Lambda?

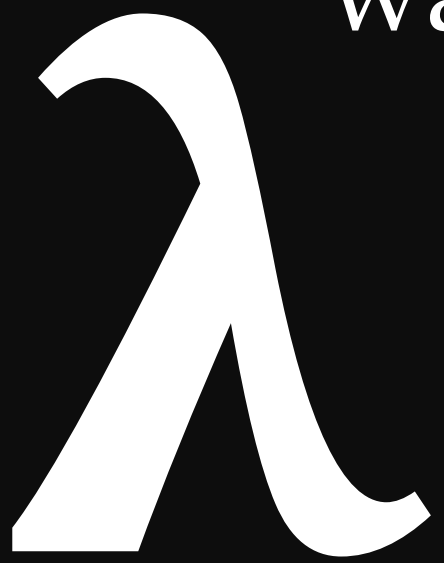
You Keep Using that Letter

@KevlinHenney

AWS

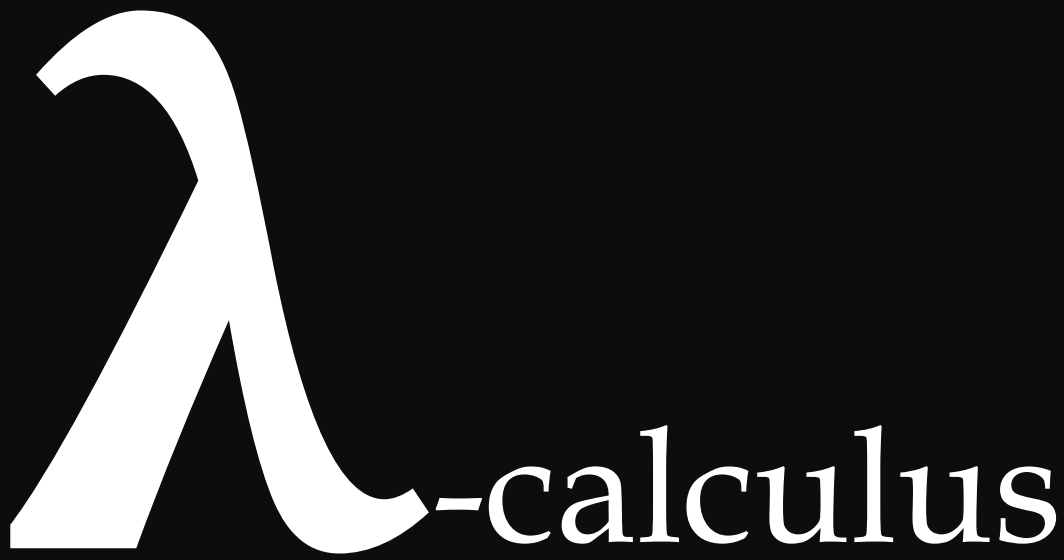
wavelength

Half-Life



calculus

decay constant


λ-calculus

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.[1]

By Alonzo Church.[2]

1. **Introduction.** In this paper we present a set of postulates for the foundation of formal logic, in which we avoid use of the free, or real, variable, and in which we introduce a certain restriction on the law of excluded middle as a means of avoiding the paradoxes connected with the mathematics of the transfinite.

Our reason for avoiding use of the free variable is that we require that every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations. That the use of the free variable involves violation of this requirement, we believe is readily seen. For example, the identity

(1)

We do not attach any character of uniqueness or absolute truth to any particular system of logic.

The entities of formal logic are abstractions, invented because of their use in describing and systematizing facts of experience or observation, and their properties, determined in rough outline by this intended use, depend for their exact character on the arbitrary choice of the inventor.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

In 1911 Russell & Whitehead published Principia Mathematica, with the goal of providing a solid foundation for all of mathematics. In 1931 Gödel's Incompleteness Theorem shattered the dream, showing that for any consistent axiomatic system there will always be theorems that cannot be proven within the system.

Adrian Colyer

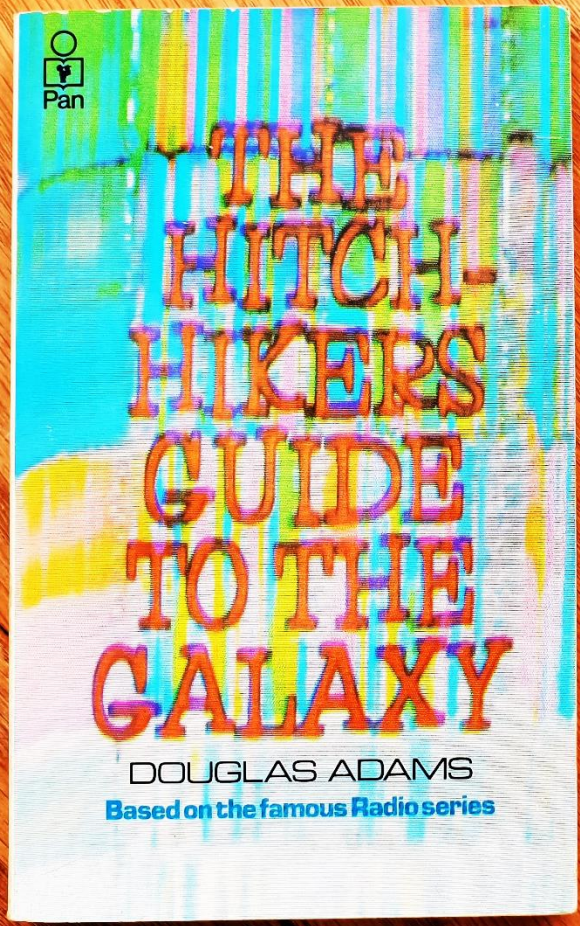
https://blog.acolyer.org/2020/02/03/measure-mismeasure-fairness/

One premise of many models of fairness in machine learning is that you can measure (‘prove’) fairness of a machine learning model from within the system – i.e. from properties of the model itself and perhaps the data it is trained on.

To show that a machine learning model is fair, you need information from outside of the system.

Adrian Colyer

https://blog.acolyer.org/2020/02/03/measure-mismeasure-fairness/


Pan
THE HITCH-HIKERS GUIDE TO THE GALAXY
DOUGLAS ADAMS
Based on the famous Radio series

We demand rigidly defined areas of doubt and uncertainty!

LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

Despite the fancy name, a lambda is just a function... peculiarly... without a name.

https://rubymonk.com/learning/books/1-ruby-primer/chapters/34-lambdas-and-blocks-in-ruby/lessons/77-lambdas-in-ruby

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

We select a particular list of symbols, consisting of the symbols { , }, (,), λ, [,], and an enumerably infinite set of symbols *a*, *b*, *c*, ⋮ to be called *variables*.

And we define the word *formula* to mean any finite sequence of symbols out of this list.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

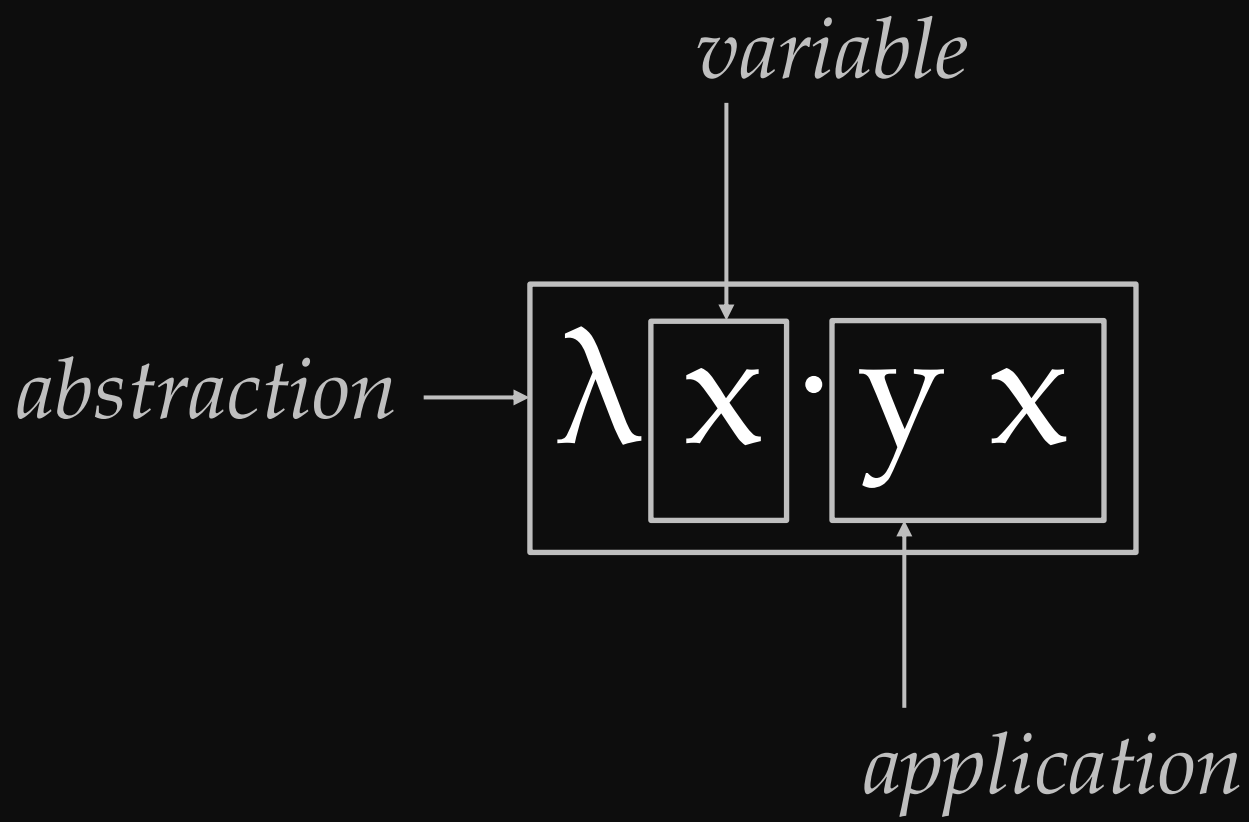
By ALONZO CHURCH.

