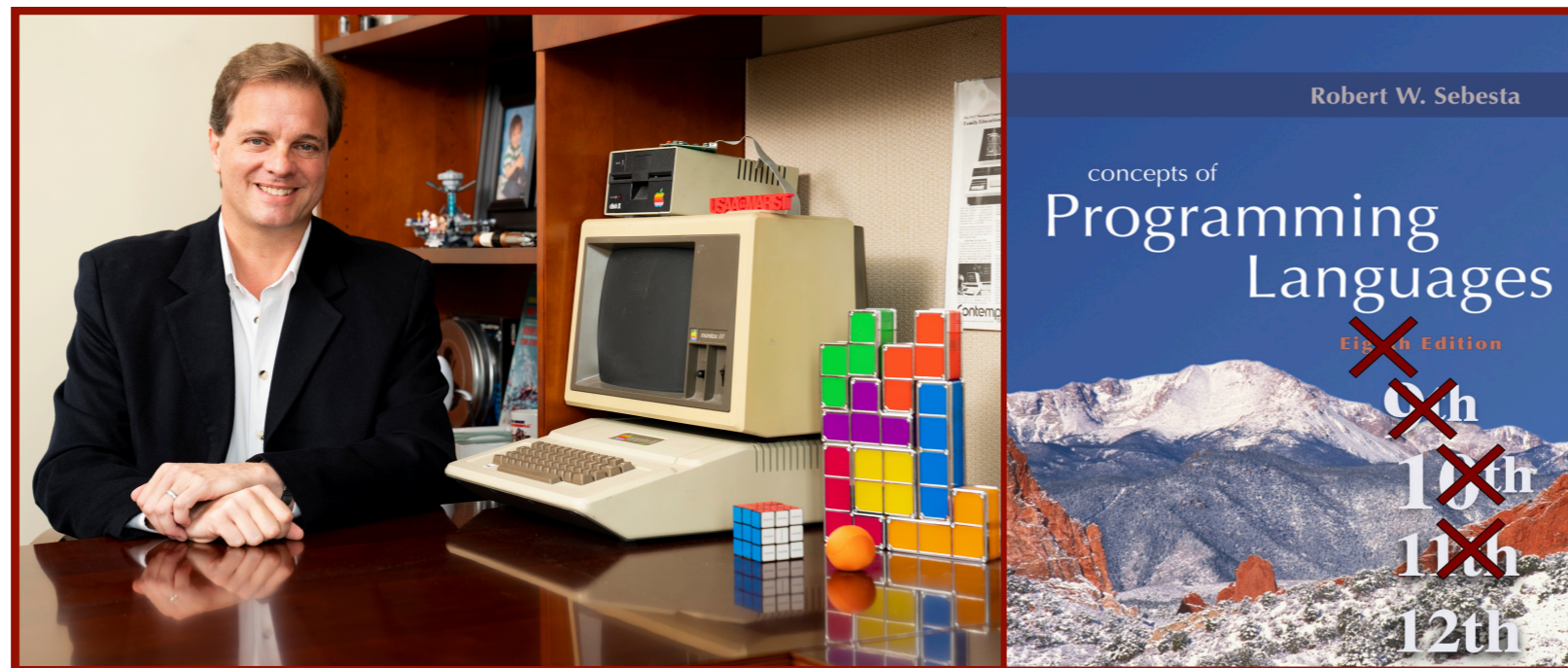

Lambda Calculus



Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Pure, untyped Lambda Calculus

History

1600s - Gottfried Leibniz

“Algebra of Thought”

1. Wanted to create a universal language in which all problems could be stated.
2. Wanted to find a decision method to solve all of those problems.

Set theory and first-order (predicate) logic will do for #1, at least for problems that can be expressed in mathematical terms.
(Is that all problems?)

#2 poses an interesting philosophical question:
Can one solve all problems formulated in a universal language?



Gottfried Wilhelm Leibniz
Wikipedia



Brian May
Queen

Lambda Calculus

History

1920s - Moses Schönfinkel — perhaps a student of David Hilbert — presents the “building blocks of logic”, a formal system of logic based on what we now call *combinator functions*.

- He showed that by appropriately combining them one could effectively define any function, or, in modern terms, that they could support **universal computation**.
- He also worked out what would eventually become known as “Currying”, even though its namesake, Haskell Curry, credited Schönfinkel in his work.
- See <https://writings.stephenwolfram.com/2020/12/where-did-combinators-come-from-hunting-the-story-of-moses-schonfinkel/> for his fascinating story.



Wikipedia

Moses Ilyich Schönfinkel



stephenwolfram.com

Lambda Calculus

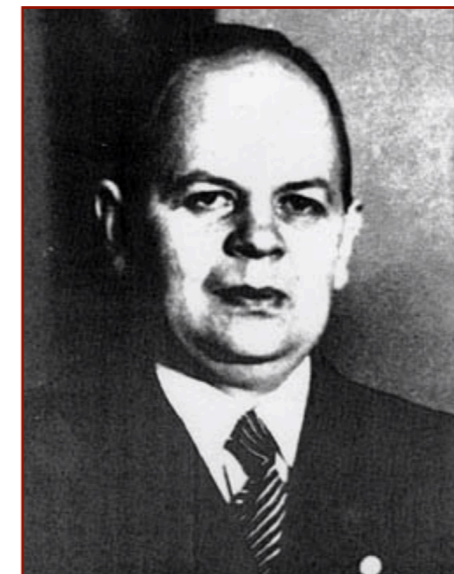
History

1928 - David Hilbert and Wilhelm Ackermann update Leibniz's philosophical question with their *Decision Problem*:

- Based on the idea of **universal computation**, they asked...
- Is there an **algorithm** that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement?
- Well... is there?



David Hilbert
Wikipedia



Wilhelm Ackermann
Wikipedia

Lambda Calculus

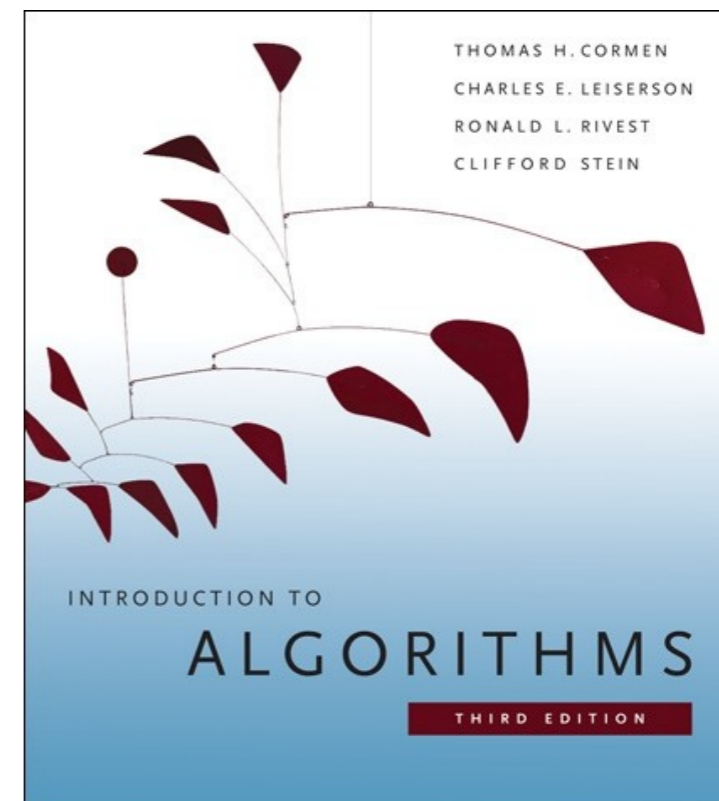
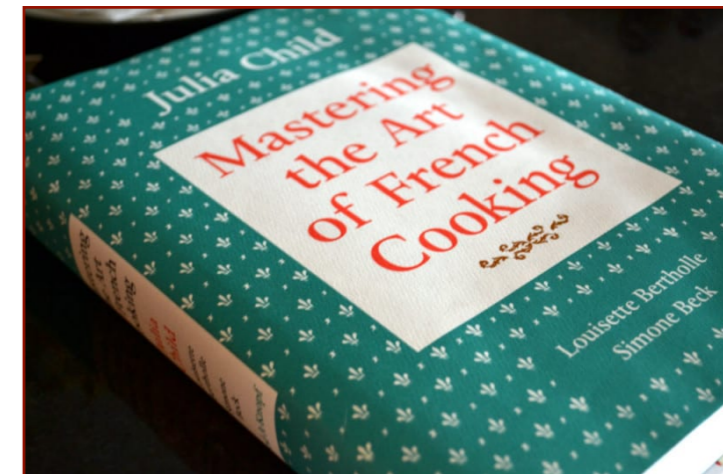
History

Consider the *Decision Problem* question.

- Is there an **algorithm** that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement?

What's an algorithm?

- An effective or computable procedure.
- An effectively calculable function.
- A decidable predicate.
- A recipe for solving a problem.
- How does one express an algorithm?



Lambda Calculus

History

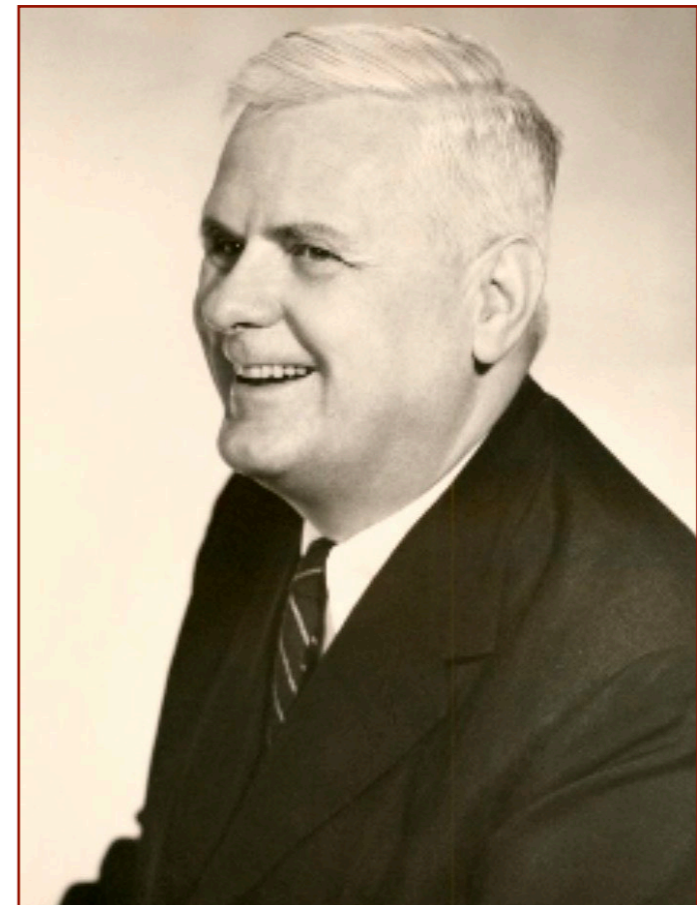
Consider the *Decision Problem* question.

- Is there an **algorithm** that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement?

What's an algorithm?

- An effective or computable procedure.
- An effectively calculable function.
- A decidable predicate.
- A recipe for solving a problem.

- How does one express an algorithm?
 - mathematically: **Lambda Calculus**



Alonzo Church
Wikipedia

Lambda Calculus

History

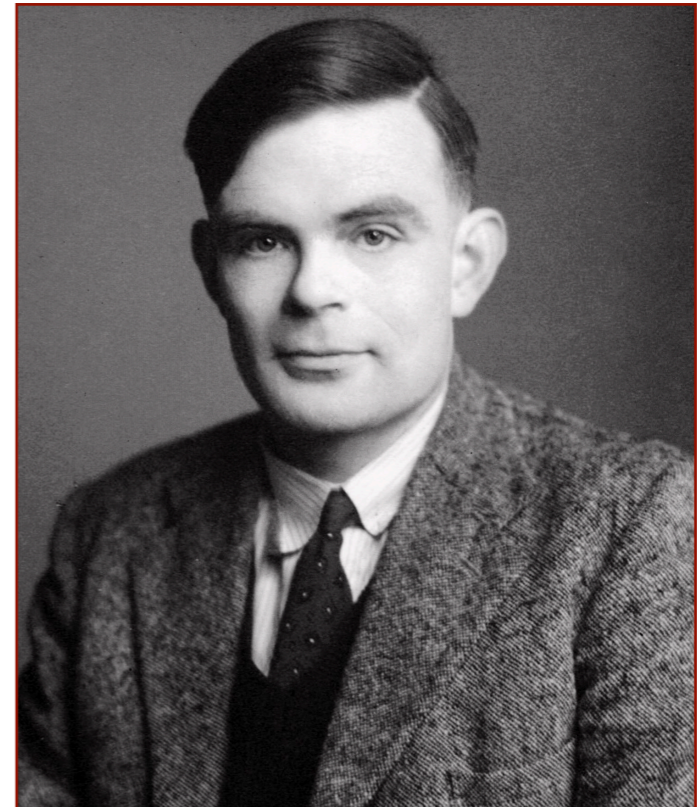
Consider the *Decision Problem* question.

- Is there an **algorithm** that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement?

What's an algorithm?

- An effective or computable procedure.
- An effectively calculable function.
- A decidable predicate.
- A recipe for solving a problem.

