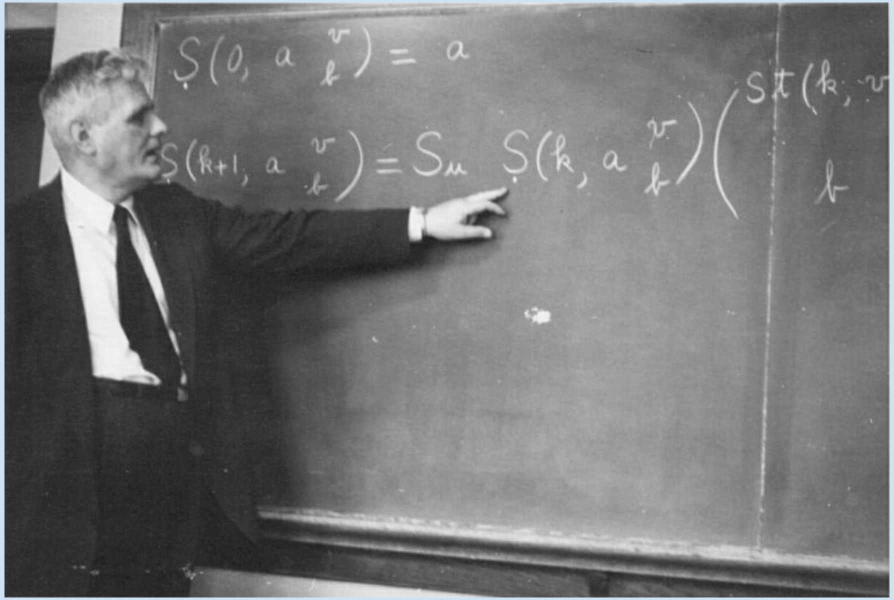
Lambda Calculus

Anson Tao
Anand Singh
Hamza Qadri

Background

- Created by mathematician Alonzo Church in the 1930s
- Simple set of rules for expressing functions as first-class values
- Many applications and important results
- We are able to express *any* computable function by combining and applying single-argument anonymous functions



Introduction

- **Abstraction:** Anonymous function that takes a single argument, ie.

  λx.(x + 1)

- **Free and Bound variables:** Bound variables attached as parameters to expression, free variables are unbound.

  x is bound, but y is free in: λx.(x + y)

- **Application:** Done through β-reduction (substituting bound variable with expression).

((λx. (λy.x + y)) 3 4)	
((λy.3 + y) 4)	Reduction on x arg.
(3 + 4)	Reduction on y arg.
7	Final result.

- **Expression:** Overarching syntactical concept that is defined in terms of variables, abstractions, and applications
  - Variables (e.g. x) are themselves expressions
  - Abstractions (e.g. λx.x) are expressions
  - The application of an expression to another (e.g. MN) is an expression

Main Result: The Power of Lambda Calculus

Turing Completeness:

- Refers to the ability of a set of rules to simulate a Turing machine
- A Turing machine, and thus a Turing-complete language, can implement any algorithm
- Lambda calculus is Turing complete - we present an informal proof for this

Conditional Branching:

- $T = \lambda x.\lambda y.x$ (select first argument)
- $F = \lambda x.\lambda y.y$ (select second argument)
- $\text{ifthenelse} = \lambda i.\lambda t.\lambda e.ite$

Example: If True then M else N

$(\text{ifthenelse})TMN$

$= (\lambda i.\lambda t.\lambda e.ite)(\lambda x.\lambda y.x)MN$

$= (\lambda t.\lambda e.(\lambda x.\lambda y.x)te)MN$

$= (\lambda e.(\lambda x.\lambda y.x)Me)N$

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

$= (\lambda y.M)N = \mathbf{M}$

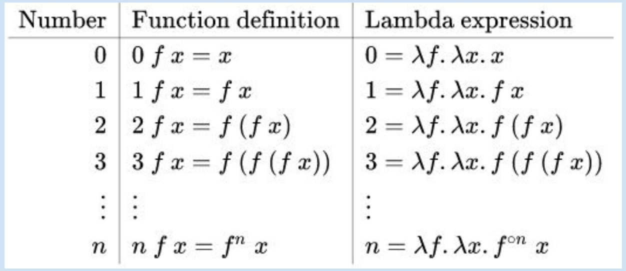
Main Result

Logic:

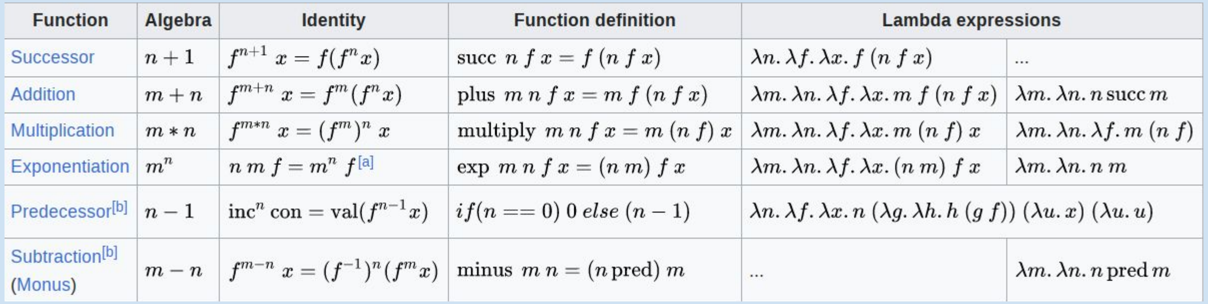
- Defined in terms of booleans and conditional branching
- not(x): if x then F else T
  - (ifthenelse)xFT
- and(x) and or(x) are left as an exercise to the viewer!

Arithmetic:

- Natural numbers are represented as Church numerals
  - n = λf.λx.f^n(x)
- S = λn.λf.λx.f(nfx)
- A = λm.λn.nSm



Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f.\lambda x.x$
1	$1\ f\ x = f\ x$	$1 = \lambda f.\lambda x.f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f.\lambda x.f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\lambda x.f^{\circ n}\ x$