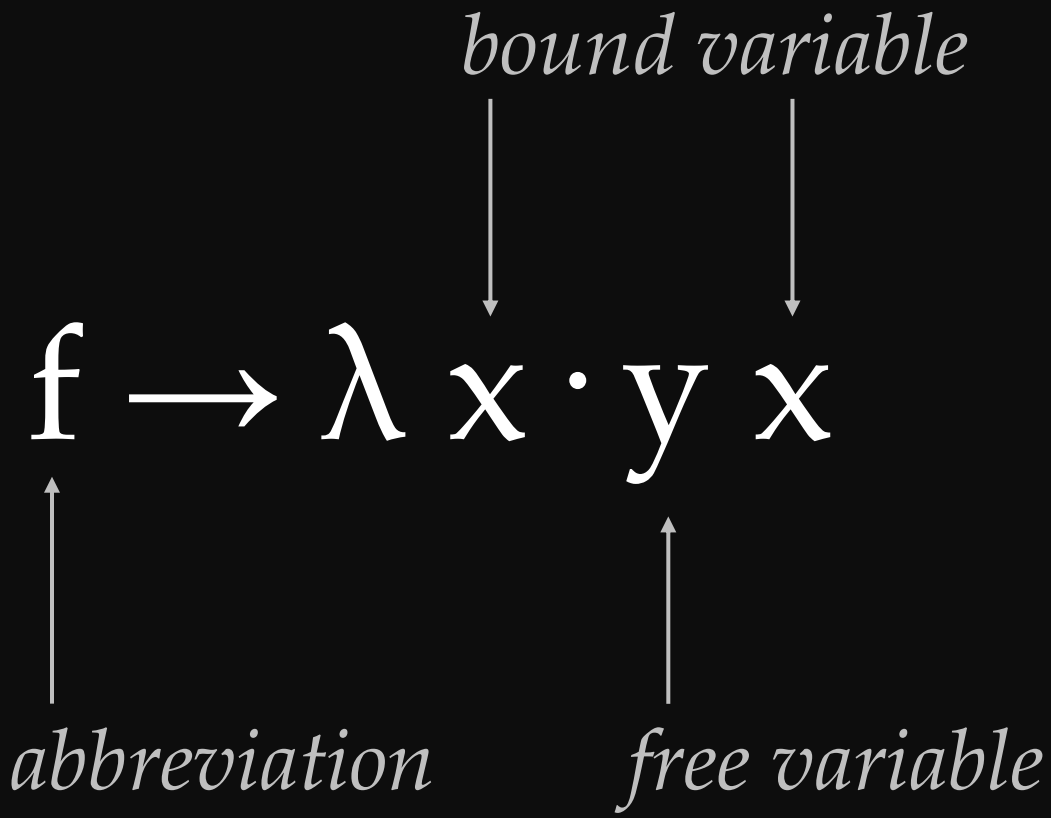
1. **Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

$$f(x) = formula$$

$$f \rightarrow \lambda x \cdot formula$$


variable
abstraction
λ x · y x
application


bound variable
f → λ x · y x
abbreviation
free variable

$$\text{square}(x) = x \times x$$

$$\text{square} \rightarrow \lambda\, x \cdot x \times x$$


square → λ😠 · 😠 × 😠

□ 7

square 7

$$(\lambda\, x \cdot x \times x)\ 7$$

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

AN UNSOLVEABLE PROBLEM OF ELEMENTARY NUMBER THEORY

By Alonzo Church.

1. **Introduction.** There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers such that $f(x_1, x_2, \cdots, x_n) = 2$ is necessary and sufficient for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z such that $x^n + y^n = z^n$. For this may be interpreted as required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

NUMBER

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By ALONZO CHURCH.

---

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly

0

$$0 \rightarrow \lambda\, f \cdot \lambda\, x \cdot x$$
$$1 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(x)$$
$$2 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(x))$$
$$3 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(x)))$$
$$4 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(x))))$$
$$5 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(f(x)))))$$
$$6 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(f(f(x))))))$$

$$0 \rightarrow \lambda\, f\, x \cdot x$$
$$1 \rightarrow \lambda\, f\, x \cdot f(x)$$
$$2 \rightarrow \lambda\, f\, x \cdot f(f(x))$$
$$3 \rightarrow \lambda\, f\, x \cdot f(f(f(x)))$$
$$4 \rightarrow \lambda\, f\, x \cdot f(f(f(f(x))))$$
$$5 \rightarrow \lambda\, f\, x \cdot f(f(f(f(f(x)))))$$
$$6 \rightarrow \lambda\, f\, x \cdot f(f(f(f(f(f(x))))))$$

$$0 \to \lambda\, f\, x \cdot x$$
$$1 \to \lambda\, f\, x \cdot f\, x$$
$$2 \to \lambda\, f\, x \cdot f^2\, x$$
$$3 \to \lambda\, f\, x \cdot f^3\, x$$
$$4 \to \lambda\, f\, x \cdot f^4\, x$$
$$5 \to \lambda\, f\, x \cdot f^5\, x$$
$$6 \to \lambda\, f\, x \cdot f^6\, x$$

$$0 \to \lambda\, f\, x \cdot f^{0}\, x$$
$$1 \to \lambda\, f\, x \cdot f^{1}\, x$$
$$2 \to \lambda\, f\, x \cdot f^{2}\, x$$
$$3 \to \lambda\, f\, x \cdot f^{3}\, x$$
$$4 \to \lambda\, f\, x \cdot f^{4}\, x$$
$$5 \to \lambda\, f\, x \cdot f^{5}\, x$$
$$6 \to \lambda\, f\, x \cdot f^{6}\, x$$

7

```
7.times
```

```ruby
7.times {|i| puts i}
```

0
1
2
3
4
5
6

$$0 \rightarrow \lambda\, f\, x \cdot x$$
$$1 \rightarrow \text{succ } 0$$
$$2 \rightarrow \text{succ succ } 0$$
$$3 \rightarrow \text{succ succ succ } 0$$
$$4 \rightarrow \text{succ succ succ succ } 0$$
$$5 \rightarrow \text{succ succ succ succ succ } 0$$
$$6 \rightarrow \text{succ succ succ succ succ s}$$

$$
\begin{aligned}
0 &\to \lambda\, f\, x \cdot x \\
1 &\to \mathrm{succ}^1\, 0 \\
2 &\to \mathrm{succ}^2\, 0 \\
3 &\to \mathrm{succ}^3\, 0 \\
4 &\to \mathrm{succ}^4\, 0 \\
5 &\to \mathrm{succ}^5\, 0 \\
6 &\to \mathrm{succ}^6\, 0
\end{aligned}
$$

$$0 \to \lambda\, f\, x \cdot x$$
$$1 \to \text{succ}\ 0$$
$$2 \to \text{succ}\ 1$$
$$3 \to \text{succ}\ 2$$
$$4 \to \text{succ}\ 3$$
$$5 \to \text{succ}\ 4$$
$$6 \to \text{succ}\ 5$$

$$\text{succ} \rightarrow \lambda\, n\, f\, x \cdot f\, (n\, f\, x)$$

You may have heard of lambdas before. Perhaps you've used them in other languages.