- How does one express an algorithm?
 - mathematically: Lambda Calculus
 - mechanically: **Turing Machine**



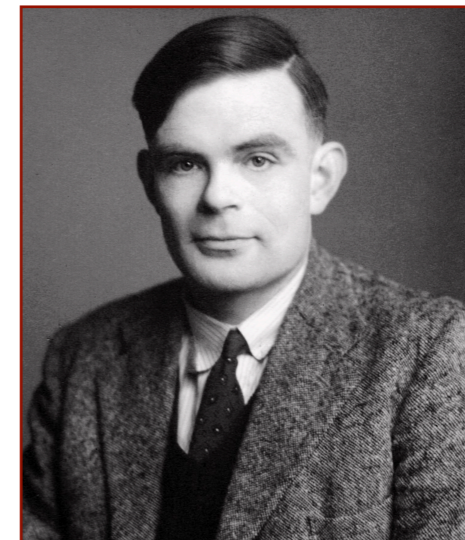
Alan Turing
Godrey Argent Studio, via The Royal Society

Lambda Calculus

History

1936 - Church and Turing independently answer the *Decision Problem* question: No.

- Church's Theorem
 - uses the Lambda Calculus as a model for algorithms / universal computation.
- Turing's Proof
 - uses a Turing Machine as a model for algorithms / universal computation.
- Their conclusion:
There is **no** algorithm that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement.

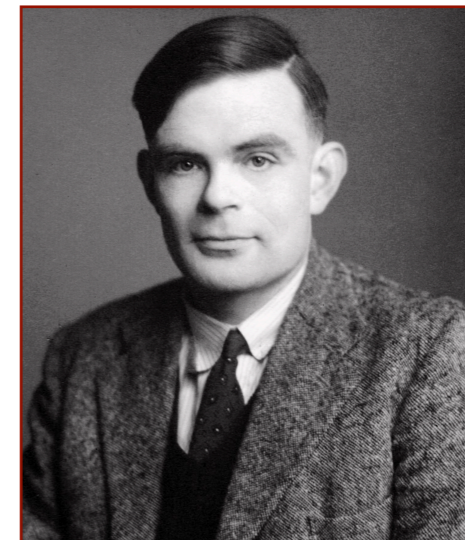


Lambda Calculus

History

1936 - Church and Turing independently answer the *Decision Problem* question: No.

- Church's Theorem and Turing's Proof state:
- There is **no** algorithm that takes as input a description of a formal language and a mathematical statement expressed in that language and outputs true or false depending on the mathematical validity of the statement.
- Their proofs of the **undecidability** of the *Decision Problem* involve, among other things, Cantor's diagonalization argument and the *halting problem* (the idea that it's possibly impossible to prove that a given algorithm terminates).



A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking.

Imagine the following program:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking.

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

This halts and returns true if the passed-in program does not halt when applied to itself, and it loops forever (i.e., does not halt) otherwise.

Trouble indeed.

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking.

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)`?

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking.

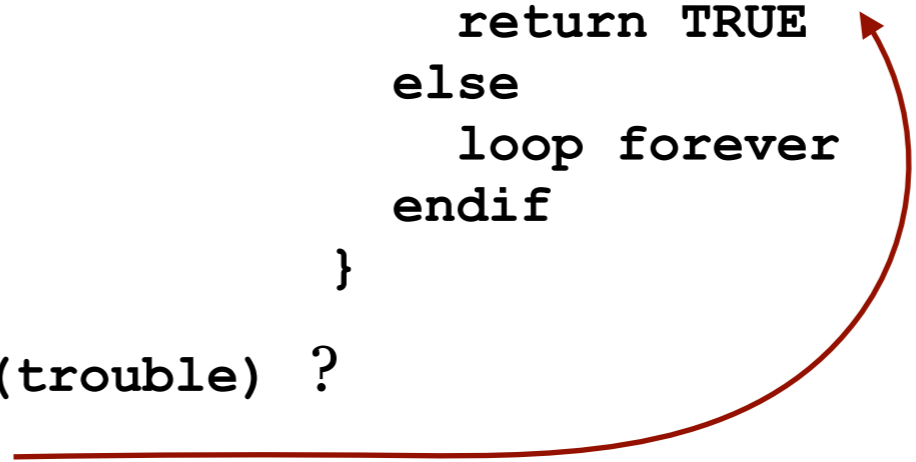
Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)` ?

It evaluates `halts(trouble, trouble)`



A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking?

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)` ?

It evaluates `halts(trouble,trouble)` There are two possibilities:

(1) it returns TRUE. Not TRUE is FALSE so loop forever, meaning do not halt.

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking?

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)` ?

It evaluates `halts(trouble,trouble)` There are two possibilities:

- (1) it returns TRUE. Not TRUE is FALSE so loop forever, meaning do not halt.
- (2) it returns FALSE. Not FALSE is TRUE so return TRUE, meaning it halts.

What?

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



Surely, I must be joking...

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)` ?

It evaluates `halts(trouble,trouble)` There are two possibilities:

- (1) it returns TRUE. Not TRUE is FALSE so loop forever, meaning do not halt.
- (2) it returns FALSE. Not FALSE is TRUE so return TRUE, meaning it halts.

In other words, if `halts(trouble,trouble)` halts then it doesn't, and if `halts(trouble,trouble)` doesn't halt then it does. It halts and loops at the same time.

A Fun Aside: The Halting Problem

What's that about proving termination being possibly impossible?



I'm not joking. And don't call me Shirley.

Imagine the following programs:

```
halts(p,i) = function {  
  if program p halts on input i  
    return TRUE  
  else  
    return FALSE  
  endif  
}
```

```
trouble(p) = function {  
  if not halts(p,p)  
    return TRUE  
  else  
    loop forever  
  endif  
}
```

What happens when we call `trouble(trouble)` ?

It evaluates `halts(trouble,trouble)` There are two possibilities:

(1) it returns TRUE. Not TRUE is FALSE so loop forever, meaning do not halt.

(2) it returns FALSE. Not FALSE is TRUE so return TRUE, meaning it halts.

In other words, if `halts(trouble,trouble)` halts then it doesn't, and if `halts(trouble,trouble)` doesn't halt then it does. It halts and loops at the same time.

`halts()` is a contradiction. Contradictions cannot exist. Therefore, by our own reasoning, `halts()` cannot exist and promptly vanishes in a puff of logic.

Lambda Calculus

History

1936 - Church and Turing independently answer the *Decision Problem* question: No.

- Church-Turing Thesis:
Lambda Calculus == Turing Machine
- Turing's 1936 paper:

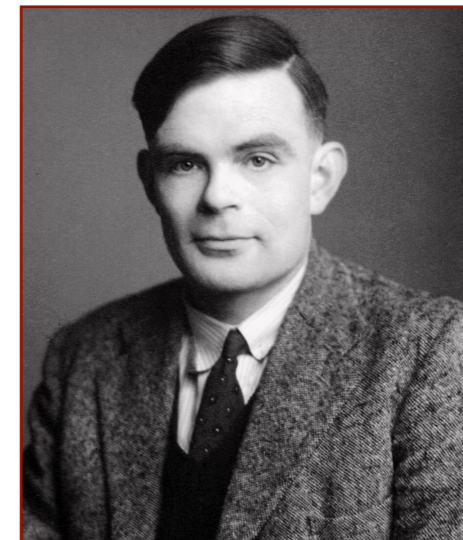
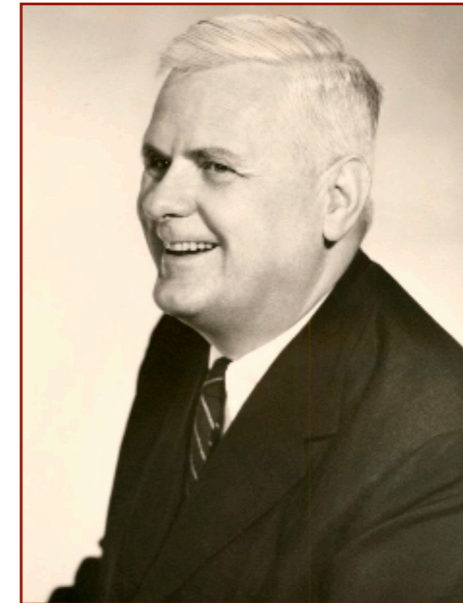
1936.] ON COMPUTABLE NUMBERS. 231

have valuable applications. In particular, it is shown (§11) that the Hilbertian Entscheidungsproblem can have no solution.

In a recent paper Alonzo Church† has introduced an idea of “effective calculability”, which is equivalent to my “computability”, but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem‡. The proof of equivalence between “computability” and “effective calculability” is outlined in an appendix to the present paper.

1. *Computing machines.*

“Entscheidungsproblem” is “Decision Problem” in German.

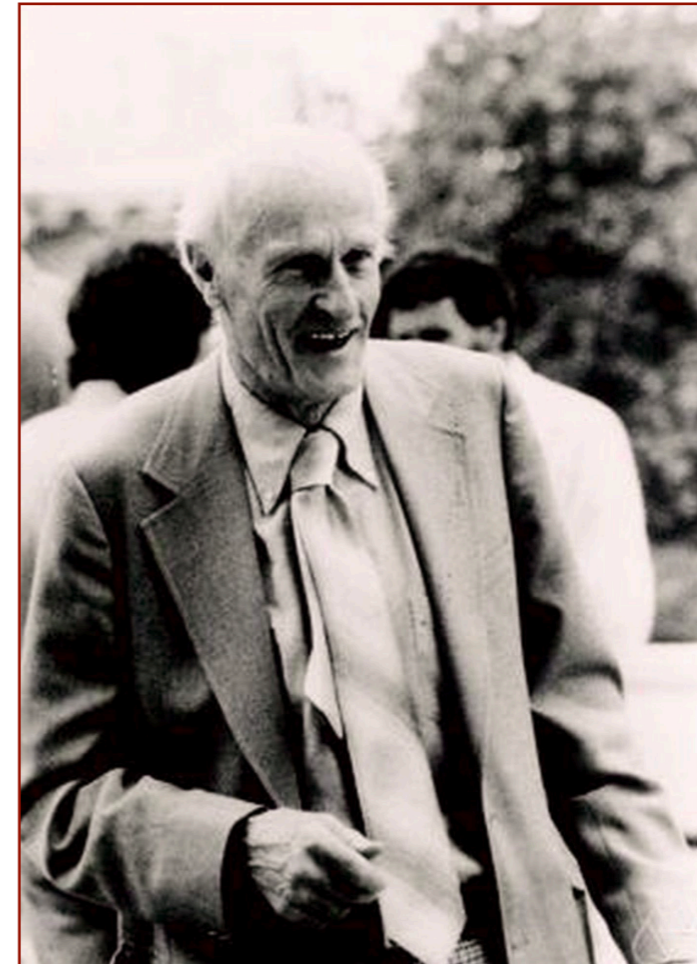


Lambda Calculus

History

1940s - Stephen Kleene

- Church's student.
- Showed the Lambda Calculus to be a universal computing language.
- Worked on recursion theory and computable functions.
- Described automata models with a mathematical notation called *regular sets*, which we often call Regular Expressions
 - A **RegEx** is a string that describes a set of other strings according to certain syntax rules.
 - A feature in many programming languages.
 - Used to specify grammar formalities.
 - Basis in/of Formal Languages and Automata Theory



Stephen Cole Kleene
Wikipedia

Lambda Calculus

History

1940s and 1950s - Haskell Curry

- Extended Moses Schönfinkel's work in Combinatory Logic.
- Worked to show that Combinatory Logic could provide a foundation for mathematics.
- Discovers/develops the Y combinator.
- Combinatory Logic is the foundation for many functional programming languages
 - Haskell is one of them.
 - Many (most?) other functional languages also stem from the Lambda Calculus and Combinator Theory.
 - The earliest one is . . .



Haskell Brooks Curry

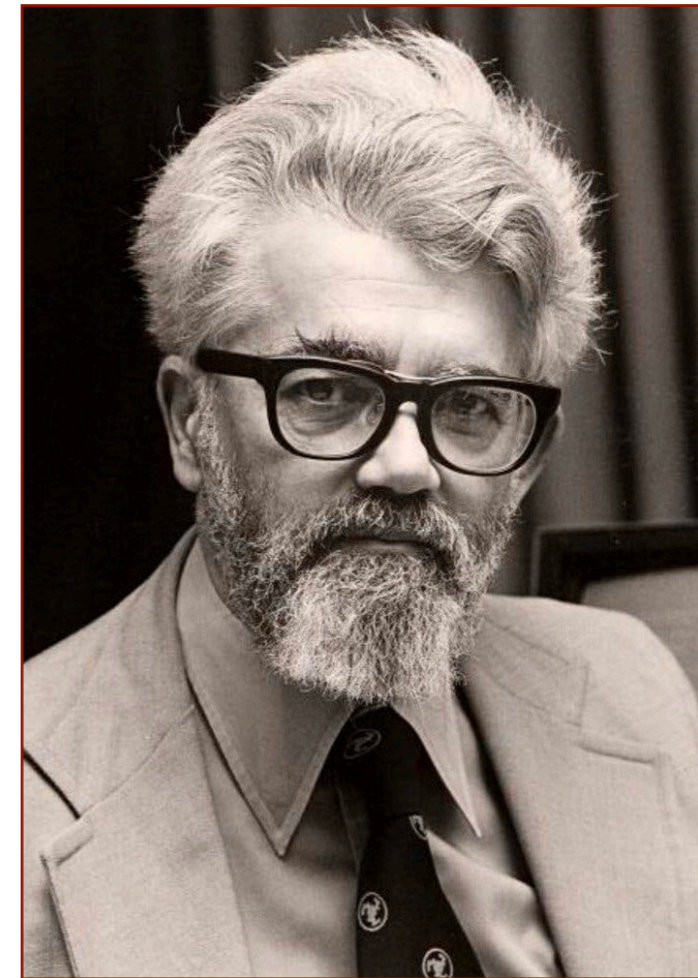
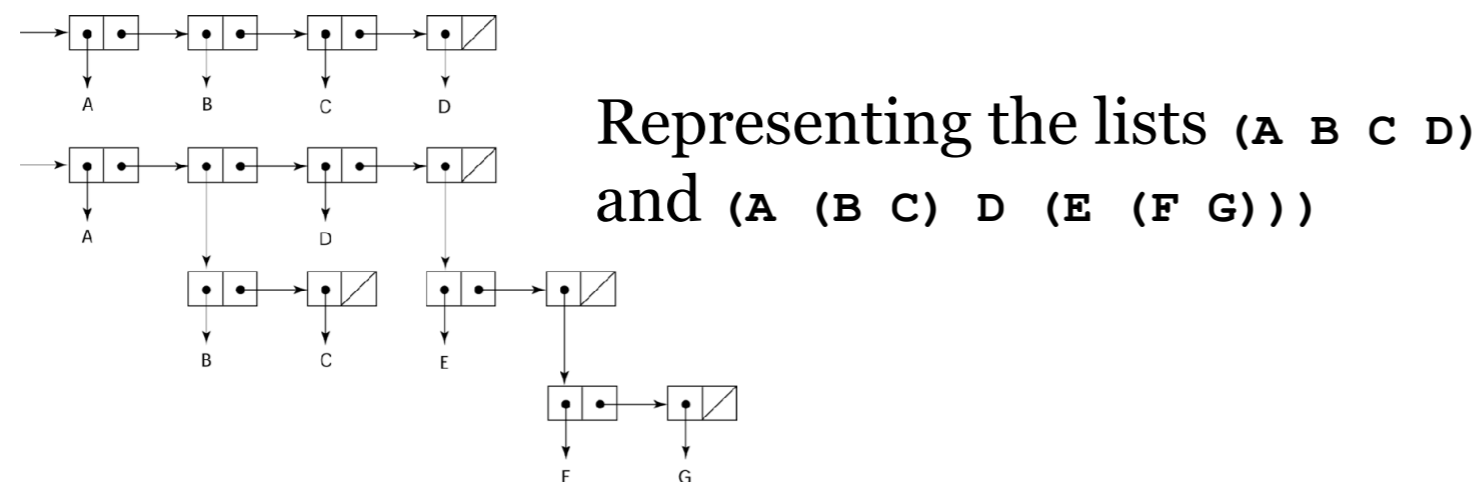
Wikipedia

