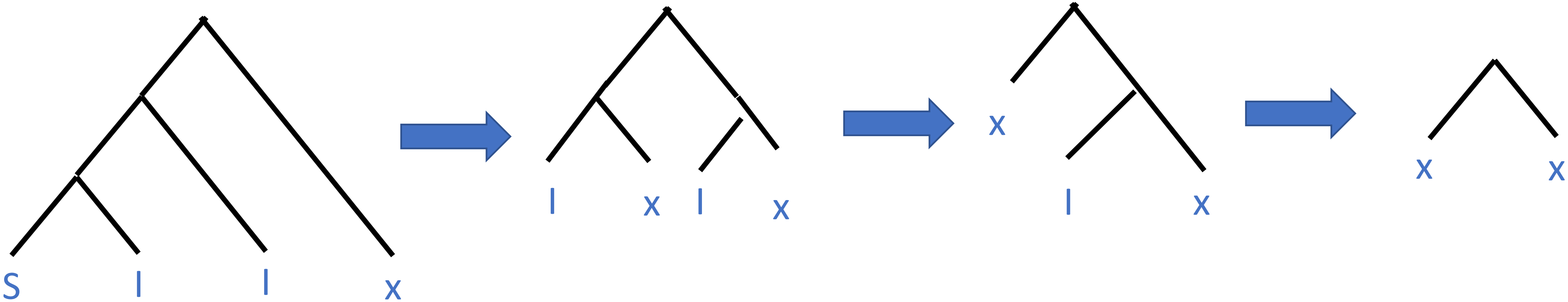
What Do These Do?

- K x → ?
- S x y → ?

- Answer: Nothing!
  - No rewrite rules apply until the combinator has all its arguments
  - K x is a *partially applied* function
  - A partially applied function is a function, and can be passed around, copied, etc.

Another Example

S I I x → (I x) (I x) → x (I x) → x x

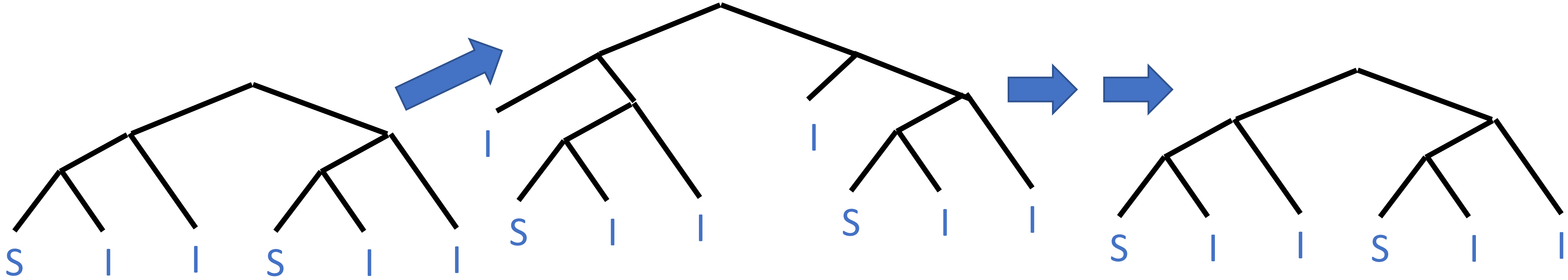


And Another Example

S I I x → (I x) (I x) → x (I x) → x x

So …

(S I I) (S I I) → (I (S I I)) (I (S I I)) → (S I I) (I (S I I)) → (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

General 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

- Recursive function calls are just a little more involved

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

So $S\ I\ I\ x \rightarrow^{*} x\ x = S\ (K\ f)\ (S\ I\ I)\ x \rightarrow ((K\ f)\ x)\ ((S\ I\ I)\ x) \rightarrow^{*} f\ (x\ x) \rightarrow^{*} f\ (f\ (x\ x))$

- We will focus on a different form of looping later in the lecture

Conditionals

- To have branching behavior, we need Booleans.