https://rubymonk.com/learning/books/1-ruby-primer/chapters/34-lambdas-and-blocks-in-ruby/lessons/77-lambdas-in-ruby

```
auto square(auto x)
{
    return x * x;
}
```

```
auto square = [](auto x)
{
    return x * x;
};
```

square(7)

```
[](auto x)
{
    return x * x;
}(7)
```

```
[](auto x) {return x * x;}(7)
```

They’re anonymous, little functional spies sneaking into the rest of your code.

https://rubymonk.com/learning/books/1-ruby-primer/chapters/34-lambdas-and-blocks-in-ruby/lessons/77-lambdas-in-ruby

Excel is the world's most popular functional language

Simon Peyton-Jones

LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

```
(lambda (x) (* x x))
```

```
((lambda (x) (* x x)) 7)
```

Revised Report on the Algorithmic Language

Algol 68

Edited by
A. van Wijngaarden, B. J. Mailloux,
J. E. L. Peck, C. H. A. Koster, M. Sintzoff

```
(int x) int: x * x
```

```
proc (int) int square;
square := (int x) int: x * x;
int result := square (7);
```

```
((int x) int: x * x) (7)
```

Lambdas in Ruby are also objects, just like everything else!

https://rubymonk.com/learning/books/1-ruby-primer/chapters/34-lambdas-and-blocks-in-ruby/lessons/77-lambdas-in-ruby

Lambdas in C++ are also objects!

```
[](auto x) {return x(y);}
```

```
struct __lambda
{
   auto operator()(auto x)
   {
      return x(y);
   }
};
```

Lambdas in C++ are function objects!

Lambdas in C++ are not functors!

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing — is this true?"

Qc Na looked pityingly at his student and replied, "Foolish pupil — objects are merely a poor man's closures."

http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

The concept of closures was developed in the 1960s for the mechanical evaluation of expressions in the λ-calculus.

Peter J. Landin defined the term *closure* in 1964 as having an *environment part* and a *control part*.

https://en.wikipedia.org/wiki/Closure_(computer_programming)

Joel Moses credits Landin with introducing the term *closure* to refer to a lambda expression whose open bindings (free variables) have been closed by (or bound in) the lexical environment, resulting in a *closed expression*, or *closure*.

https://en.wikipedia.org/wiki/Closure_(computer_programming)

This usage was subsequently adopted by Sussman and Steele when they defined Scheme in 1975, a lexically scoped variant of LISP, and became widespread.

https://en.wikipedia.org/wiki/Closure_(computer_programming)

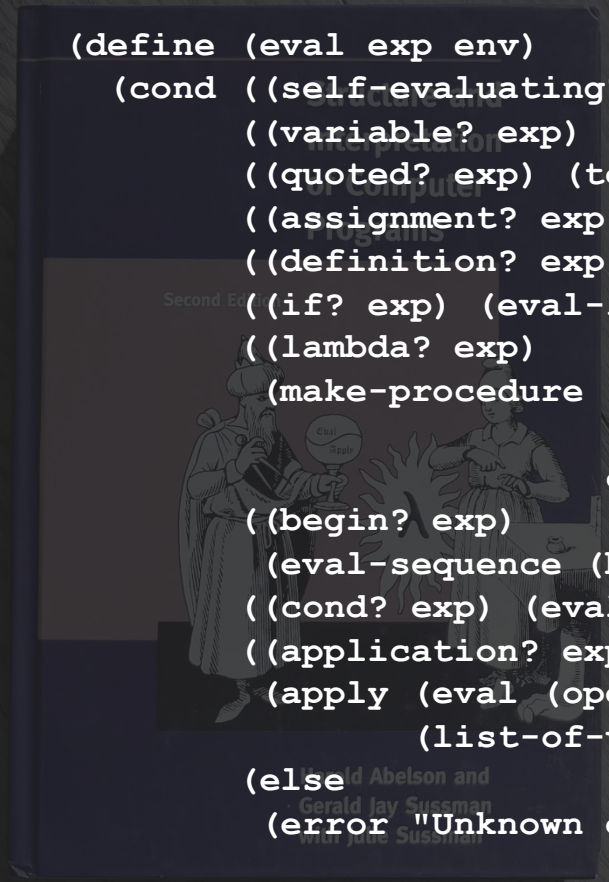
Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system.

http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

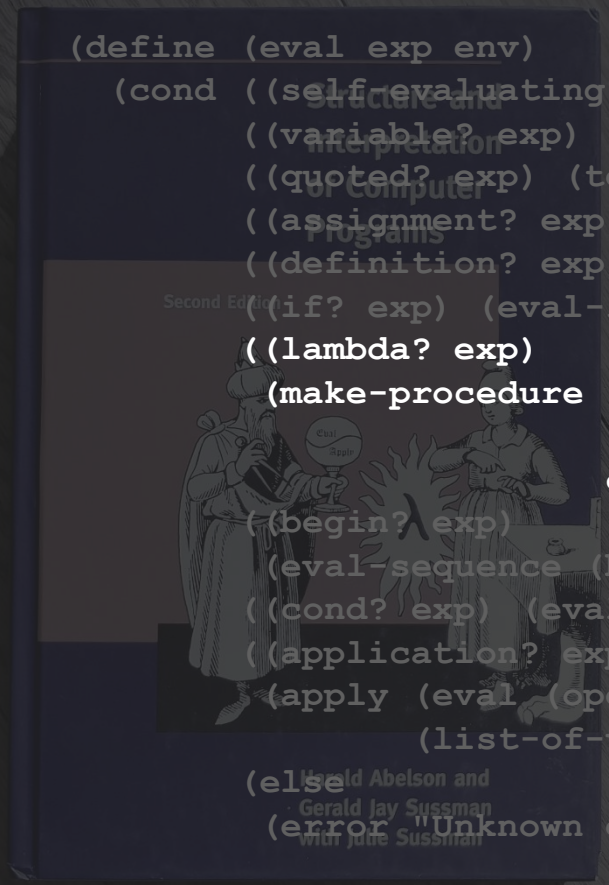
Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman



```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```




```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

This work developed out of an initial attempt to understand the actorness of actors.

This interpreter attempted to intermix the use of actors and LISP lambda expressions in a clean manner.

"Scheme: An Interpreter for Extended Lambda Calculus"
Gerald Jay Sussman & Guy L Steele Jr

When it was completed, we discovered that the “actors” and the lambda expressions were identical in implementation.

“Scheme: An Interpreter for Extended Lambda Calculus”
Gerald Jay Sussman & Guy L Steele Jr

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((alpha? exp)
         (make-procedure (alpha-parameters exp)
                         (alpha-body exp)
                         env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((alpha? exp)
         (make-procedure (alpha-parameters exp)
                         (alpha-body exp)
                         env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures."

Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object."

http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

At that moment, Anton became enlightened.

http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

On Understanding Data Abstraction, Revisited

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form.

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

λ-calculus was the first object-oriented language.

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming. Despite over 25 years of research there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language

So what's the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form.