Lambda Calculus

History

1950s - John McCarthy invents LISP

- **LIS**t **P**rocessing language
- Designed at MIT by John McCarthy
- AI research needed a language to process data in lists (rather than arrays) and also
- Symbolic computation (rather than numeric).
- Only two data types: atoms and lists



John McCarthy
New York Times

- Syntax is based on the Lambda Calculus

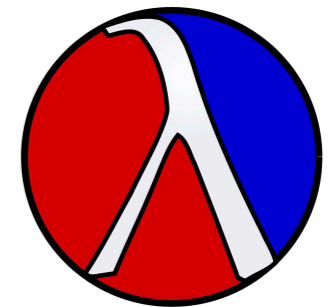
Lambda Calculus

History

1960s through 1980s - More functional Languages

- ISWIM - Peter Landin (“syntactic sugaring of the Lambda Calculus”)
- ML
- Miranda
- Haskell
- Erlang
- Scheme
- F#

- ... and many others



Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

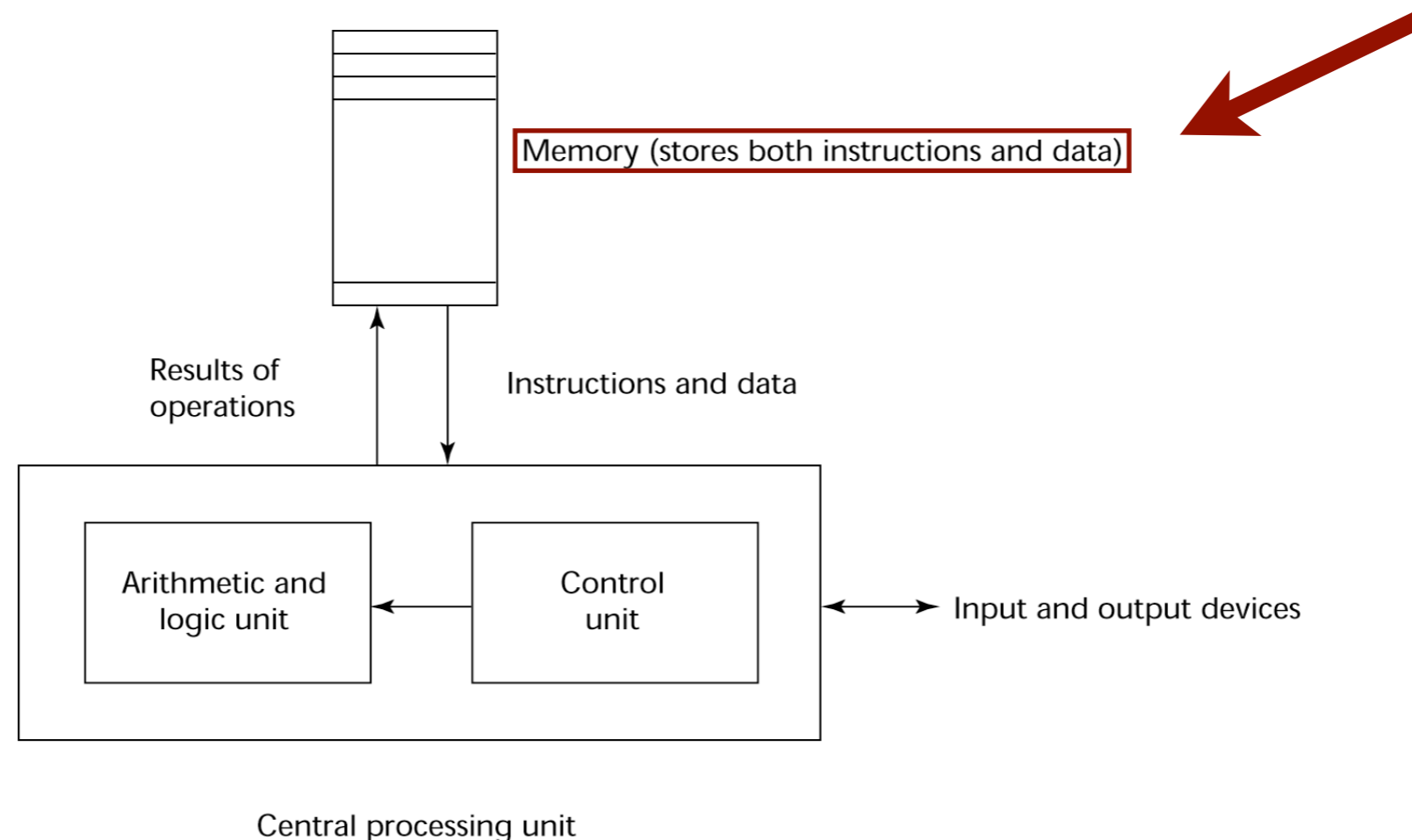
An **expression**, E , represents both algorithm and input. (Reflect on that for a moment: There is no difference between code and data.)

Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

An **expression**, E , represents both algorithm and input. (Reflect on that for a moment: There is no difference between code and data.)

Remember the von Neumann Architecture:



Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

An expression, E , represents both algorithm and input. (Reflect on that for a moment: There is no difference between code and data.)

Reduction happens when we **substitute** a part of E (let's call that P) with another part (P') according to the rewrite rules.

We repeat this process of reduction until the resulting expression cannot be reduced any further (i.e., it contains no more parts that can be reduced). This results in the **Normal Form** of E and represents the output of the functional program.

Let's do an example using basic math.

Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

Expression: $(7 + 4) \cdot (8 + 5 \cdot 3)$

Rewrite rules: addition and multiplication in math

$$\begin{aligned} (7 + 4) \cdot (8 + 5 \cdot 3) &\rightarrow 11 \cdot (8 + 5 \cdot 3) \\ &\rightarrow 11 \cdot (8 + 15) \\ &\rightarrow 11 \cdot (23) \\ &\rightarrow 253 \end{aligned}$$

← Normal form

Rewriting the expression using substitution, following the rules of addition and multiplication, yields the normal form (result).

By the way, we didn't need to do reductions in this order.

Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

Expression: $(7 + 4) \cdot (8 + 5 \cdot 3)$

Rewrite rules: addition and multiplication in math

$$\begin{aligned} (7 + 4) \cdot (8 + 5 \cdot 3) &\rightarrow (7 + 4) \cdot (8 + 15) \\ &\rightarrow 11 \cdot (8 + 15) \\ &\rightarrow 11 \cdot (23) \\ &\rightarrow 253 \end{aligned}$$

← Normal form

Reduction systems often satisfy the *Church-Rosser* property, which states that the normal form can be obtained regardless of the order in which reductions are performed (so long as they obey the rewrite rules).

Reduction works with symbolic systems too.

Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

Expression: `first(sort(append('dog', 'rabbit') (sort(('mouse', 'cat')))))`

Rewrite rules: list operations

Reduction and Functional Programming

A functional program consists of **expressions** and **rewrite rules**.

Expression: `first(sort(append('dog', 'rabbit') (sort(('mouse', 'cat')))))`

Rewrite rules: list operations

```
first(sort(append('dog', 'rabbit') (sort(('mouse', 'cat')))))  
→ first(sort(append('dog', 'rabbit') ('cat', 'mouse')))  
→ first(sort('dog', 'rabbit', 'cat', 'mouse'))  
→ first('cat', 'dog', 'mouse', 'rabbit')  
→ 'cat' ←————— Normal form
```

Alonzo Church generalized this.

Lambda Calculus

The smallest universal programming language in the world.

Two keywords: “ λ ” and “.”

Why λ ?

According to Dutch logician and lambda calculus scholar Henk

Barendregt:

We end this introduction by telling what seems to be the story how the letter ‘ λ ’ was chosen to denote function abstraction. In [1] Principia Mathematica the notation for the function f with $f(x) = 2x + 1$ is $\hat{x}.2x + 1$. Church originally intended to use the notation $\hat{x}.2x + 1$. The typesetter could not position the hat on top of the x and placed it in front of it, resulting in

$$\wedge x.2x + 1.$$

Then another typesetter changed it into $\lambda x.2x + 1$.

[1] Russell and Whitehead, *Principia Mathematica*, vol. 1 and 2, Cambridge University Press, 1910–13.

Lambda Calculus

The smallest universal programming language in the world.

Two keywords: “ λ ” and “.”

One transformation rule: Substitution

One function definition scheme:

```
<expression> := <function> | <application> | <var>
<function>   :=  $\lambda$  <var> . <expression>
<application> := <expression> <expression>
               := (<expression> <expression>)
<var>        := a, b, c, ... , z
```

Lambda Calculus

The smallest universal programming language in the world.

Two keywords: “ λ ” and “.”

One transformation rule: Substitution

One function definition scheme:

```
<expression> := <function> | <application> | <var>
<function>   :=  $\lambda$  <var> . <expression>
<application> := <expression> <expression>
               := (<expression> <expression>)
<var>        := a, b, c, ... , z
```

By the way, that is 7 productions (if we assume a range for the vars). That's all we need for computing. Really. **That's it.** Other languages have hundreds or thousands of productions in their grammar. We have seven.

Lambda Calculus

The smallest universal programming language ~~in the world~~.

`<expression>` := `<function>` | `<application>` | `<var>`
`<function>` := λ `<var>` . `<expression>`
`<application>` := `<expression>` `<expression>`
 := (`<expression>` `<expression>`)
`<var>` := `a`, `b`, `c`, ... , `z`

$\lambda z . E$

Read it as “Lambda z dot E”.

Lambda functions are abstractions of computation.

Lambda Calculus

The smallest universal programming language ~~in the world~~.

`<expression>` := `<function>` | `<application>` | `<var>`
`<function>` := λ `<var>` . `<expression>`
`<application>` := `<expression>` `<expression>`
 := (`<expression>` `<expression>`)
`<var>` := a, b, c, ... , z

$\lambda z . E$

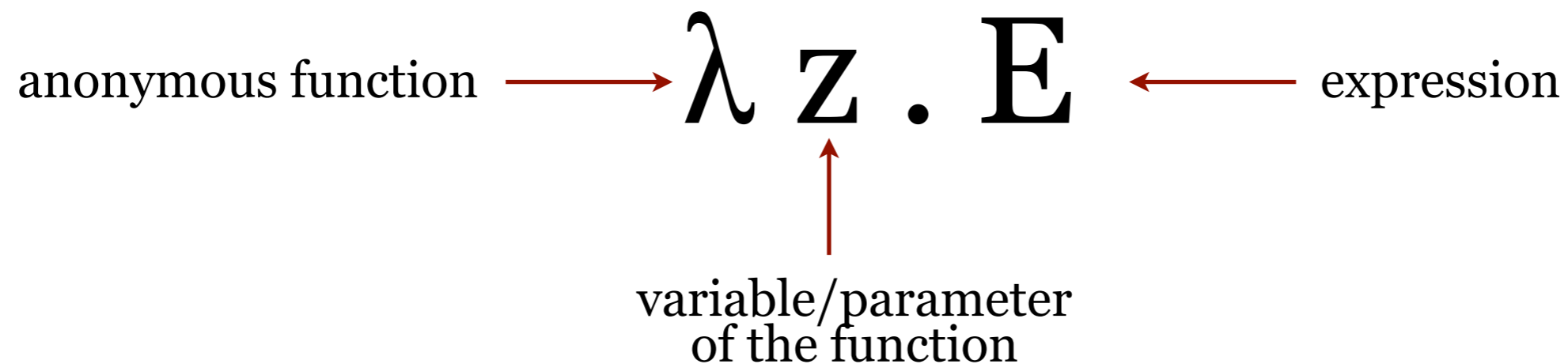
Read it as “Lambda z dot E”.