- We use an *encoding.*
  - We choose combinators to represent true, false
  - And combinators not, or, and that have the correct behavior on those values

- An abstract data type
  - Except there is no type system to enforce the abstractions

Booleans

- Represent true by a function that picks the first of two arguments
- Represent false by a function that picks the second of two arguments

- 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

If-Then-Else

- Let B be a Boolean

- If B then X else Y   =   B X Y

Writing Combinators

- Let's say we want a combinator

swap x y = y x

- How do we write swap using S, K, and I?

swap = S (K (S I)) (S (K K) I)

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
  - 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(x,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

Back To Swap

- Recall swap x y = y x

- Arguments are abstracted starting with the last argument and progressing to the first argument
  - Because (swap x) y = y x
  - First abstract y in the definition of swap x, then abstract x from the definition of swap
  - We drop the red color for A, just remember it is not a combinator but mapping that produces a combinator from an expression with variables!

- First eliminate y in y x:

  swap x = A(y x, y) = S A(y,y) A(x,y) = S I A(x,y) = S I (K x)

- Now eliminate x from the result of the previous step:

  swap =
  A(S I (K x), x) =
  S A(S I, x) A(K x, x) =
  S (K (S I)) A(K x, x) =
  S (K (S I)) (S A(K,x) A(x,x)) =
  S (K (S I)) (S (K K) A(x,x)) =
  S (K (S I)) (S (K K) I)

Discussion

- Abstraction is a very simple, systematic algorithm

- But tedious
  - The resulting expressions can be huge and hard to read
  - Especially if the combinator takes multiple arguments

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

Back To Swap, Again …

- Recall swap x y = y x

- First eliminate y in y x:

  A(y x, y) = c2 A(y,y) x = c2 I x

- Now eliminate x from the result of the previous step:

  A(c2 I x, x) =
  A((c2 I) x, x) =
  c1 (c2 I) A(x, x) =
  c1 (c2 I) I

Defining c1

- c1 x y z = x (y z)

- Shortcut
  - Observe that c1 x y = S (K x) y
  - Then c1 x = S (K x)
  - Then c1 = A(S (K x), x) = S (K S) (S (K K) I)
  - Note S (K K) I = K
  - So c1 = S (K S) K

- Running the abstraction algorithm directly gives
  - c1 x y z = x (y z)
  - c1 x y = S (K x) (S (K y) I)
  - c1 x = S (K (S (K x))) (S (S (K S) (S (K K) I)) (K I))
  - c1 = S (S (K K) (S (K S) (S (K K) I))) (K (S (S (K S) (S (K K) I)) (K I)))

- The abstraction algorithm is not guaranteed to produce the smallest combinator!
  - But it is guaranteed to give one that is correct

Defining c2

- c2 x y z = (x z) y

- A((x z) y, z) = S (c1 x I) (K y)
- A(S (c1 x I) (K y), y) = S (K (S (c1 x I))) (c1 K I)
- A(S (K (S (c1 x I))) (c1 K I), x) =
  S ((c1 S (c1 K (c1 S (S (c1 c1 I) (K I)))))) (K (c1 K I))

Another Abstract Type: Pairs

Pairing must satisfy

pair x y first = x

pair x y second = y

Choose

first = T

second = F

Then

pair x y z = z x y

pair x y = c2 (c2 I x) y

pair x = c1 (c2 (c2 I x)) I

pair = c2 (c1 c1 (c1 c2 (c1 (c2 I) I))) I

A Brief Interlude

- SKI is an example of a language with *higher-order functions*
  - Functions can take functions as arguments and return functions as results
- Examples
  - swap x
  - and B
  - pair (and B)
  - S

- Many languages are *first order*
  - Functions can only work on data types that are not themselves functions

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

Replace arguments a and i by a pair:

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))

Now define functions:

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

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)

fac’ p = pair (m p) (i2 p)

Abstracting into combinators:

fac = c2 (c2 I fac’) (pair one one)

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)
```

Next Time …

- *Confluence*: A non-trivial property of the SKI calculus

- A brief survey of combinator languages