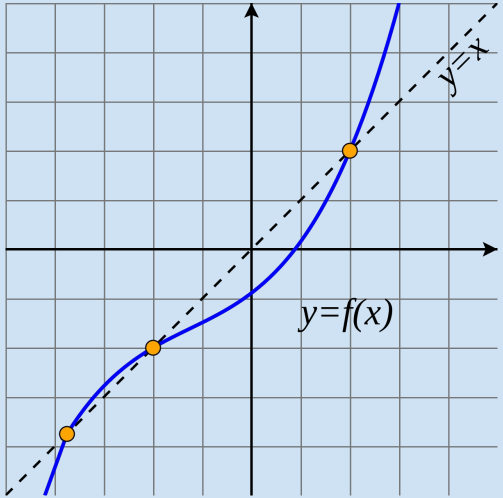


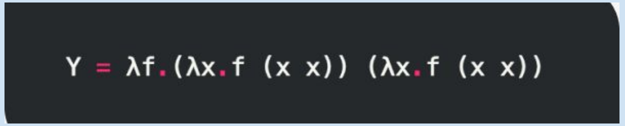
Function	Algebra	Identity	Function definition	Lambda expressions	
Successor	$n+1$	$f^{n+1}\ x = f(f^n x)$	$\operatorname{succ}\ n\ f\ x = f\ (n\ f\ x)$	$\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$	...
Addition	$m+n$	$f^{m+n}\ x = f^m(f^n x)$	$\operatorname{plus}\ m\ n\ f\ x = m\ f\ (n\ f\ x)$	$\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$	$\lambda m.\lambda n.n \operatorname{succ} m$
Multiplication	$m*n$	$f^{m*n}\ x = (f^m)^n\ x$	$\operatorname{multiply}\ m\ n\ f\ x = m\ (n\ f)\ x$	$\lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x$	$\lambda m.\lambda n.\lambda f.m\ (n\ f)$
Exponentiation	m^n	$n\ m\ f = m^n\ f$[a]	$\exp\ m\ n\ f\ x = (n\ m)\ f\ x$	$\lambda m.\lambda n.\lambda f.\lambda x.(n\ m)\ f\ x$	$\lambda m.\lambda n.n\ m$
Predecessor[b]	$n-1$	$\operatorname{inc}^n \operatorname{con} = \operatorname{val}(f^{n-1}x)$	$if(n == 0)\ 0\ else\ (n-1)$	$\lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)$	
Subtraction[b] (Monus)	$m-n$	$f^{m-n}\ x = (f^{-1})^n(f^m x)$	$\operatorname{minus}\ m\ n = (n \operatorname{pred})\ m$	...	$\lambda m.\lambda n.n \operatorname{pred} m$

Main Result

Loops via Recursion

- We cannot refer to an abstraction within its own body
- Solution: fixed-point combinators
    - Every function in lambda calculus has a fixed point
    - A fixed-point combinator (example: Y combinator) generates a function's fixed point: F(YF) = YF
    - By rewriting a function in terms of its fixed point using inverse β-reduction, and then rewriting it in terms of the Y combinator, we can achieve recursion within the rules of pure lambda calculus

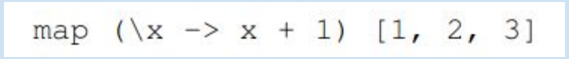




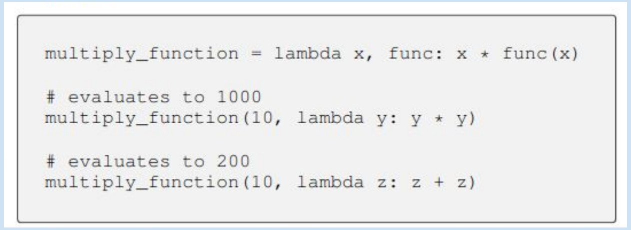
```
Y = λf.(λx.f (x x)) (λx.f (x x))
```

Applications to Functional Programming

- We have shown that it is possible to represent common programming constructs in lambda calculus
- Could we use lambda calculus as a basis for programming languages?
  - Yes! Functional programming languages (Haskell, Clojure, etc.)
  - Even many non-functional languages support the paradigm
- Advantages of functional programming paradigm
  - Maintainability: functions have no external effects, so there is less coupling
  - Debugging: pure functions have no state or side effects, so finding bugs is easier
  - Modularity: functions can be composed together without affecting each other
  - Concurrency: it is easier and more natural to implement concurrency due to the separation of functions



```
map (\x -> x + 1) [1, 2, 3]
```