Lambda functions are abstractions of fundamental, universal computation.

Lambda Calculus

The smallest universal programming language ~~in the world~~.

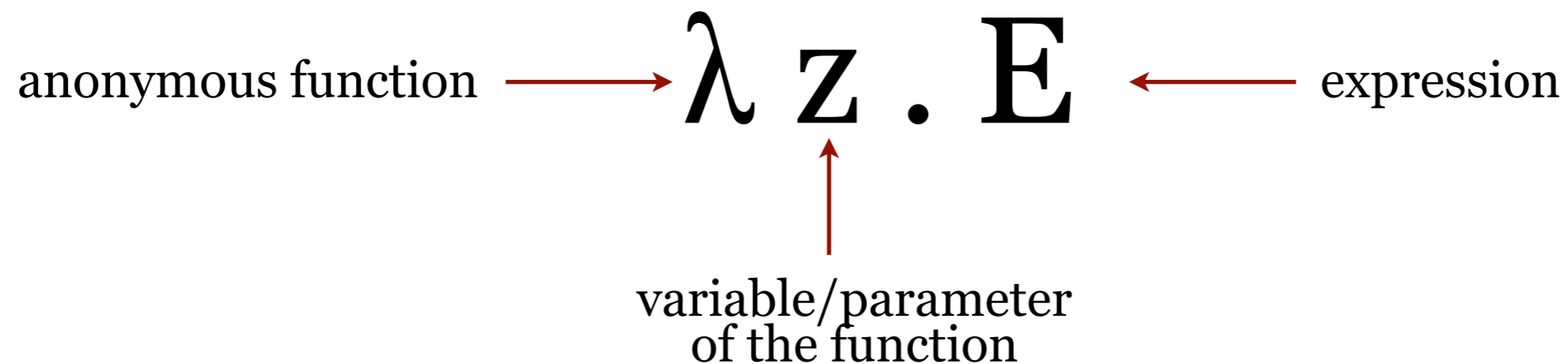
`<expression>` := `<function>` | `<application>` | `<var>`
`<function>` := λ `<var>` . `<expression>`
`<application>` := `<expression>` `<expression>`
 := (`<expression>` `<expression>`)
`<var>` := `a, b, c, ... , z`



Lambda Calculus

The smallest universal programming language ~~in the world~~.

$\langle \text{expression} \rangle ::= \langle \text{abstraction} \rangle \mid \langle \text{combinator} \rangle \mid \langle \text{param} \rangle$
 $\langle \text{abstraction} \rangle ::= \lambda \langle \text{param} \rangle . \langle \text{expression} \rangle$
 $\langle \text{application} \rangle ::= \langle \text{expression} \rangle \langle \text{expression} \rangle$
 $\quad ::= (\langle \text{expression} \rangle \langle \text{expression} \rangle)$
 $\langle \text{param} \rangle ::= a, b, c, \dots, z$



Lambda Calculus

The smallest universal programming language

$$\lambda z . E$$

Example: $f(x) = x^2 + 2 \cdot x + 1$

$g(x) = x^2 + 2 \cdot x + 1$

∴ It doesn't matter what we call it...

$whatever(x) = x^2 + 2 \cdot x + 1$

∴ ... so let's call it ...

$lambda(x) = x^2 + 2 \cdot x + 1$

∴ ... and write it like this:

$\lambda x . x^2 + 2 \cdot x + 1$

Lambda Calculus

The smallest universal programming language

$$\lambda z . E$$

Example:

$f(x) = x^2 + 2 \cdot x + 1$	$f(3) = 16$
$g(x) = x^2 + 2 \cdot x + 1$	$g(3) = 16$
$\lambda x . x^2 + 2 \cdot x + 1$	$(\lambda x . x^2 + 2 \cdot x + 1) (3) = 16$

Wait, what?

An aside about parameters:

Formal parameters are declared in the function.

Actual parameters are what's used when calling the function.

Lambda Calculus

The smallest universal programming language

$\lambda z . E$

We have a function called **f** that takes **x** as a parameter for the expression $x^2 + 2x + 1$.

$$\mathbf{f}(x) = x^2 + 2 \cdot x + 1$$

$$\mathbf{g}(x) = x^2 + 2 \cdot x + 1$$

$$\lambda x . x^2 + 2 \cdot x + 1$$

We have a function called **g** that takes **x** as a parameter for the expression $x^2 + 2 \cdot x + 1$.

$$f(3) = 16 \quad \text{We can invoke } \mathbf{f} \text{ and } \mathbf{g} \text{ by name.}$$

$$g(3) = 16$$

$$(\lambda x . x^2 + 2 \cdot x + 1) (3) = 16$$

Lambda Calculus

The smallest universal programming language

$\lambda z . E$

We have a function called **f** that takes **x** as a parameter for the expression $x^2 + 2x + 1$.

$$\mathbf{f}(x) = x^2 + 2 \cdot x + 1$$

$$\mathbf{g}(x) = x^2 + 2 \cdot x + 1$$

$$\lambda x . x^2 + 2 \cdot x + 1$$

$$f(3) = 16$$

$$g(3) = 16$$

$$(\lambda x . x^2 + 2 \cdot x + 1) (3) = 16$$

We can invoke **f** and **g** by name.

We have a function called **g** that takes **x** as a parameter for the expression $x^2 + 2 \cdot x + 1$.

We have an anonymous function that takes **x** as a parameter in the expression $x^2 + 2 \cdot x + 1$.

Since it's anonymous we can't invoke it by name so we have to give its code.

Imagine this as a function in some programming language.

Lambda Calculus

The smallest universal programming language

Imagine this as a function in some **any** programming language.

$$f(x) = x^2 + 2x + 1$$

$$f(3) = 16$$

```
function f(x)
begin
  return (x*x)+(2*x)+1
end
```

```
function g(x) {
  return (x*x)+(2*x)+1;
}
```

```
int whatever(x) {
  return (x*x)+(2*x)+1;
}
```

```
fun f x = (x*x)+(2*x)+1;
```

```
lambda (x) :
return (x*x)+(2*x)+1;
```

```
λ x . (x*x)+(2*x)+1
```

We can write a function that takes x as a parameter for the expression $x^2 + 2 \cdot x + 1$ in any programming language ...
... including a universal language like the Lambda Calculus.

To evaluate the function (to reduce the expression to its normal form) we need **substitution**.

$$(\lambda x . x^2 + 2 \cdot x + 1) (3) = 16$$

Lambda Calculus

Substitution

$$(\lambda x . x^2 + 2 \cdot x + 1) (3) = 16$$

“Substitute 3 for x in $x^2 + 2 \cdot x + 1$ ” is written
 $(x^2 + 2 \cdot x + 1) [3/x]$

Let's try it:

$$(\lambda \mathbf{x} . \mathbf{x}^2 + 2 \cdot \mathbf{x} + 1) (\mathbf{3})$$

$$\mathbf{3}^2 + 2 \cdot \mathbf{3} + 1 = 16$$

$$9 + 2 \cdot \mathbf{3} + 1 = 16$$

$$9 + 6 + 1 = 16$$

$$16 = 16 \longleftarrow \text{Normal form}$$

Lambda Calculus

Substitution

The general case:

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ **applied** to N , substitute N for z in E .”



That word — “applied” — seems important.

Lambda Calculus

What's Application?

$$\begin{array}{c} \langle \text{var} \rangle \\ \downarrow \\ (\lambda z . E)N \\ \uparrow \\ \langle \text{expression} \rangle \end{array} = E[N/z]$$

“Substitute N for z in E.”

<expression> := <function> | <application> | <var>
<function> := λ <var> . <expression>
<application> := <expression> <expression>
 := (<expression> <expression>)
<var> := a, b, c, ... , z

Lambda Calculus

What's Application?

$$\begin{array}{c} \langle \text{function} \rangle \\ \downarrow \\ (\lambda z . E) N \\ \uparrow \\ \langle \text{expression} \rangle \end{array} = E[N/z]$$

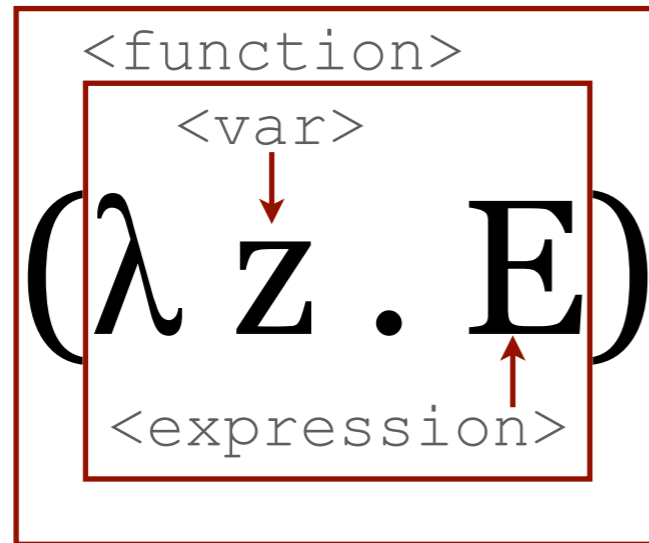
“Substitute N for z in E.”

$\langle \text{expression} \rangle ::= \langle \text{function} \rangle \mid \langle \text{application} \rangle \mid \langle \text{var} \rangle$
 $\langle \mathbf{function} \rangle ::= \lambda \langle \text{var} \rangle . \langle \text{expression} \rangle$
 $\langle \text{application} \rangle ::= \langle \text{expression} \rangle \langle \text{expression} \rangle$
 $\quad ::= (\langle \text{expression} \rangle \langle \text{expression} \rangle)$
 $\langle \text{var} \rangle ::= a, b, c, \dots, z$

Lambda Calculus

What's Application?

<expression>



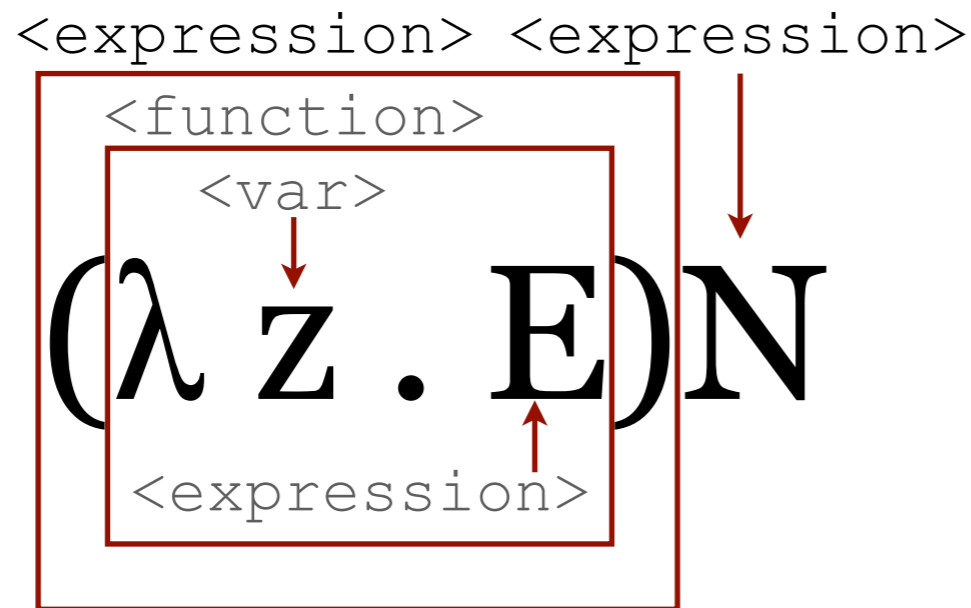
$$= E[N/z]$$

“Substitute N for z in E.”

<**expression**> := <function> | <application> | <var>
<function> := λ <var> . <expression>
<application> := <expression> <expression>
 := (<expression> <expression>)
<var> := a, b, c, ... , z

Lambda Calculus

What's Application?



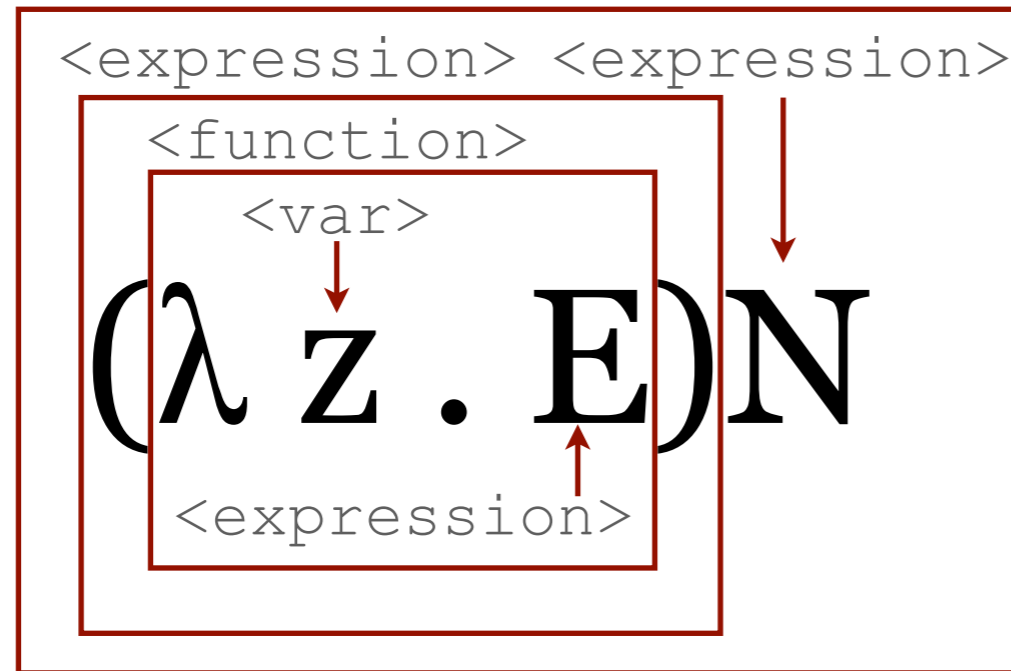
$$= E[N/z]$$

“Substitute N for z in E.”