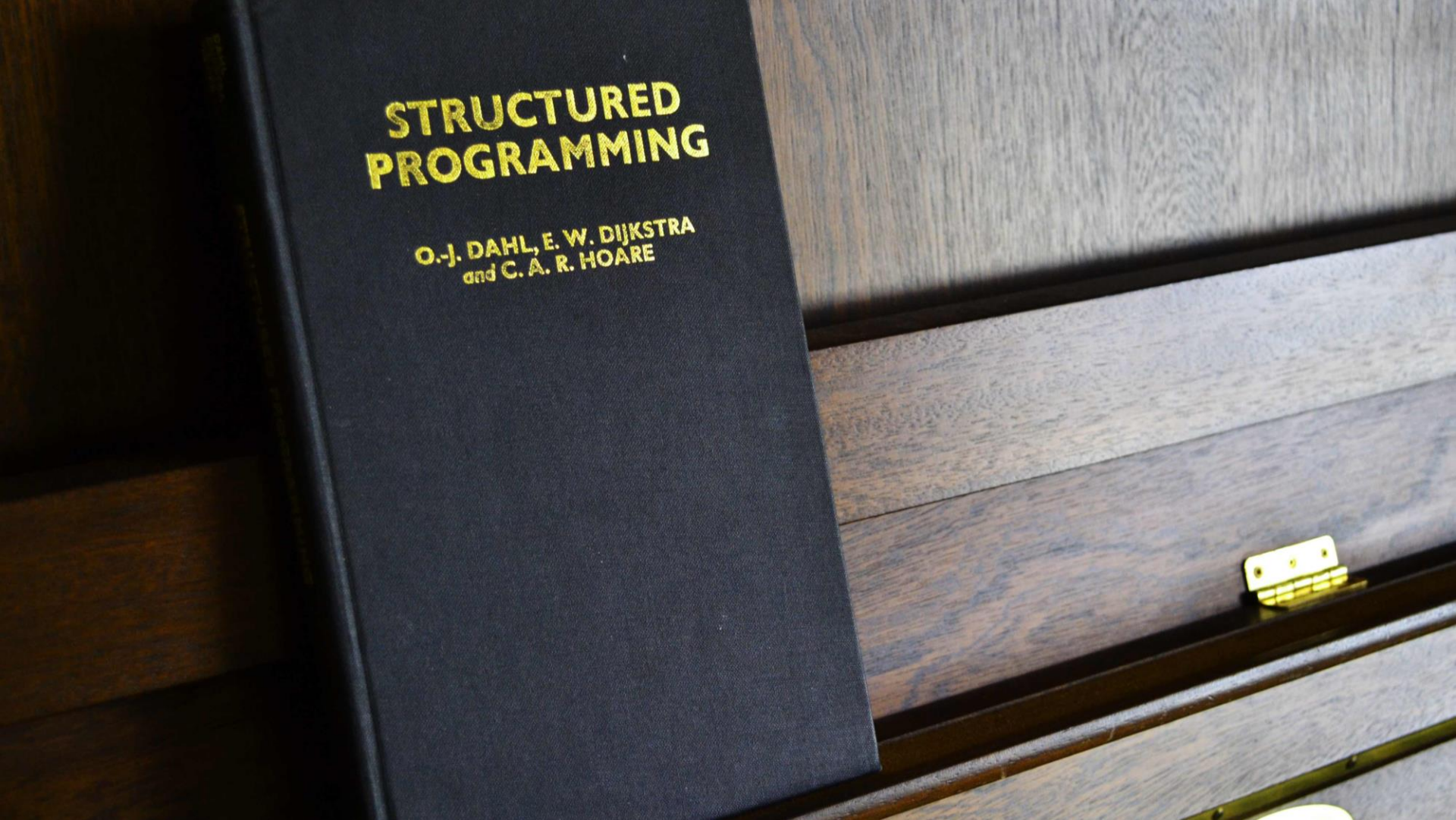
```
stack<std::string> words;

assert(words.depth() == 0);
assert(words.top() == std::nullopt);

words = words.push("C");
words = words.push("C++");

assert(words.depth() == 2);
assert(words.top() == "C++");

words = words.pop();

assert(words.top() == "C");
```

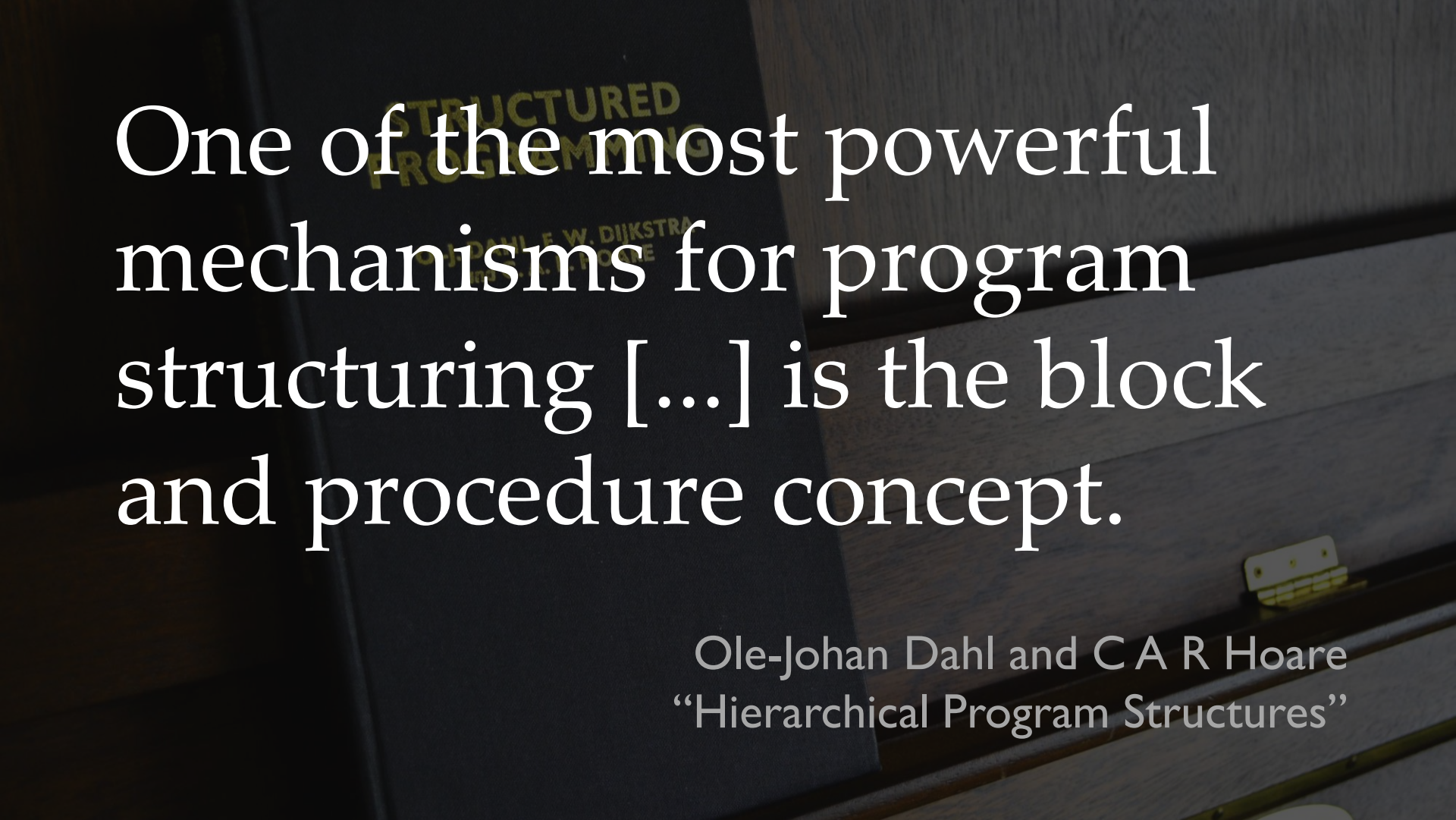
```
template<typename T>
struct stack
{
    stack();
    stack(const T & head, const stack & tail);

    std::function<std::size_t()> depth;
    std::function<std::optional<T>()> top;
    std::function<stack()> pop;
    std::function<stack(const T &)> push;
};
```

```
stack() :
    depth(
        []
            { return 0; }),
    top(
        []
            { return std::nullopt; }),
    pop(
        []
            { return stack(); }),
    push(
        [] (const auto & new_top)
            { return stack(new_top, stack()); })
{
}
```

```
stack(const T & head, const stack & tail) :
    depth(
        [=]
            { return 1 + tail.depth(); }),
    top(
        [=]
            { return head; }),
    pop(
        [=]
            { return tail; }),
    push(
        [=] (const auto & new_top)
            { return stack(new_top, tail.push(head)); })
{
}
```

```
stack(const T & head, const stack & tail) :
    depth(
        [size = 1 + tail.depth()]
            { return size; }),
    top(
        [head]
            { return head; }),
    pop(
        [tail]
            { return tail; }),
    push(
        [head, tail] (const auto & new_top)
            { return stack(new_top, tail.push(head)); })
{
}
```