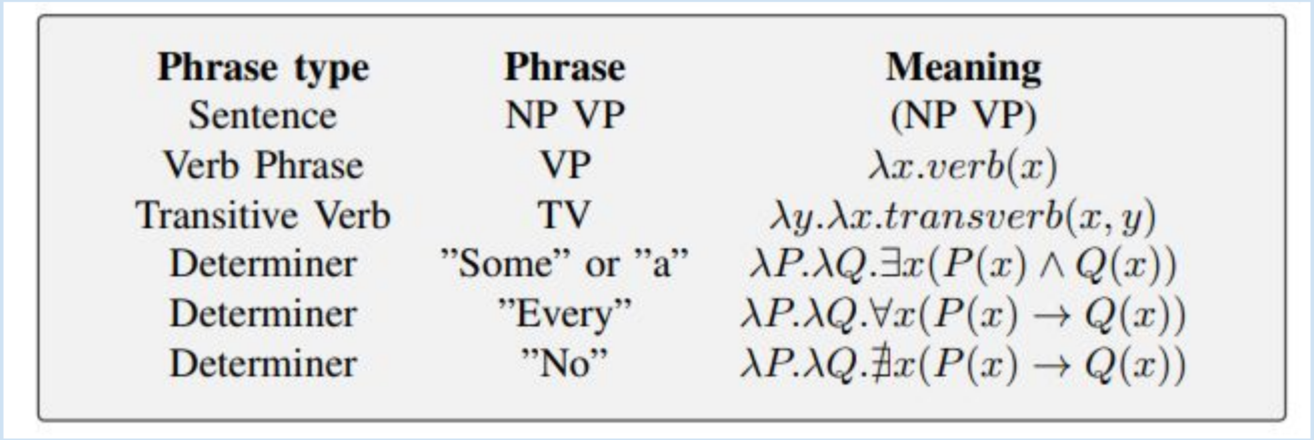
```
multiply_function = lambda x, func: x * func(x)

# evaluates to 1000
multiply_function(10, lambda y: y * y)

# evaluates to 200
multiply_function(10, lambda z: z + z)
```

Generalization: Montague Semantics and NLP

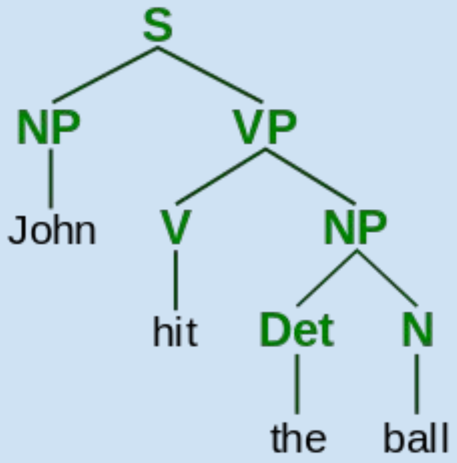
- Aims to provide a formal treatment of natural language meaning by using lambda calculus to represent the meaning of expressions
  - Determiners: Specify reference and quantity (“a”, “an”, “some,” “the”)
  - Noun Phrase: Functions as subject in sentence
  - Verb Phrase: Verbs and complements, objects, and modifiers



Phrase type	**Phrase**	**Meaning**
Sentence	NP VP	(NP VP)
Verb Phrase	VP	$\lambda x.verb(x)$
Transitive Verb	TV	$\lambda y.\lambda x.transverb(x, y)$
Determiner	”Some” or ”a”	$\lambda P.\lambda Q.\exists x(P(x) \wedge Q(x))$
Determiner	”Every”	$\lambda P.\lambda Q.\forall x(P(x) \rightarrow Q(x))$
Determiner	”No”	$\lambda P.\lambda Q.\nexists x(P(x) \rightarrow Q(x))$

Generalization: Montague Semantics and NLP

- Rules of montague grammar can be used in addition to a parse tree to transform a sentence into some proposition.