\langle **expression** \rangle := \langle function \rangle | \langle application \rangle | \langle var \rangle
 \langle function \rangle := λ \langle var \rangle . \langle expression \rangle
 \langle application \rangle := \langle expression \rangle \langle expression \rangle
 := (\langle expression \rangle \langle expression \rangle)
 \langle var \rangle := a, b, c, ... , z

Lambda Calculus

What's Application?



$$= E[N/z]$$

“Substitute N for z in E.”

`<application>`

There's that word again.

`<expression> ::= <function> | <application> | <var>`
`<function> ::= λ <var> . <expression>`
`<application> ::= <expression> <expression>`
`::= (<expression> <expression>)`
`<var> ::= a, b, c, ... , z`

Lambda Calculus

Application

$(E_1 E_2)$

Read it as “ E_1 applied to E_2 .”

E_1 and E_2 are expressions.

E_1 and E_2 are sometimes Combinators, which are descendants of Moses Schönfinkel’s *combinator functions*.

`<expression>` := `<function>` | `<application>` | `<var>`
`<function>` := λ `<var>` . `<expression>`
`<application>` := `<expression>` `<expression>`
 := **(`<expression>` `<expression>`)**
`<var>` := `a, b, c, ... , z`

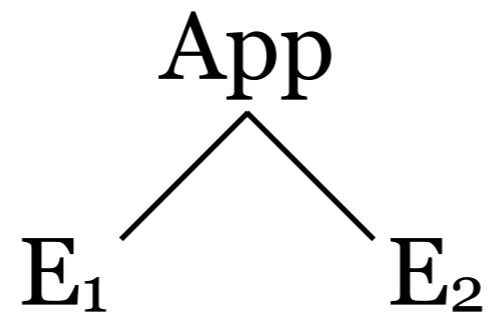
Lambda Calculus

Application

$$(E_1 E_2)$$

Read it as “ E_1 applied to E_2 .”

Think of it like a tree:



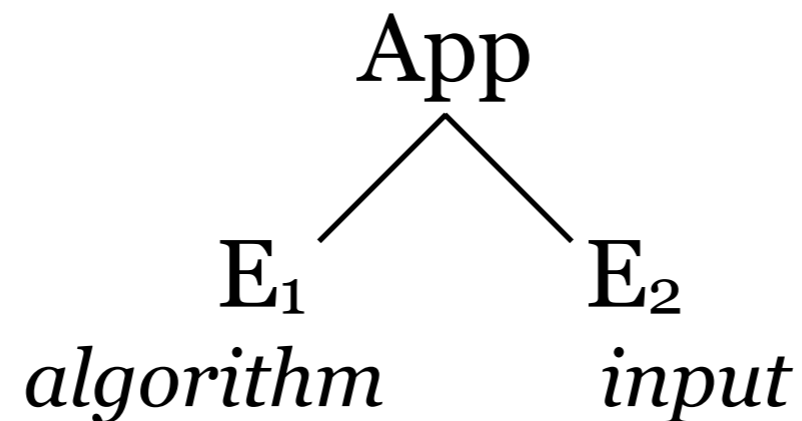
Lambda Calculus

Application

$(E_1 E_2)$

Read it as “ E_1 applied to E_2 .”

Think of it like a tree:



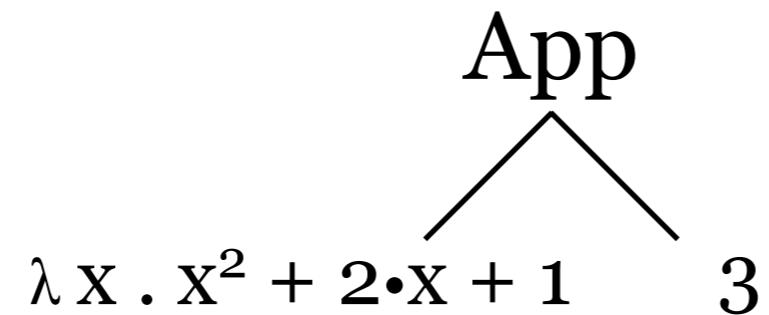
“The E_1 algorithm applied to E_2 input.”

Lambda Calculus

Application

$$(\lambda x . x^2 + 2 \cdot x + 1) (3)$$

Think of it like a tree:



This is the “algorithm” $(\lambda x . x^2 + 2 \cdot x + 1)$ applied to the input 3.

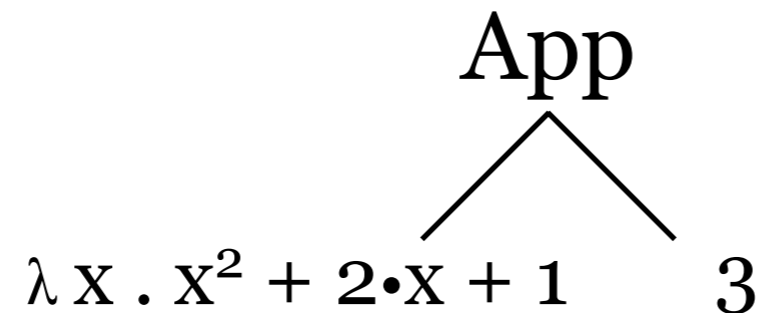
Wait a second! Where did this “3” come from?

Lambda Calculus

Wait a second!

$(\lambda x . x^2 + 2 \cdot x + 1) (3)$

Think of it like a tree:



There is no “3” in the syntax. What’s going on here?

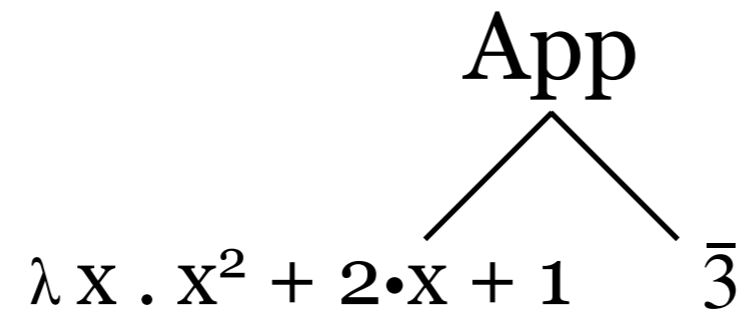
`<expression> := <function> | <application> | <var>`
`<function> := λ <var> . <expression>`
`<application> := <expression> <expression>`
`:= (<expression> <expression>)`
`<var> := a, b, c, . . . , z`

Lambda Calculus

Wait a second!

$$(\lambda x . x^2 + 2 \cdot x + 1) (3)$$

Think of it like a tree:



It's okay. “3” is *syntactic sugar* for a Church numeral.

$$\text{Let } \bar{0} = \lambda f x . x$$

$$\text{Let } \bar{1} = \lambda f x . (f x)$$

$$\text{Let } \bar{2} = \lambda f x . (f (f x))$$

$$\text{Let } \bar{3} = \lambda f x . (f (f (f x)))$$

⋮

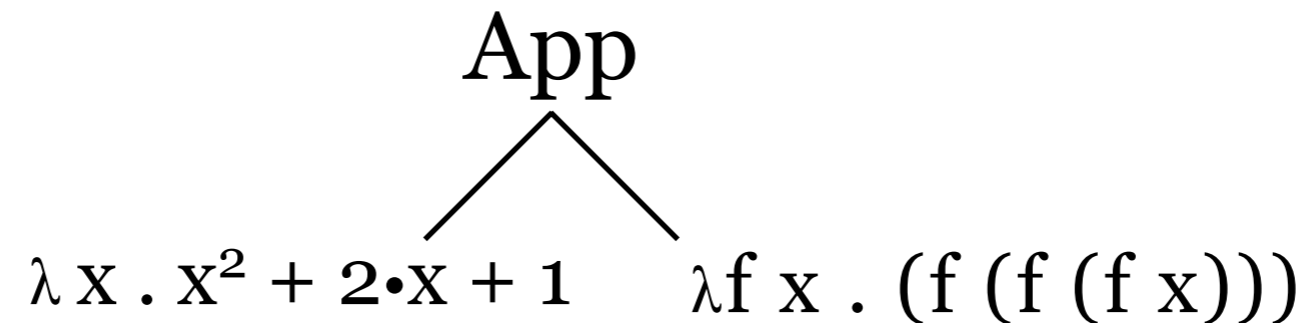
$$\text{Let } \bar{n} = \lambda f x . (f^n . x)$$

Lambda Calculus

Application

$$(\lambda x . x^2 + 2 \cdot x + 1) (3)$$

Think of it like a tree:



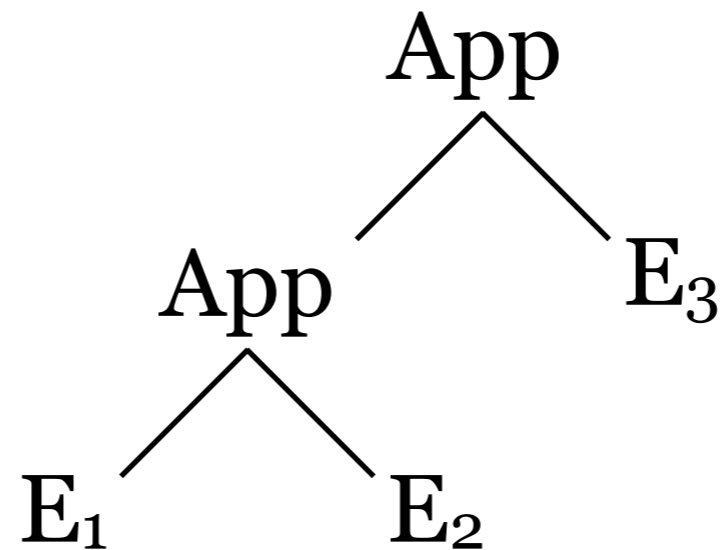
This is $(\lambda x . x^2 + 2 \cdot x + 1)$ applied to the input $\lambda f x . (f (f (f x)))$.

Lambda Calculus

Application is left-associative

$$E_1 E_2 E_3 = (E_1 E_2) E_3$$

Think of it like a tree:

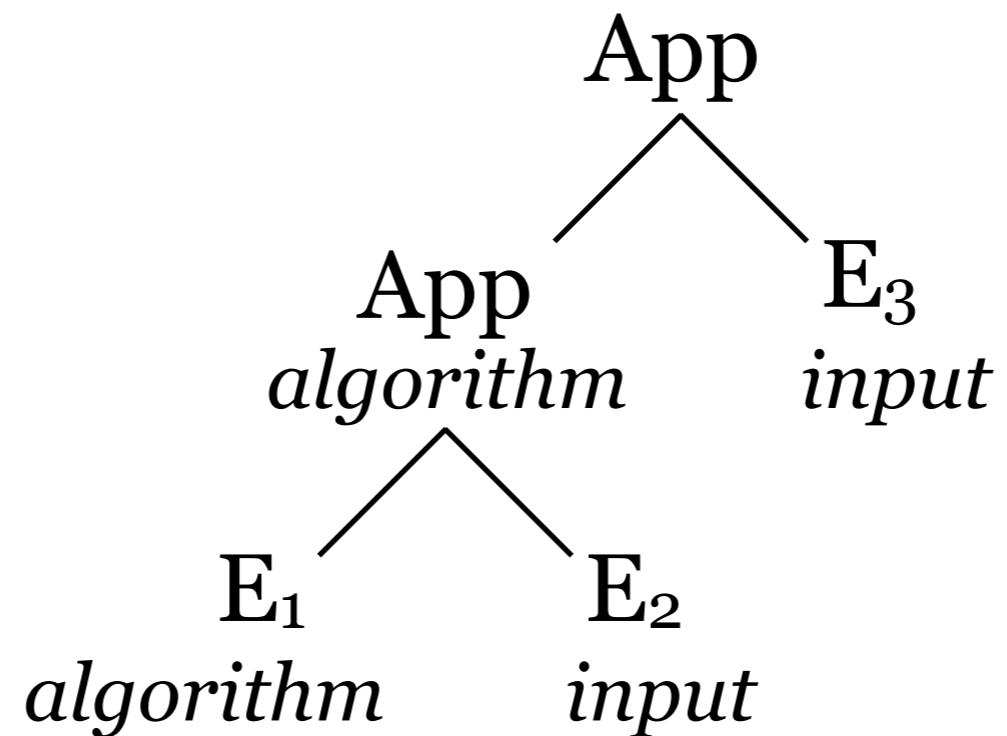


Lambda Calculus

Application is left-associative

$$E_1 E_2 E_3 = (E_1 E_2) E_3$$

Think of it like a tree:



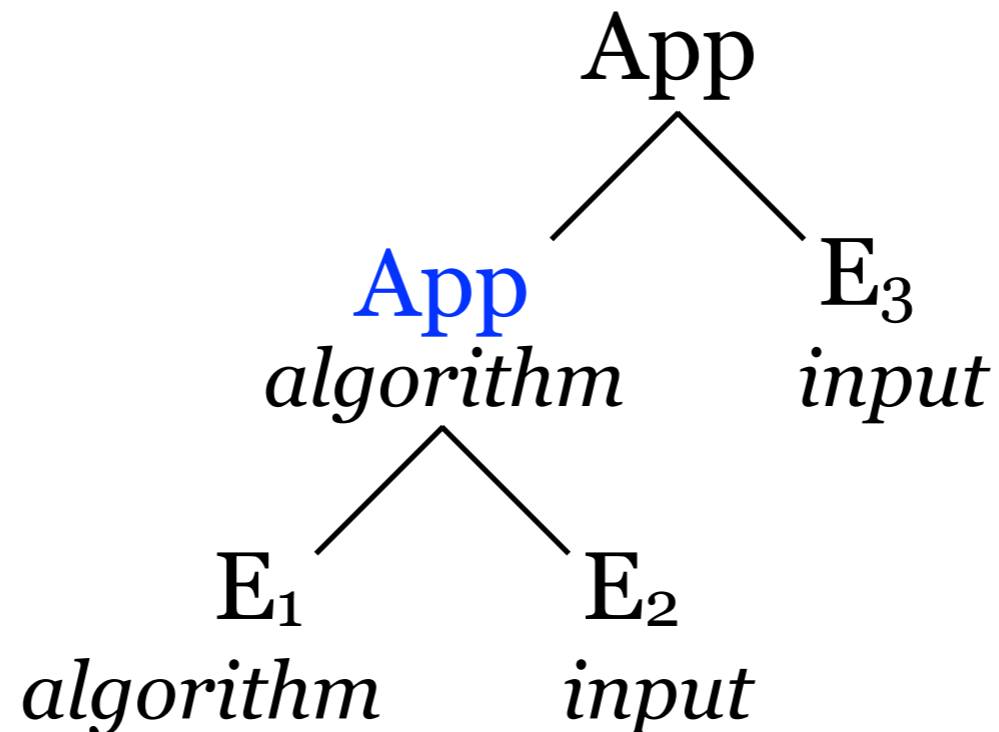
Lambda Calculus

Application is the essence of functional programming.