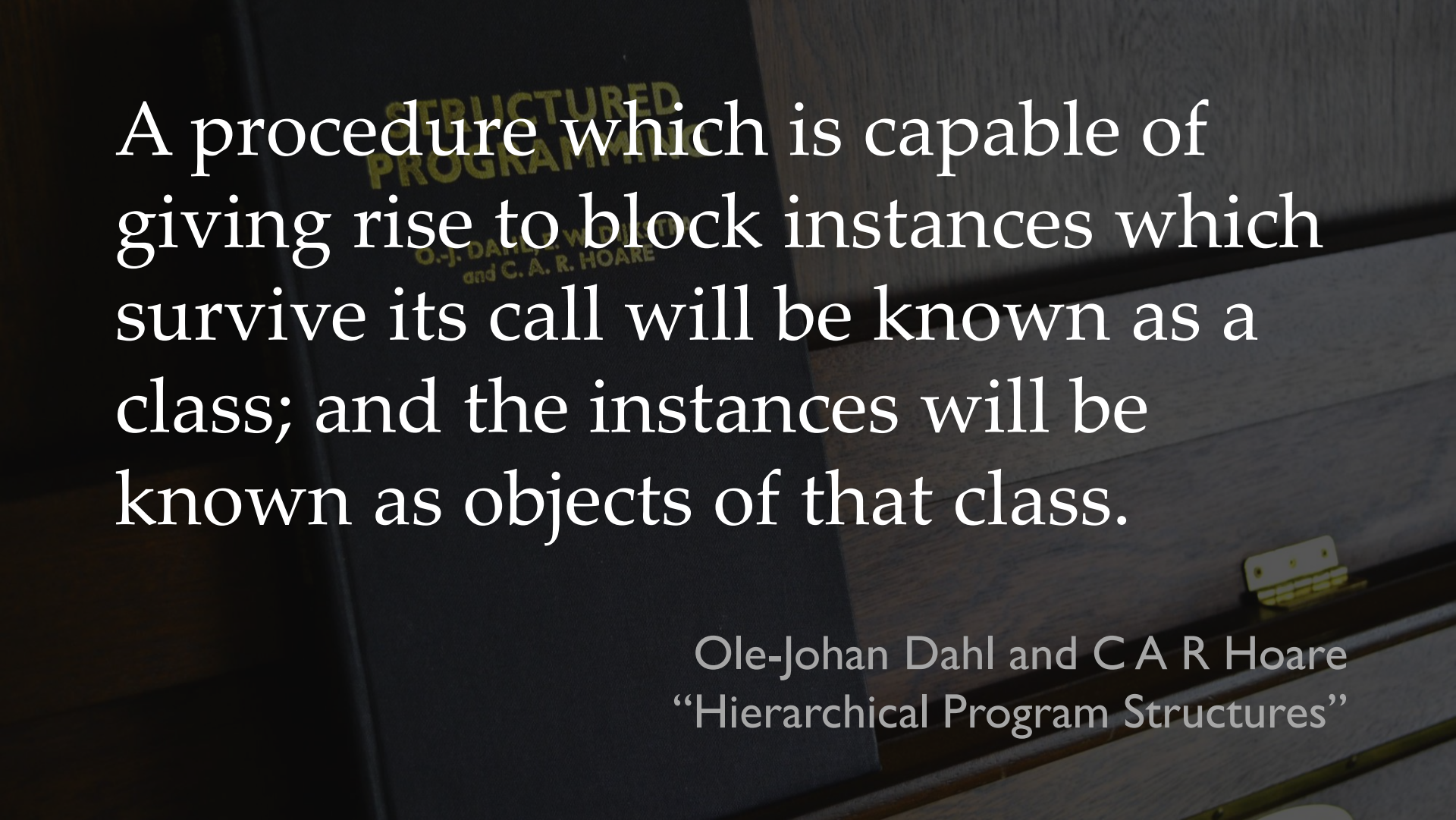

STRUCTURED
PROGRAMMING
O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE



One of the most powerful mechanisms for program structuring [...] is the block and procedure concept.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
begin
    ref(Rock) array items(1:capacity);
    integer count;
    integer procedure Depth; ...
    ref(Rock) procedure Top; ...
    procedure Push(top); ...
    procedure Pop; ...
    count := 0
end;
```



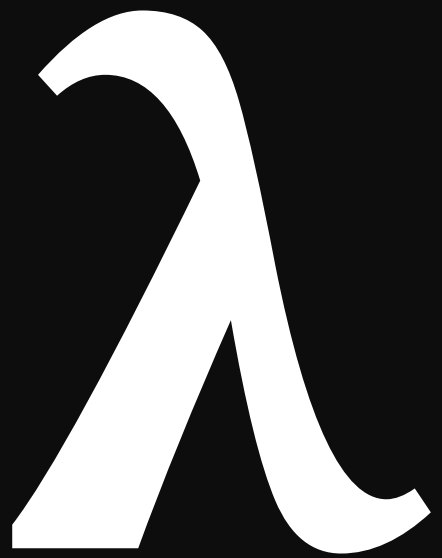
A procedure which is capable of giving rise to block instances which survive its call will be known as a class; and the instances will be known as objects of that class.

Ole-Johan Dahl and C A R Hoare
"Hierarchical Program Structures"