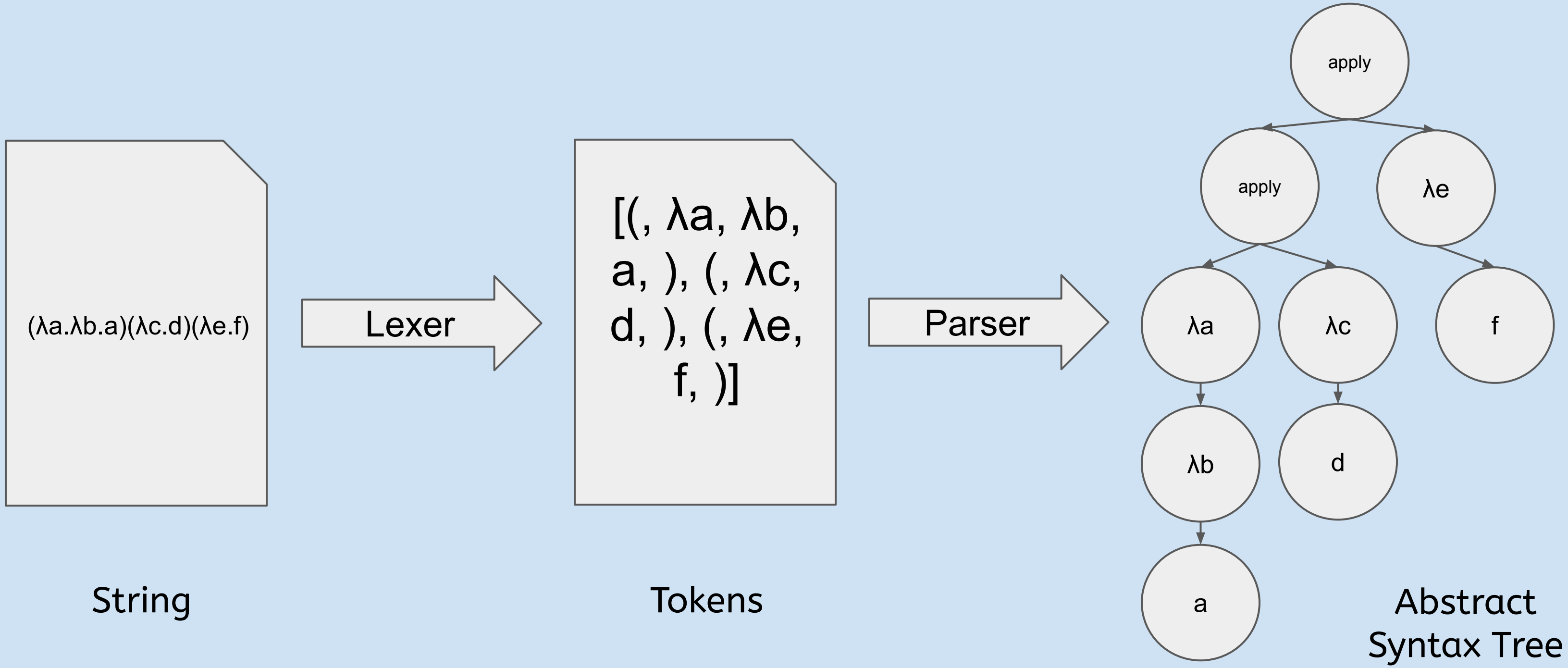


"Hit" is a transitive verb here, so it is likely transformed into something like this:

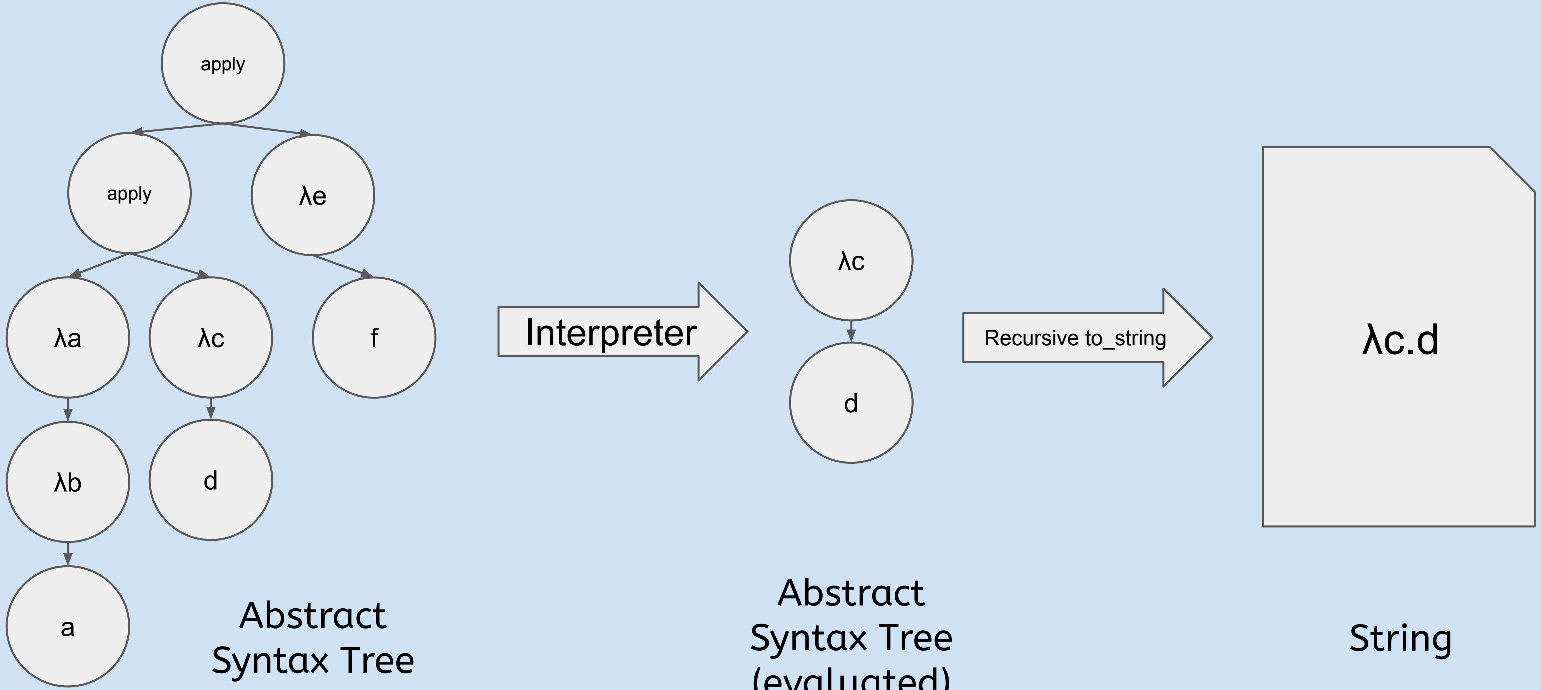
λx.λy.hit(x,y)(John, ball)

But mapping a CFG to lambda-calculus from text is still currently being researched.
https://www.computer.org/csdl/proceedings-article/ictai/2008/3440b135/12OmNx7ouR0

Lambda Calculus Interpreter



Lambda Calculus Interpreter



Abstract Syntax Tree

Abstract Syntax Tree (evaluated)

String

References

- Brilliant Math and Science Wiki: Lambda Calculus - This website rigorously yet clearly defines the syntax of lambda calculus as well as how to express common constructs in it. It was a very useful reference for our main result.

- Stanford Library of Philosophy: The Lambda Calculus - This is another introduction to lambda calculus that explains some aspects of its usage in more detail and also introduced us to our generalization topic.

- Jason Eisner: Functional Programming - This document explains the relationship between functional programming and lambda calculus. It allowed us to understand how lambda calculus serves as the basis for the functional programming paradigm, and programming languages in general.

- Stanford Library of Philosophy: Montague Semantics - This website, also from the Stanford Library of Philosophy, offered a very detailed introduction to and description of Montague semantics that we referred to extensively.

- Richard Montague: The Proper Treatment of Quantification in Ordinary English - This is a relevant research paper from Richard Montague, the mathematician who pubished what became known as Montague semantics. As such, it was very useful for us.

- NLTK: Sample usage for logic - This is the official documentation for Python's NLTK library; as such, it was a useful reference when writing the section on NLTK.

- Lisperator: Writing a parser - This website served as a good (albeit generic) guide for how to develop interpreters, and gave us sufficient direction to begin writing our lexer and parser.

- University of Wisconsin: Lambda Calculus - This document introduced us to abstract syntax trees for lambda calculus. Though our ASTs are slightly different from the examples given here, this page still served as a good introduction.