Remember, each expression can be code or data.

In this example, we're applying the E_1 "code" to the E_2 input, which results in (reduces to) a **partially evaluated intermediate expression** that acts like "code" when applied to E_3 .

This is the essence of functional programming.

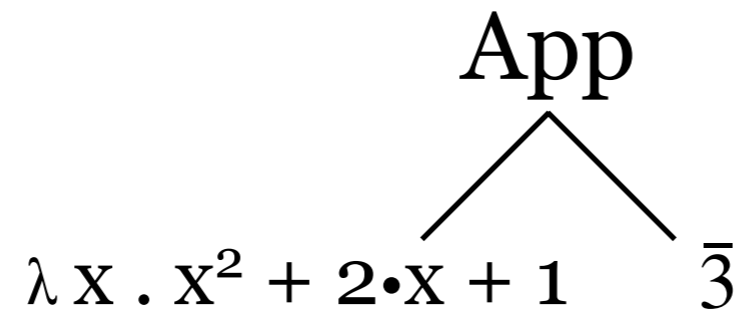


Lambda Calculus

Application

$$(\lambda x . x^2 + 2 \cdot x + 1) (3)$$

Think of it like a tree:



To perform application — that is, to execute the fundamental operation of computing — we use **substitution** to power **reduction**.

Lambda Calculus

Back to Substitution

The general case:

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ applied to N , **substitute** N for z in E .”



I told you it was important.

Lambda Calculus

Back to Substitution

The general case:

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ applied to N , substitute N for z in E .”

$$\begin{array}{l} (\lambda x . x^2 + 2 \cdot x + 1) (3) \\ (x^2 + 2 \cdot x + 1) [3/x] \end{array}$$

Substitution
binds actual parameter 3
to formal parameter x

Lambda Calculus

Back to Substitution

The general case:

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ applied to N , substitute N for z in E .”

$$\begin{array}{l} (\lambda x . x^2 + 2 \cdot x + 1) (3) \\ (x^2 + 2 \cdot x + 1) [3/x] \end{array}$$

Substitution
binds actual parameter 3
to formal parameter x

Note: When binding an actual parameter (3) to a formal parameter (x), we remove formal parameter from the resulting expression.

Lambda Calculus

Back to Substitution

The general case:

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ applied to N , substitute N for z in E .”

$$(\lambda x . x^2 + 2 \cdot x + 1) (3)$$

$$(x^2 + 2 \cdot x + 1) [3/x]$$

$$3^2 + 2 \cdot 3 + 1$$

$$9 + 2 \cdot 3 + 1$$

$$9 + 6 + 1$$

$$16$$

Substitution

binds actual parameter 3
to formal parameter x

Reductions

Normal form

Lambda Calculus

Substitution powers the Beta Reduction $\xrightarrow{\beta}$
Substitution is our model of computation.

$$(\lambda z . E)N = E[N/z]$$

“To evaluate $\lambda z . E$ applied to N , substitute N for z in E .”

$$\begin{aligned} & (\lambda x . x^2 + 2 \cdot x + 1) (3) \\ & (x^2 + 2 \cdot x + 1) [3/x] \\ & \rightarrow 3^2 + 2 \cdot 3 + 1 \\ & \rightarrow 9 + 2 \cdot 3 + 1 \\ & \rightarrow 9 + 6 + 1 \\ & \qquad \qquad \qquad 16 \end{aligned} \quad \left. \begin{array}{l} \curvearrowright \\ \curvearrowright \\ \curvearrowright \\ \curvearrowright \end{array} \right\} \beta\text{-Reductions}$$

Lambda Calculus

Handling Parameters / Partial Evaluation

$$\begin{array}{ccc} (\lambda x . \lambda y . (x y)) q & & (\lambda x y . (x y)) q \\ (\lambda y . (x y)) [q/x] & \xrightarrow{\beta\text{-Reduction}} & (\lambda y . (x y)) [q/x] \\ \rightarrow \lambda y . (q y) & & \rightarrow \lambda y . (q y) \end{array}$$

Note the shorthand: $\lambda x . \lambda y$ can be written $\lambda x \lambda y$ or $\lambda x y$.

In any case, it represents the same anonymous function that takes two parameters: x and y and returns $(x y)$. In terms of a programming language, it would be like this function...

```
function  $\lambda(x, y)$ 
begin
  return  $(x y)$ 
end
```

called with only one parameter, q . The best we can do is evaluate the part that we have, noting that the return value will be $(q y)$ and that we're still looking for the second parameter, y .

This is partial evaluation.

Lambda Calculus

Handling Parameters / Full Evaluation

$(\lambda x . \lambda y . (x y)) q r$
 $(\lambda y . (x y)) [q/x] \rightarrow \lambda y . (q y) r$ β -Reduction
 $(q y) [r/y] \rightarrow (q r)$ β -Reduction

$(\lambda x y . (x y)) q r$
 $(\lambda y . (x y)) [q/x] \rightarrow \lambda y . (q y) r$
 $(q y) [r/y] \rightarrow (q r)$ β -Reduction

In terms of a programming language...

```
function  $\lambda(x, y)$   
begin  
  return (x y)  
end
```

called with both parameters, q and r , returning $(q r)$.

Lambda Calculus

Functions as Parameters

$$\begin{aligned} & ((\lambda x . \lambda y . y x) 3) \text{succ} \\ & \quad (\lambda y . y x) [3/x] \\ & \xrightarrow{\beta} (\lambda y . y 3) \text{succ} \\ & \quad (y 3) [\text{succ}/y] \\ & \xrightarrow{\beta} \text{succ } 3 \\ & \xrightarrow{\beta} 4 \end{aligned}$$

Assume succ is the successor function: $\lambda n f x . n f (f x)$

The expression at the top applies the second argument to the first. In this case, it applies successor to 3.

We can use functions as parameters!

Lambda Calculus

Functions as Parameters

$$\begin{aligned} & ((\lambda x . \lambda y . y x) 3) \text{ pred} \\ & \quad (\lambda y . y x) [3/x] \\ & \xrightarrow{\beta} (\lambda y . y 3) \text{ pred} \\ & \quad (y 3) [\text{pred}/y] \\ & \xrightarrow{\beta} \text{pred } 3 \\ & \xrightarrow{\beta} 2 \end{aligned}$$

Assume pred is the predecessor function.

The expression at the top applies the second argument to the first. In this case, it applies predecessor to 3.

We can use functions as parameters! So cool.

Lambda Calculus

Functions as Parameters

$$\begin{aligned} & ((\lambda f . (\lambda x . (f (f x)))) \text{sqr}) 3 \\ & \quad (\lambda x . (f (f x))) [\text{sqr}/f] \\ \xrightarrow{\beta} & (\lambda x . (\text{sqr} (\text{sqr} x))) 3 \end{aligned}$$

$$\begin{aligned} & (\text{sqr} (\text{sqr} x)) [3/x] \\ \xrightarrow{\beta} & (\text{sqr} (\text{sqr} 3)) \\ \xrightarrow{\beta} & (\text{sqr} 9) \\ \xrightarrow{\beta} & 81 \end{aligned}$$

Assume `sqr` is the square function.

The expression at the top applies the first argument to the second argument twice. In this case, it applies `sqr` to `(sqr 3)`.

This is the basis of functional programming.

Lambda Calculus

Parameter Grouping and Partial Evaluation

$$\begin{aligned} & (\lambda a . \lambda m . m a) (q q) \\ & \lambda m . m a [(q q)/a] \\ \xrightarrow{\beta} & \lambda m . m (q q) \end{aligned}$$

We need to be careful with parentheses grouping. Note that there is only one parameter here: $(q q)$. Once we substitute/bind $(q q)$ for/to a in $\lambda m . m a$ and beta reduce, there are no more parameters to process so we have to stop with this partial evaluation.

But why not continue?

$$\begin{aligned} & \lambda m . m (q q) \\ & m[(q q)/m] \\ \xrightarrow{\beta} & (q q) \end{aligned}$$

Because the parameter $(q q)$ was already associated with a in the first beta reduction. We're not allowed to use it a second time.

Lambda Calculus

Parameter Grouping and Partial Evaluation

$$\begin{aligned} & (\lambda a . \lambda m . m a) (q q) \\ & \lambda m . m a [(q q)/a] \\ \xrightarrow{\beta} & \lambda m . m (q q) \quad \longleftarrow \text{Stop here.} \end{aligned}$$

Why not continue?

$$\begin{aligned} & \lambda m . m (q q) \\ & m[(q q)/m] \\ \xrightarrow{\beta} & q \end{aligned}$$

Because the parameter $(q q)$ was already associated with a in the first beta reduction. We're not allowed to use it a second time. Think of a function in a programming language:

```
function λ(x,y)
begin
  return (x y)
end
```

You can't take one actual parameter and use it for both formal parameters. That'd be madness!

Lambda Calculus

More on Partial Evaluation

Assume we have a plus operation as conventionally defined. (We don't, but that's not the point right now.)

$$\begin{aligned} & (\lambda a . \lambda m . a + m) (3) \\ &= (\lambda a m . a + m) (3) \\ & \quad \lambda m . a + m [3/a] \\ & \xrightarrow{\beta} \lambda m . 3 + m \end{aligned}$$

At this point we have a partially evaluated function that can best be called “add3”. Whatever parameter we eventually pass into it will have 3 added to it and the result (normal form) returned.

$$\begin{aligned} & (\lambda m . 3 + m) (4) \\ & \quad 3 + m [4/m] \\ & \xrightarrow{\beta} 3+4 \\ & \quad \quad 7 \end{aligned}$$

Lambda Calculus

Just One More on Partial Evaluation

$$\begin{aligned} & (\lambda a . \lambda m . a + m) (3) \\ &= (\lambda a m . a + m) (3) \\ & \quad \lambda m . a + m [3/a] \\ & \xrightarrow{\beta} \lambda m . 3 + m \end{aligned}$$

At this point we have a partially evaluated function that can best be called “add3”. But we cannot assume that the second parameter will also be 3 and beta-reduce it to 6. That would be madness! And it illustrates why we can use parameters only once.

Lambda Calculus

A Mystery Function

$\lambda x . x$

or

$\lambda a . a$

or

$\lambda q . q$

What happens when we apply it to something?

$(\lambda x . x) E$

Lambda Calculus

A Mystery Function

$\lambda x . x$

or

$\lambda a . a$

or

$\lambda q . q$

What happens when we apply it to something?

$(\lambda x . x) E$

$x [E/x]$

$\xrightarrow{\beta} E$

Lambda Calculus

A Mystery Function

$\lambda x . x$

or

$\lambda a . a$

or

$\lambda q . q$

What happens when we apply it to something else?

$(\lambda a . a) (\text{pred})$

$a [\text{pred}/a]$

$\xrightarrow{\beta} \text{pred}$

Lambda Calculus

Identity Function and Alpha-equivalence

$\lambda x . x$

or

$\lambda a . a$

or

$\lambda q . q$

This is the identity function. Let's call it "I" for short.

Note that we can rename variables if we're careful* about not changing the meaning of the expression.

This is called alpha equivalence. We say $\lambda x . x =_{\alpha} \lambda a . a$

*There's much more to consider when renaming variables, like which ones are **free** and which ones are **bound**, and making sure the free ones stay free.

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

Consider the *I combinator*, the identity function.

$$I \equiv \lambda x . x$$

The variable x is bound because it's a parameter used in the expression.

Combinators are λ functions with no free variables. We've finally caught up to where Schönfinkel was over 100 years ago.