```
class Stack(capacity);
    integer capacity;
begin
    ref(Rock) array items(1:capacity);
    integer count;
    integer procedure Depth; ...
    ref(Rock) procedure Top; ...
    procedure Push(top); ...
    procedure Pop; ...
    count := 0
end;
```

We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful.
In fact, mathematics is, to a large extent, invention of better notations.

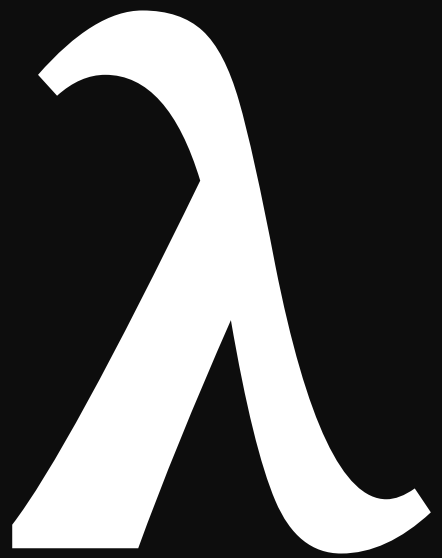
Richard Feynman




λa

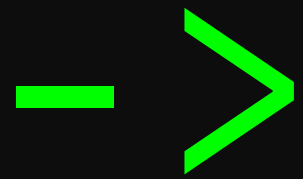

Aa

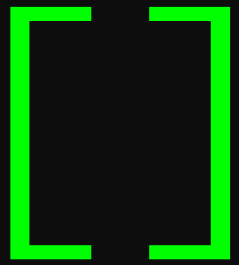

λa

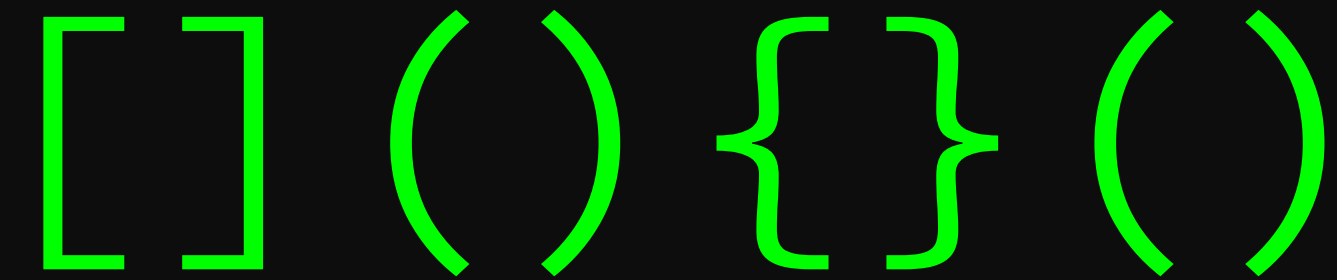



lambda

function

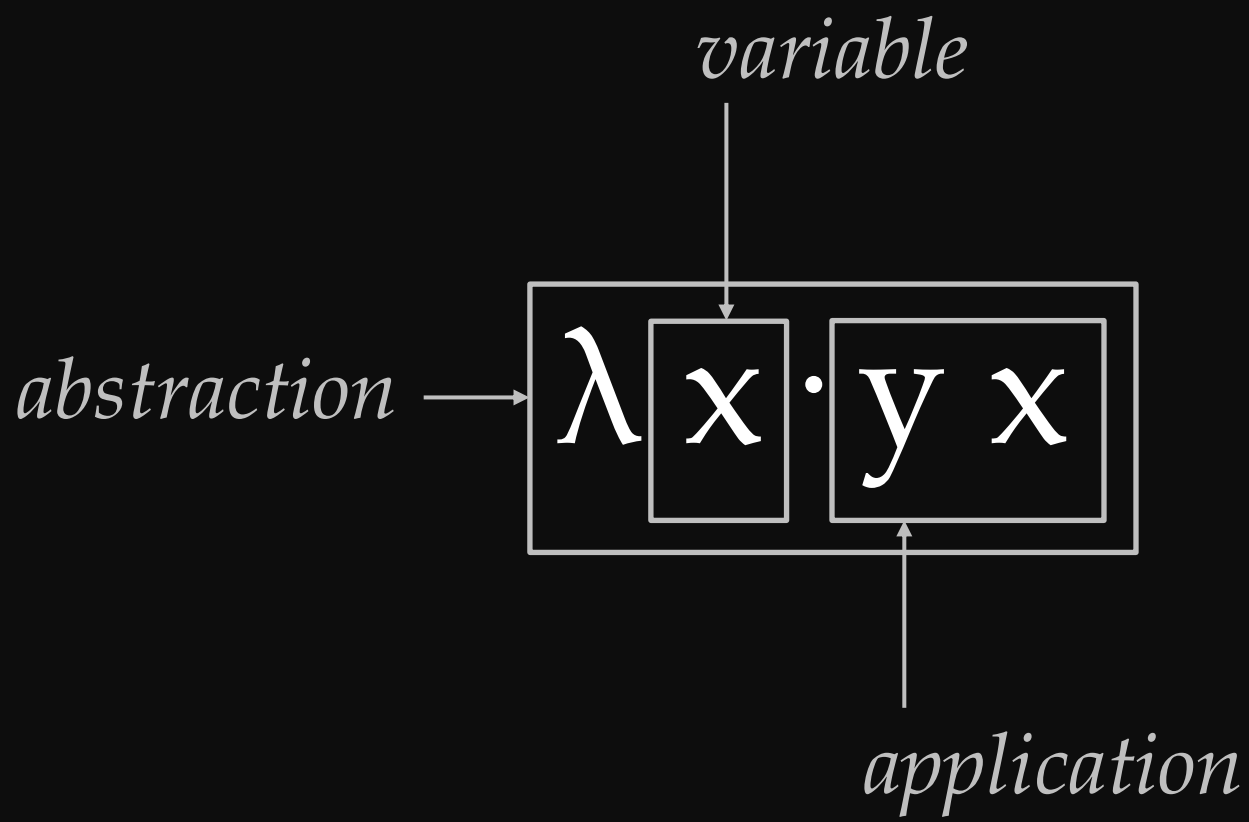


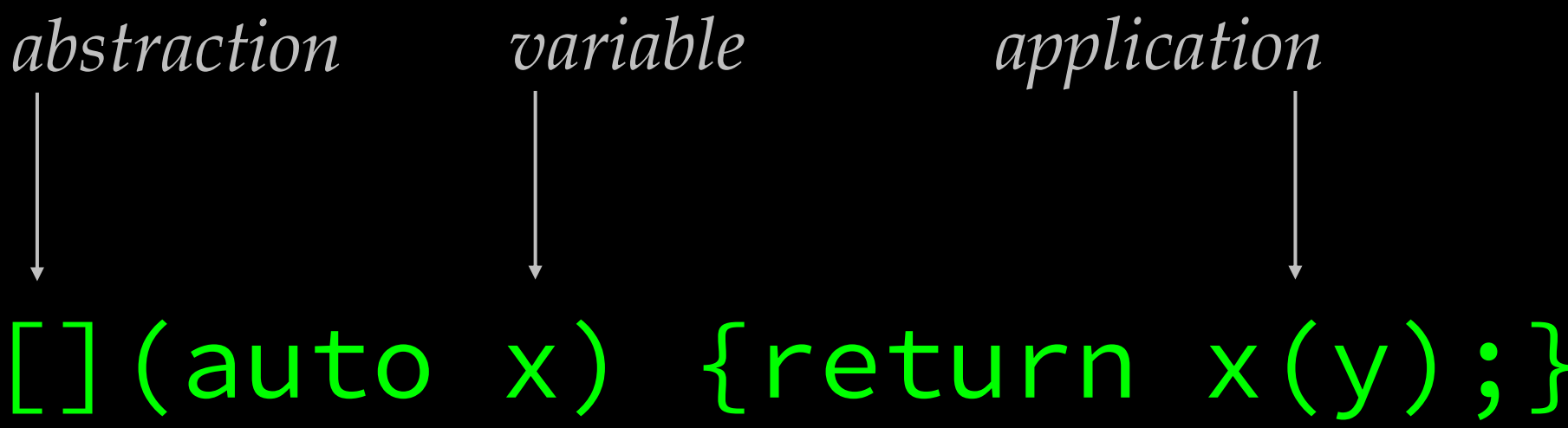




```
[[]](+[]+())[[]]{}={})()
```

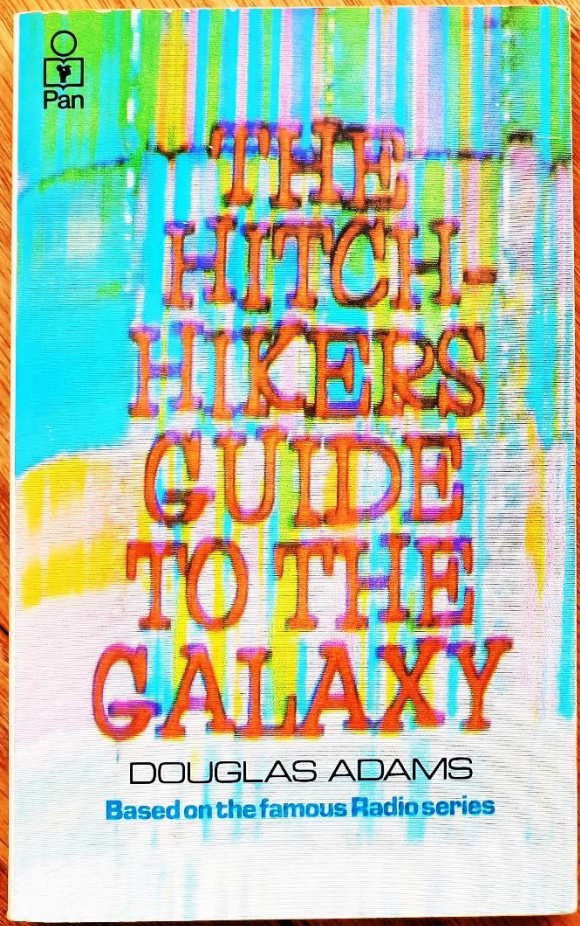
https://twitter.com/shafikyaghmour/status/1128790426519871496

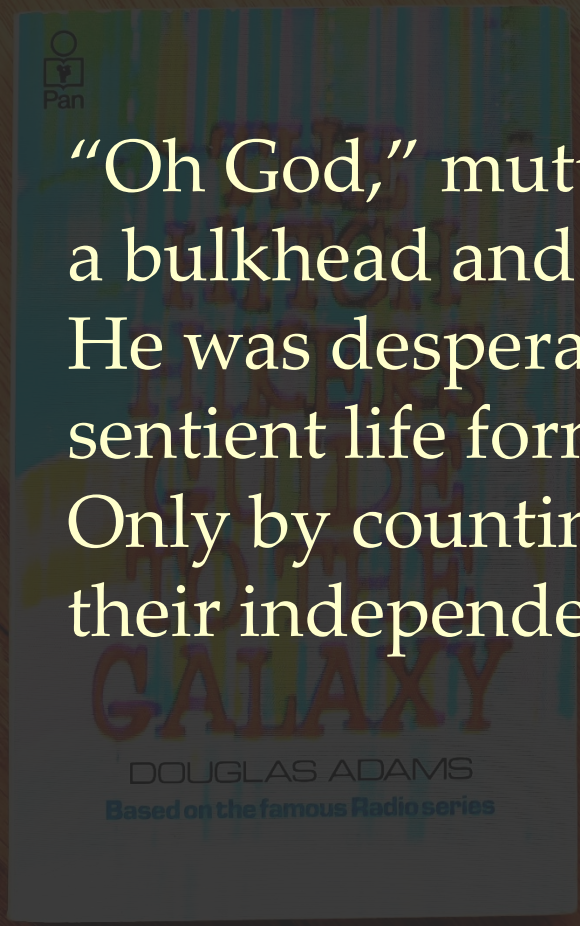

variable
abstraction
λ x · y x
application


abstraction
variable
application
[](auto x) {return x(y);}