Lambda Calculus

Free and Bound Variables

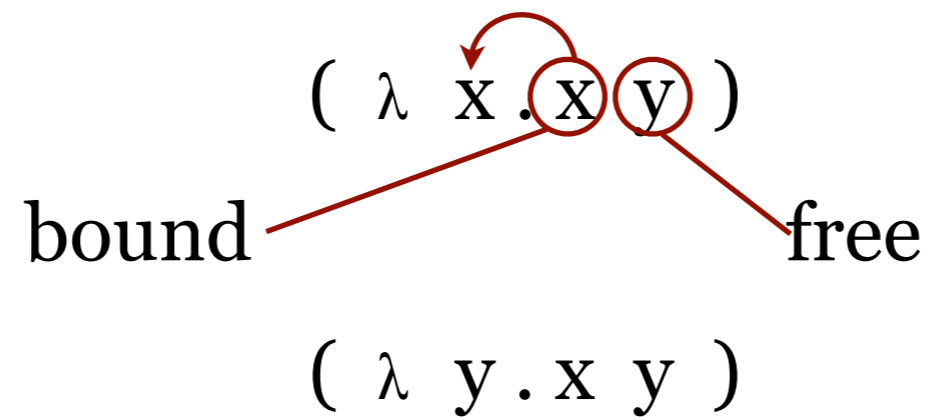
Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x. x y)$$
$$(\lambda y. x y)$$

Lambda Calculus

Free and Bound Variables

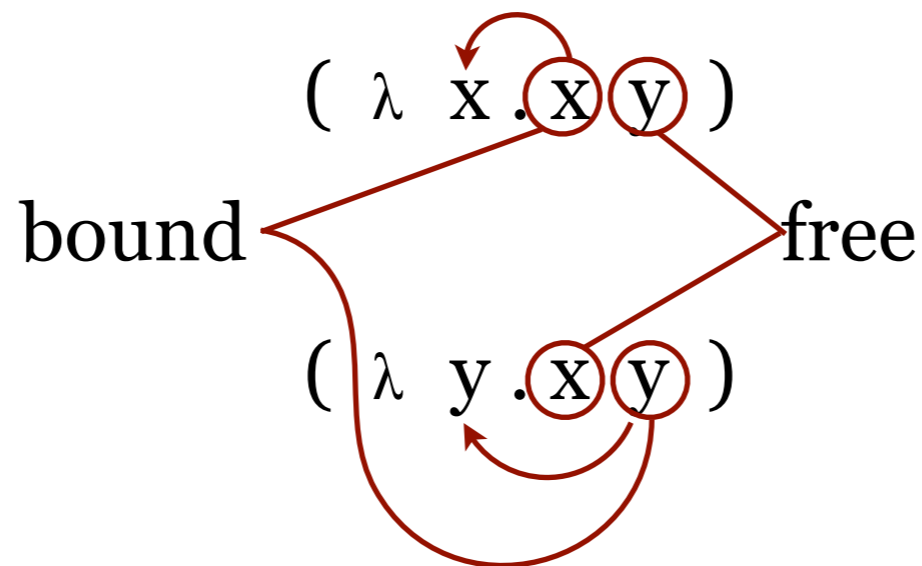
Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.



Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.



Lambda Calculus

Free and Bound Variables

Free Variables

$$\text{FV}(z) = \{z\}$$

$$\text{FV}(E_1 E_2) = \text{FV}(E_1) \cup \text{FV}(E_2)$$

$$\text{FV}(\lambda z.E) = \text{FV}(E) \setminus \{z\}$$

Bound Variables

$$\text{BV}(z) = \emptyset$$

$$\text{BV}(E_1 E_2) = \text{BV}(E_1) \cup \text{BV}(E_2)$$

$$\text{BV}(\lambda z.E) = \text{BV}(E) \cup \{z\}$$

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x . \lambda y . (x y)) q$$

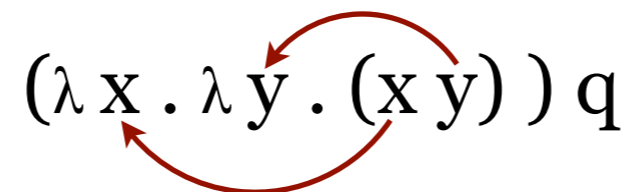
?

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$(\lambda x . \lambda y . (x y)) q$

The diagram shows the lambda expression $(\lambda x . \lambda y . (x y)) q$. Two red curved arrows originate from the lambda symbols. The first arrow starts at the first λ and points to the x in the body $(x y)$. The second arrow starts at the second λ and points to the y in the body $(x y)$. This illustrates that both x and y are bound variables within the scope of their respective lambda functions.

Both x and y are bound.

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$(\lambda x . (y x)) q$

?

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x . (y x)) q$$

Only x is bound. The variable y is free.

```
function  $\lambda(x)$ 
begin
  return (y x) // y is unbound/global
end
```

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x . (y x)) q$$

Only x is bound. The variable y is free.

```
function  $\lambda(x)$ 
begin
  return (y x) // y is unbound/global
end
```

We are allowed to rename bound variables. It's like renaming the local variables inside of a function. That's fine. But if you rename globals you use inside your function you ~~may~~ will break things.

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x . (y x)) q$$

Only x is bound. The variable y is free. We can rename bound vars.

$$(\lambda x . (y x)) q \stackrel{\alpha}{=} (\lambda a . (y a)) q$$

We cannot rename free vars.

$$(\lambda x . (y x)) q \not\stackrel{\alpha}{=} (\lambda x . (a x)) q$$

The var y is not passed in to the function yet it's present in the 'code' so it's global in that sense. Therefore we cannot rename it.

Lambda Calculus

Free and Bound Variables

Free variables are “unbound” or “global” in the sense that they are not in the scope of the λ function. Bound variables are those that are used in the expression and passed-in as a var/parameter.

$$(\lambda x . (y x)) q$$

Only x is bound. The variable y is free. We can rename bound vars.

$$(\lambda x . (y x)) q \equiv_{\alpha} (\lambda a . (y a)) q$$

But we cannot rename bound vars to capture free vars.

$$(\lambda x . (y x)) q \not\equiv_{\alpha} (\lambda y . (y y)) q$$

The variable y was free and must remain so. This means that while we can rename the bound variable x , we cannot rename it to y because doing so would capture y and thus deprive y of its freedom.

Lambda Calculus

Free and Bound Variables and Alpha-equivalence

Not all α -renames are allowed. We are not allowed to “capture” free variables.

$$\lambda x.E \stackrel{=}{\alpha} \lambda y.E \text{ iff } y \notin FV(E)$$

We can α -rename x to y in $\lambda x.E$ iff y is not free in E .

Because... if there were a free y in E , we would be “capturing” it by renaming our parameter x as y . And we cannot capture free variables; we cannot deny them their freedom. It would be wrong.

Perform α -renames and conversions **very** carefully.

Lambda Calculus

Free and Bound Variables

Free Variables

$$\text{FV}(z) = \{z\}$$

$$\text{FV}(E_1 E_2) = \text{FV}(E_1) \cup \text{FV}(E_2)$$

$$\text{FV}(\lambda z.E) = \text{FV}(E) \setminus \{z\}$$

λ expressions with no free variables are closed terms / Combinators.
Consider the *I combinator*:

$$I \equiv \lambda x . x$$

$$\begin{aligned} \text{FV}(I) &\equiv \text{FV}(\lambda x . x) \\ &= \text{FV}(x) \setminus \{x\} \\ &= \{x\} \setminus \{x\} \\ &= \emptyset \end{aligned}$$

Lambda Calculus

Combinators

We've seen the I combinator:

$$I \equiv \lambda x . x$$

It's a closed-term λ expression, meaning it has no free variables.

Usage:

$$\begin{array}{l} I \textit{ whatever} \\ \xrightarrow{\beta} \textit{ whatever} \end{array}$$

There are other combinators.

Lambda Calculus

Combinators

$$I \equiv \lambda x . x$$

$$K \equiv \lambda x y . x$$

$$K' \equiv \lambda x y . y$$

$$S \equiv \lambda x y z . x z (y z)$$

These are closed-term λ expressions, meaning they have no free variables. We can use I, K, and S like “macros” for their λ expressions.

Lambda Calculus

Combinators

$$I \equiv \lambda x . x$$

$$K \equiv \lambda x y . x$$

$$K' \equiv \lambda x y . y$$

$$S \equiv \lambda x y z . \underline{x z} (y z)$$

λ calculus is left-associative, so this is the application of $(x z)$ to $(y z)$

Lambda Calculus

Combinators

$$\mathbf{I} \equiv \lambda x . x$$

Identity.

$$\mathbf{K} \equiv \lambda x y . x$$

Take the first.

$$\mathbf{K}' \equiv \lambda x y . y$$

Take the second.

$$\mathbf{S} \equiv \lambda x y z . x z (y z)$$

Substitute and apply.

Usage:

$$\mathbf{I} E \xrightarrow{\beta} E$$

$$\mathbf{K} E_1 E_2 \xrightarrow{\beta} E_1$$

$$\mathbf{K}' E_1 E_2 \xrightarrow{\beta} E_2$$

$$\mathbf{S} E_1 E_2 E_3 \xrightarrow{\beta} E_1 E_3 (E_2 E_3)$$

Lambda Calculus

Combinators

$$I \equiv \lambda a . a$$

Identity.

$$K \equiv \lambda b c . b$$

Take the first.

$$K' \equiv \lambda b c . c$$

Take the second.

$$S \equiv \lambda e p z . e z (p z)$$

Substitute and apply.

After some α -conversions:

$$I M \xrightarrow{\beta} M$$

$$K M N \xrightarrow{\beta} M$$

$$K' M N \xrightarrow{\beta} N$$

$$S M N L \xrightarrow{\beta} M L (N L)$$

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

After more α -conversions:

$$I M \xrightarrow{\beta} M$$

$$K M N \xrightarrow{\beta} M$$

$$K' M N \xrightarrow{\beta} N$$

$$S M N L \xrightarrow{\beta} M L (N L)$$

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

α -convert

Example

$$I M$$

$$(\lambda x . x) M$$

$$x [M/x]$$

$$\xrightarrow{\beta} M$$

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

α -convert

Example

K M N

$$(\lambda x y . x) M N$$

$$[M/x] \text{ in } (\lambda y . x)$$

$$\xrightarrow{\beta} (\lambda y . M) N$$

$$[N/y] \text{ in } M$$

$$\xrightarrow{\beta} M$$

How do we know there's no y in M ? Even if there were a y in M , it's not the same as the y that's the second parameter in the closed term/combinator K .

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

α -convert

$$K' M N$$

$$\begin{aligned} & (\lambda x y . y) M N \\ & [M/x] \text{ in } (\lambda y . y) \\ \xrightarrow{\beta} & (\lambda y . y) N \\ & [N/y] \text{ in } y \\ \xrightarrow{\beta} & N \end{aligned}$$

Example

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

α -convert

Example

$$K' M N$$

$$(\lambda x y . y) M N$$

$$[M/x] \text{ in } (\lambda y . y)$$

$$\xrightarrow{\beta} (\lambda y . y) N$$

$$[N/y] \text{ in } y$$

$$\xrightarrow{\beta} N$$

I

Lambda Calculus

Combinators

$$I \equiv \lambda q . q$$

Identity.

$$K \equiv \lambda u v . u$$

Take the first.

$$K' \equiv \lambda u v . v$$

Take the second.

$$S \equiv \lambda a d c . a c (d c)$$

Substitute and apply.

α -convert

Example

S M N L

$$(\lambda e p z . e z (p z)) M N L$$

$$\lambda p z . e z (p z) [M/e]$$

$$\xrightarrow{\beta} \lambda p z . M z (p z) N L$$

$$\lambda z . M z (p z) [N/p]$$

$$\xrightarrow{\beta} \lambda z . M z (N z) L$$

$$M z (N z) [L/z]$$

$$\xrightarrow{\beta} M L (N L)$$

Lambda Calculus

Combinators

~~$$I \equiv \lambda x . x$$~~

$$K \equiv \lambda x y . x$$

~~$$K' \equiv \lambda x y . y$$~~

$$S \equiv \lambda x y z . x z (y z)$$

~~Identity.~~

Take the first.

~~Take the second.~~

Substitute and apply.

We don't need all of these.

In fact, all we need are S and K.

(Ponder that.)

All of computation can be expressed with two combinators: S and K.

It can get tricky, though.

For example, $Y = S(K(S(SKK)(SKK)))(S(S(KS)K)(K(S(SKK)(SKK))))$

Lambda Calculus

A full language?

- Sequence
- Alternation
- Repetition

Every programming language needs to provide mechanisms for

- sequence — executing commands one after the other
- alternation — decision-making: if-then
- repetition — looping, iteratively or recursively

So far all we have is sequence. We can apply things all we like by stringing together (sometimes very long) sequences of S and K (and other λ expressions too, but it all comes down to S and K).

We need alternation and repetition.

Lambda Calculus

Logic

We need logical values

- true
- false

and logical operators

- not
- and
- or

and logical properties

- not true = false
- not false = true
- true and true = true
- true and false = false
- true or true = true
- true or false = true

so we can build conditional statements for alternation.

- $(E_c E_1 E_2)$ — “If E_c then E_1 else E_2 ”

Sequence

→ Alternation

Repetition

Lambda Calculus

Logic

We need logical values

- true
- false

Can any of our combinators help? Perhaps K and K' ?

true = K “take the first”

false = K' “take the second”

Consider a conditional statement: $(E_c E_1 E_2)$ i.e., “If E_c then E_1 else E_2 ”

$$\begin{aligned} & (\mathit{true} E_1 E_2) \\ & \equiv (\mathbf{K} E_1 E_2) \\ & \equiv (\lambda x y . x) E_1 E_2 \\ & \quad [E_1/x] \text{ in } (\lambda y . x) \\ & \xrightarrow{\beta} (\lambda y . E_1) E_2 \\ & \quad [E_2/y] \text{ in } E_1 \\ & \xrightarrow{\beta} E_1 \end{aligned}$$

$$\begin{aligned} & (\mathit{false} E_1 E_2) \\ & \equiv (\mathbf{K}' E_1 E_2) \\ & \equiv (\lambda x y . y) E_1 E_2 \\ & \quad [E_1/x] \text{ in } (\lambda y . y) \\ & \xrightarrow{\beta} (\lambda y . y) E_2 \\ & \quad [E_2/y] \text{ in } y \\ & \xrightarrow{\beta} E_2 \end{aligned}$$

Lambda Calculus

Logic

We need logical values

- ✓ true
- ✓ false

and logical operators

- not
- and
- or

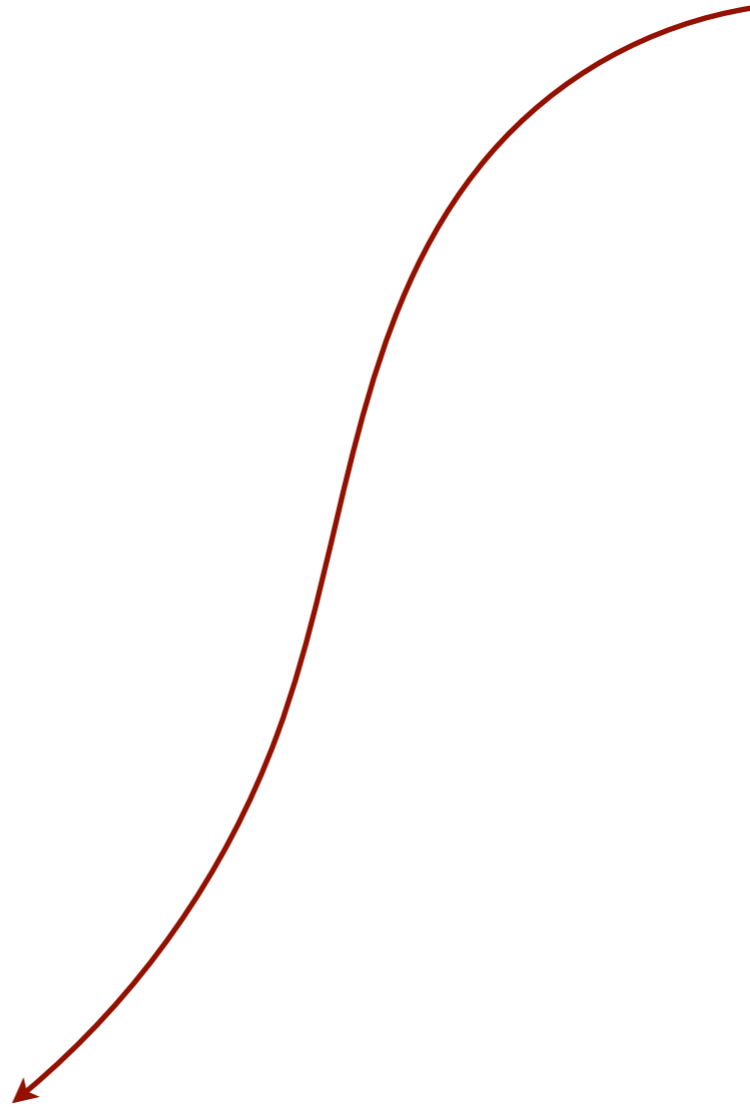
and logical properties

- not true = false
- not false = true
- true and true = true
- true and false = false
- true or true = true
- true or false = true

so we can build conditional statements.

- ✓ $(E_c E_1 E_2)$ — “If E_c then E_1 else E_2 ”

- Sequence
- Alternation
- Repetition



Lambda Calculus

Logic

We need logical operators

- not
- and
- or

$$\text{not} = (\lambda t . t \text{ false true})$$

Note: $(\lambda t . t \text{ false true})$ is one λ -expression with one parameter, t , and three elements in the “code” to the right of the dot, $t \text{ false true}$.

It's not $(\mathbf{I} \text{ false true})$, because $(\lambda t . t) \text{ false true} \neq (\lambda t . t \text{ false true})$.

Lambda Calculus

Logic

We need logical operators

- not
- and
- or

$$\text{not} = (\lambda t . t \text{ false true})$$

$$\begin{aligned} & \text{not true} \\ \equiv & (\lambda t . t \text{ false true}) \text{ true} \\ & [true/t] \text{ in } (t \text{ false true}) \\ \xrightarrow{\beta} & (\text{true false true}) \\ & \equiv (\mathbf{K} \text{ false true}) \\ \xrightarrow{\beta} & \text{false} \end{aligned}$$

$$\begin{aligned} & \text{not false} \\ \equiv & (\lambda t . t \text{ false true}) \text{ false} \\ & [false/t] \text{ in } (t \text{ false true}) \\ \xrightarrow{\beta} & (\text{false false true}) \\ & \equiv (\mathbf{K}' \text{ false true}) \\ \xrightarrow{\beta} & \text{true} \end{aligned}$$

Lambda Calculus

Logic

We need logical operators

- ✓ not
- and
- or

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

true and false

rewrite as

(and true false)

$$\equiv \lambda x y . ((x y) \text{false}) \text{true false}$$

[true/x] in $\lambda y . ((x y) \text{false})$

$$\xrightarrow{\beta} \lambda y . ((true y) \text{false}) \text{false}$$

[false/y] in $((true y) \text{false})$

$$\xrightarrow{\beta} ((true false) \text{false})$$

true false false

$$\equiv \mathbf{K} \text{false false}$$

$$\xrightarrow{\beta} \text{false}$$

true and true

rewrite as

(and true true)

$$\equiv \lambda x y . ((x y) \text{false}) \text{true true}$$

[true/x] in $\lambda y . ((x y) \text{false})$

$$\xrightarrow{\beta} \lambda y . ((true y) \text{false}) \text{true}$$

[true/y] in $((true y) \text{false})$

$$\xrightarrow{\beta} ((true true) \text{false})$$

true true false

$$\equiv \mathbf{K} \text{true false}$$

$$\xrightarrow{\beta} \text{true}$$

Lambda Calculus

Logic

We need logical operators

- ✓ not
- ✓ and
- or

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

true or false

rewrite as

(or true false)

$$\equiv \lambda x y . ((x \text{ true}) y) \text{ true false}$$

[true/x] in $\lambda y . ((x \text{ true}) y)$

$$\xrightarrow{\beta} \lambda y . ((true \text{ true}) y) \text{ false}$$

[false/y] in $((true \text{ true}) y)$

$$\xrightarrow{\beta} ((true \text{ true}) \text{ false})$$

true true false

$$\equiv \mathbf{K} \text{ true false}$$

$$\xrightarrow{\beta} \text{true}$$

false or false

rewrite as

(or false false)

$$\equiv \lambda x y . ((x \text{ true}) y) \text{ false false}$$

[false/x] in $\lambda y . ((x \text{ true}) y)$

$$\xrightarrow{\beta} \lambda y . ((false \text{ true}) y) \text{ false}$$

[false/y] in $((false \text{ true}) y)$

$$\xrightarrow{\beta} ((false \text{ true}) \text{ false})$$

false true false

$$\equiv \mathbf{K}' \text{ true false}$$

$$\xrightarrow{\beta} \text{false}$$

Lambda Calculus

Logic

We need logical values

- ✓ true
- ✓ false

and logical operators

- ✓ not
- ✓ and
- ✓ or

and logical properties

- ✓ not true = false
- ✓ not false = true
- ✓ true and true = true
- ✓ true and false = false
- ✓ true or true = true
- ✓ true or false = true

so we can build conditional statements.

- ✓ $(E_c E_1 E_2)$ — “If E_c then E_1 else E_2 ”

- Sequence
- Alternation
- Repetition

We're done here.

All we need now is repetition through recursion.

Lambda Calculus

Repetition Through Recursion

Consider a traditional recursive function:

```
function f(x)
begin
  if x <= 0
    return 1
  else
    return x * f(x-1)
  end if
end
```

Sequence

Alternation

→ Repetition

This function, f , calls itself. That's recursion. It can do that because it has a name. But what about anonymous functions? They are nameless. That's a problem.

Lambda Calculus

Repetition Through Recursion

In the 1940s and 1950s Haskell Curry worked on combinatorics and made several advances in the field. One of them was the Y combinator.



The Y combinator is a λ -calculus mechanism that allows a function to call itself. That's recursion. And that's what we need.

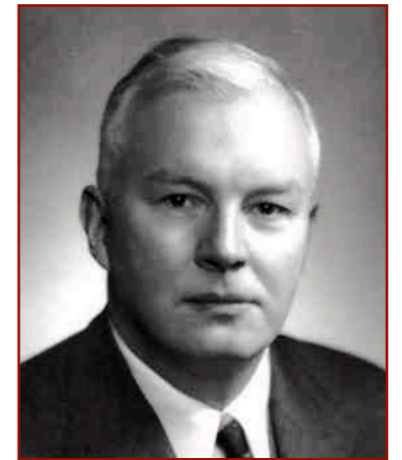
$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Where have you seen this before?

Lambda Calculus

Repetition Through Recursion

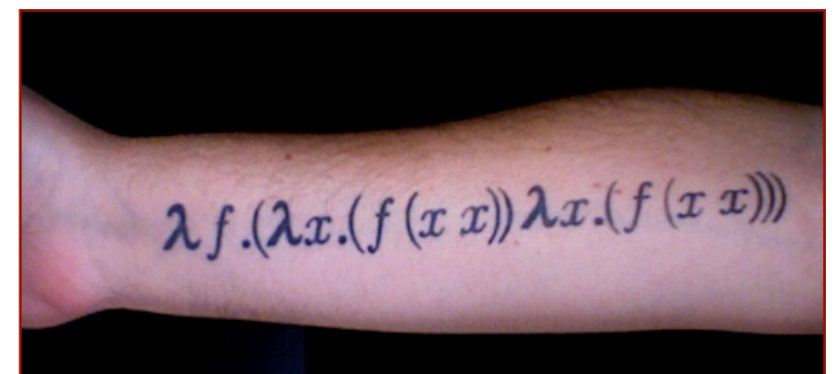
In the 1940s and 1950s Haskell Curry worked on combinatorics and made several advances in the field. One of them was the Y combinator.



The Y combinator is a λ -calculus mechanism that allows a function to call itself. That's recursion. And that's what we need.

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

Where have you seen this before?
On our web site.



Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

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

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

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

Let's try this by applying Y to some λ expression E.

$$\begin{aligned} & (Y E) \\ \equiv & (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) E \\ & [E/f] \text{ in } (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \xrightarrow{\beta} & (\lambda x. E (x x)) (\lambda x. E (x x)) \end{aligned}$$

We're not done reducing, but let's take a note of this partially evaluated expression:

$$(Y E) \xrightarrow{\beta} (\lambda x. E (x x)) (\lambda x. E (x x))$$

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

$$\begin{aligned} Y &= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \\ &\quad (Y E) \\ &\equiv (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) E \\ &\quad [E/f] \text{ in } (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \xrightarrow{\beta} & \boxed{(\lambda x. E (x x)) (\lambda x. E (x x))} \longleftarrow (Y E) \end{aligned}$$

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

$$\begin{aligned} Y &= \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \\ &\quad (Y E) \\ &\equiv (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) E \\ &\quad [E/f] \text{ in } (\lambda x. f (x x)) (\lambda x. f (x x)) \\ &\xrightarrow{\beta} \boxed{(\lambda x. E (x x)) (\lambda x. E (x x))} \longleftarrow (Y E) \end{aligned}$$

Note that these two λ expressions are closed terms (contain no free variables) and both use a (bound) variable “x”.

It will be easier to keep things straight if we α -convert one of them.

$$\equiv_{\alpha} (\lambda x. E (x x)) (\lambda y. E (y y))$$

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

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

$$\begin{aligned} & (Y E) \\ \equiv & (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) E \\ & [E/f] \text{ in } (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \xrightarrow{\beta} & (\lambda x. E (x x)) (\lambda x. E (x x)) \longleftarrow (Y E) \\ \equiv_{\alpha} & (\lambda x. E (x x)) (\lambda y. E (y y)) \\ & [(\lambda y. E (y y))/x] \text{ in } E (x x) \\ \xrightarrow{\beta} & E ((\lambda y. E (y y)) (\lambda y. E (y y))) \end{aligned}$$

We can α -convert again, for clarity.

$$\equiv_{\alpha} E ((\lambda x. E (x x)) (\lambda x. E (x x)))$$

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

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

$$\begin{aligned} & (Y E) \\ \equiv & (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) E \\ & [E/f] \text{ in } (\lambda x. f (x x)) (\lambda x. f (x x)) \\ \xrightarrow{\beta} & (\lambda x. E (x x)) (\lambda x. E (x x)) \leftarrow (Y E) \\ \equiv_{\alpha} & (\lambda x. E (x x)) (\lambda y. E (y y)) \\ & [(\lambda y. E (y y))/x] \text{ in } E (x x) \\ \xrightarrow{\beta} & E ((\lambda y. E (y y)) (\lambda y. E (y y))) \\ \equiv_{\alpha} & E ((\lambda x. E (x x)) (\lambda x. E (x x))) \\ & \equiv E (Y E) \end{aligned}$$

Lambda Calculus

Repetition Through Recursion

The Y combinator is a λ -calculus mechanism that allows a function to call itself recursively.

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

Y, applied to a λ expression E, beta-reduces to E (Y E)

$$\begin{array}{c} (Y E) \\ \xrightarrow{\beta} E (Y E) \end{array}$$

In other words, it recursively calls E with itself as a parameter.

Repeated applications result in a stack of recursive calls:

$$Y E \xrightarrow{\beta} E(Y E) \xrightarrow{\beta} E(E(Y E)) \xrightarrow{\beta} E(E(E(Y E))) \cdot \cdot \cdot$$

Lambda Calculus

Complete and Universal

With the Y combinator allowing functions to call themselves recursively, we now have sequence, alternation, and repetition in the Lambda Calculus. It is a complete and universal model of computation.

- ☑ Sequence
- ☑ Alternation
- ☑ Repetition

From Leibniz, through Church and Turing, all the way to Curry and the functional languages of today, the Lambda Calculus is all we need.