```
[](auto x) {x(y)}
```

#wishlist


Pan
THE HITCH-HIKERS GUIDE TO THE GALAXY
DOUGLAS ADAMS
Based on the famous Radio series



"Oh God," muttered Ford, slumped against a bulkhead and started to count to ten. He was desperately worried that one day sentient life forms would forget how to do this. Only by counting could humans demonstrate their independence of computers.

$$0 \rightarrow \lambda\, f \cdot \lambda\, x \cdot x$$
$$1 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(x)$$
$$2 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(x))$$
$$3 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(x)))$$
$$4 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(x))))$$
$$5 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(f(x)))))$$
$$6 \rightarrow \lambda\, f \cdot \lambda\, x \cdot f(f(f(f(f(f(x))))))$$

```
0 = f => x => x
1 = f => x => f(x)
2 = f => x => f(f(x))
3 = f => x => f(f(f(x)))
4 = f => x => f(f(f(f(x))))
5 = f => x => f(f(f(f(f(x)))
6 = f => x => f(f(f(f(f(f(x
```

```
_0 = f => x => x
_1 = succ(_0)
_2 = succ(_1)
_3 = succ(_2)
_4 = succ(_3)
_5 = succ(_4)
_6 = succ(_5)
```

```
succ = n => f => x => f(n(f)(x))
```

```
auto succ =
    [](auto n)
    {
        return [=](auto f)
        {
            return [=](auto x)
            {
                return f(n(f)(x));
            };
        };
    };
```

```
auto _0 =
    [](auto f)
    {
        return [=](auto x)
        {
            return x;
        };
    };
```

```
auto _0 = ...;
auto _1 = succ(_0);
auto _2 = succ(_1);
auto _3 = succ(_2);
auto _4 = succ(_3);
auto _5 = succ(_4);
auto _6 = succ(_5);
```

```
auto plus_1 = [](auto n)
{
    return n + 1;
};
```

```
_0(plus_1)
```

```
_0(plus_1)(0)                    0
__1(plus_1)(0)                   1
__2(plus_1)(0)                   2
__3(plus_1)(0)                   3
__4(plus_1)(0)                   4
__5(plus_1)(0)                   5
__6(plus_1)(0)                   6
```

```
auto plus_1 = [](auto n)
{
    return n + lexical_cast<decltype(n)>(1);
};
```

```
_0(plus_1)(0)                    0
__1(plus_1)(0)                   1
__2(plus_1)(0)                   2
__3(plus_1)(0)                   3
__4(plus_1)(0)                   4
__5(plus_1)(0)                   5
__6(plus_1)(0)                   6
```

```
_0(plus_1)(""s)
_1(plus_1)(""s)                                                  1
_2(plus_1)(""s)                                                 11
_3(plus_1)(""s)                                                111
_4(plus_1)(""s)                                               1111
_5(plus_1)(""s)                                              11111
_6(plus_1)(""s)                                             111111
```

square

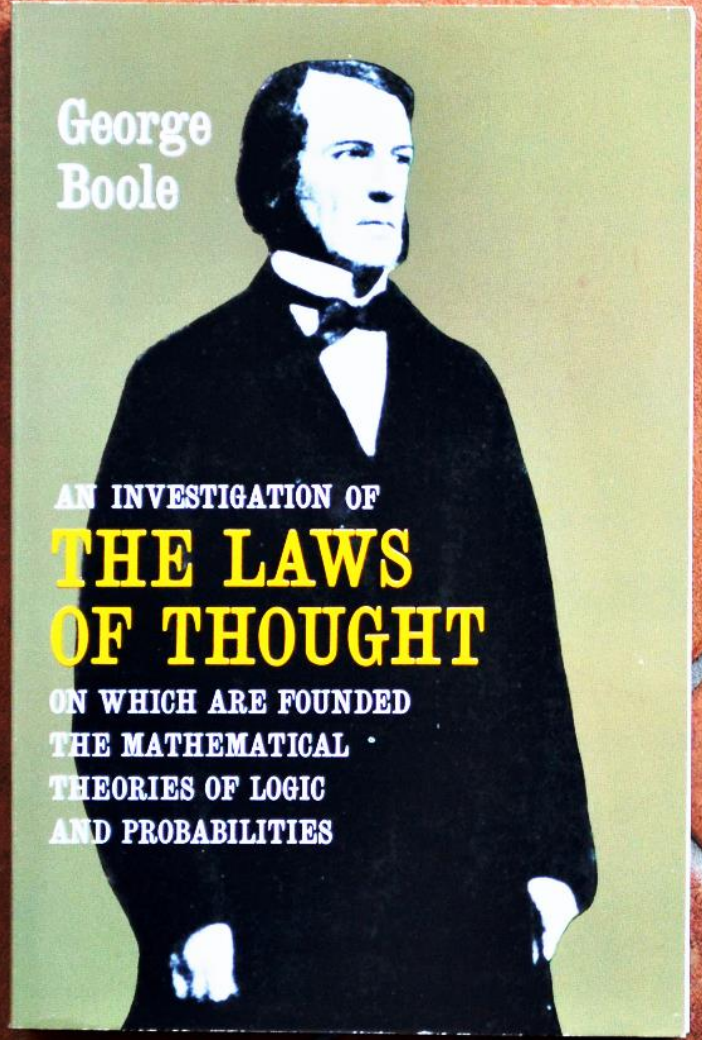
```
auto square = [](auto m)
{
    return _2(m);
};
```

```
square(_7)
```

```
square(_7)(plus_1)
```

```
square(_7)(plus_1)(0)
```

49


George
Boole
AN INVESTIGATION OF
THE LAWS
OF THOUGHT
ON WHICH ARE FOUNDED
THE MATHEMATICAL
THEORIES OF LOGIC
AND PROBABILITIES

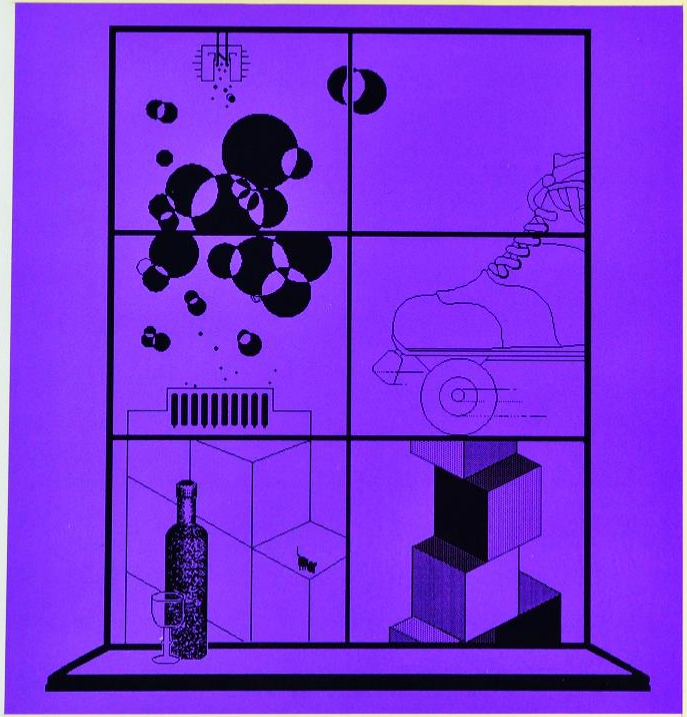
true
false

$$\text{true} \rightarrow \lambda\, a\, b \cdot a$$

$$\text{false} \rightarrow \lambda\, a\, b \cdot b$$

SMALLTALK–80

THE LANGUAGE



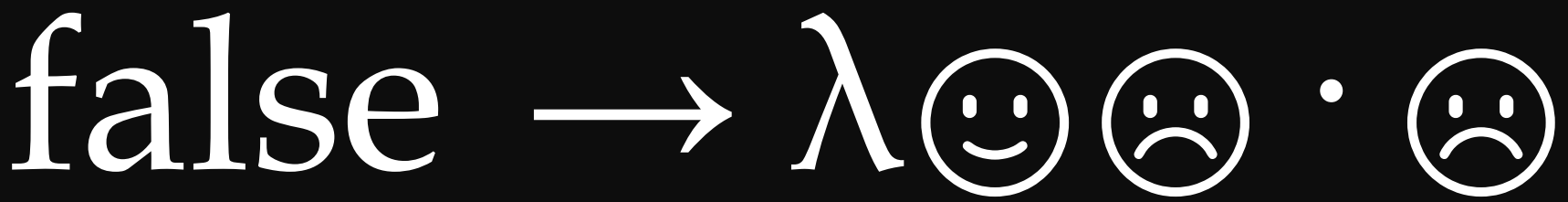
Adele Goldberg and David Robson

```
7 * 7 < limit
   ifTrue:  [^ 'OK']
   ifFalse: [^ 'Oh dear']
```

```
True
  ifTrue: toDo ifFalse: ignore
    ^ toDo value
```

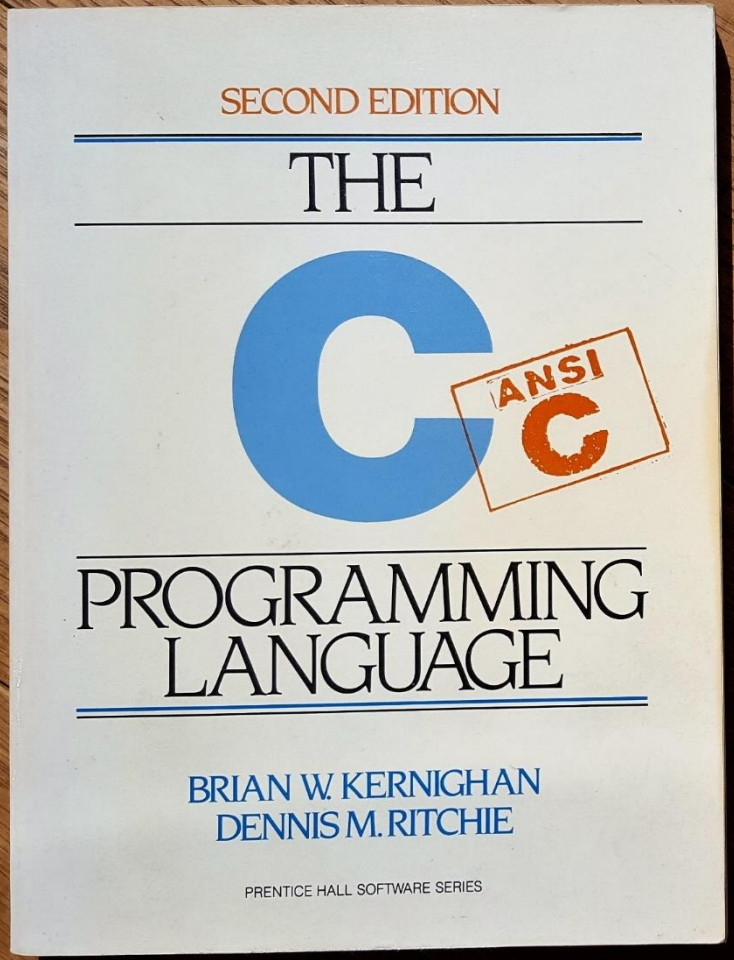
```
False
  ifTrue: ignore ifFalse: toDo
    ^ toDo value
```

$$\mathrm{false} \rightarrow \lambda\, a\, b \cdot b$$


false → λ ☺ ☹ · ☹

$$\mathrm{false} \rightarrow \lambda\, \mathrm{f}\, \mathrm{x} \cdot \mathrm{x}$$

false = 0


SECOND EDITION
THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

$$
\begin{aligned}
\text{pair} &\to \lambda\, x\, y\, f \cdot f\, x\, y \\
\text{first} &\to \lambda\, p \cdot p\, \lambda\, x\, y \cdot x \\
\text{second} &\to \lambda\, p \cdot p\, \lambda\, x\, y \cdot y
\end{aligned}
$$

$$
\begin{aligned}
\text{pair} &\to \lambda\, x\, y\, f \cdot f\, x\, y \\
\text{first} &\to \lambda\, p \cdot p\ \text{true} \\
\text{second} &\to \lambda\, p \cdot p\ \text{false}
\end{aligned}
$$

$$
\begin{aligned}
\text{cons} &\to \lambda\, x\, y\, f \cdot f\, x\, y \\
\text{car} &\to \lambda\, p \cdot p\, \text{true} \\
\text{cdr} &\to \lambda\, p \cdot p\, \text{false} \\
\text{nil} &\to \text{false}
\end{aligned}
$$

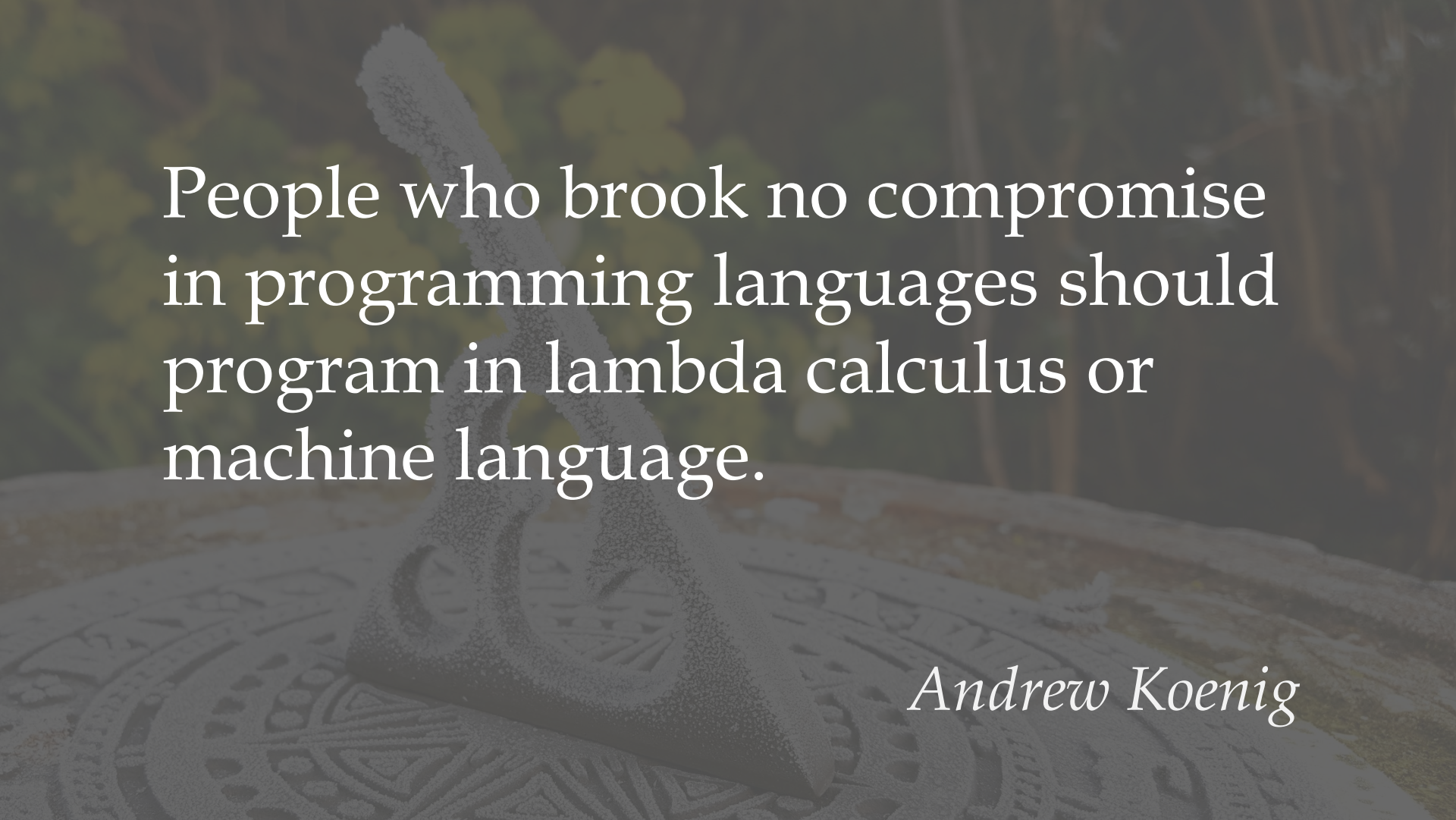
LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

$$
\begin{aligned}
\text{cons} &\to \lambda\, x\, y\, f \cdot f\, x\, y \\
\text{car} &\to \lambda\, p \cdot p\, \text{true} \\
\text{cdr} &\to \lambda\, p \cdot p\, \text{false} \\
\text{nil} &\to \text{false}
\end{aligned}
$$

$$
\begin{aligned}
\text{push} &\to \lambda\, x\, y\, f \cdot f\, x\, y \\
\text{top} &\to \lambda\, p \cdot p\ \text{true} \\
\text{pop} &\to \lambda\, p \cdot p\ \text{false} \\
\text{stack} &\to \text{false}
\end{aligned}
$$



People who brook no compromise in programming languages should program in lambda calculus or machine language.

Andrew Koenig