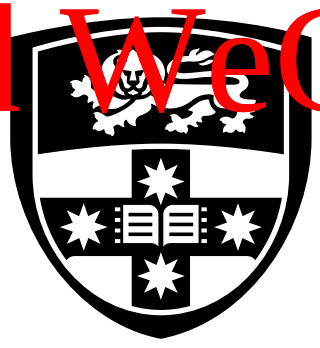
COMP2022: Formal Languages and Logic

2018, Semester 2, Week 1

Joseph Godbehere

2nd August, 2018



THE UNIVERSITY OF SYDNEY

Outline

- **Why study formal languages and logic?**

- Mathematical notions and definitions
- Introduction to Functional Programming

- Introduction to the lambda calculus

Why should we study theory?

Computers are complex machines and theory provides a new viewpoint, an elegant side to computation.

Studying theory expands your mind. IT evolves quickly. Whilst specific technical knowledge is quickly outdated, theory is not.

- Know how to express yourself clearly
- Know how to prove your work
- Know when you have or have not, solved a problem

Theory provides conceptual tools which are used in computer engineering

How this course will help

Most problems in computer science involve answering:

- Can the problem be solved by a computer program?
- If not, can you modify the problem so that it can?
- If so, can a program solve the problem in workable time?
- Can you prove that your program is correct?
- Can you prove that your program is efficient?

How this course will help

- Lambda calculus (functional programming paradigm)
  - Relationships between data
  - Some notable properties:
    - Elegant
    - Efficiency - parallelism, laziness
    - Security
  - Some example uses:
    - implementing programming languages (compilers)
    - concurrent and parallel systems
    - secure systems
    - ... and more generally, anything computable

How this course will help

- Automata Theory (imperative programming paradigm)
  - Transformations of program state
- Some notable properties:
  - Easy to design
  - Easy to reason about
- Some example uses:
  - Text processing / pattern matching (e.g. regular expressions)
  - Model checking (e.g. to verify correctness of communication protocols and electronic circuits)
  - Agent based game 'AI'
  - Hardware design
  - ...

How this course will help

Formal languages (grammars, especially context-free grammars)

- How we structure information
- Some example uses:
  - Compilers and programming langauges (syntax)
  - Natural Language Processing (e.g. machine translation, AI)
  - Data storage (e.g. XML)
  - ...

How this course will help

Logic: *Meaning*

- Program logic - logic gates, conditional statements
- Software verification
- Databases
- Artificial Intelligence
- Automated Reasoning
- ...

Understanding limitations of computing

- When developing solutions to real problems, we need to understand the limitations of what software can do, and design solutions which are realistic and efficient.
  - *Undecidable* problems: no program/algorithm can do it (for all possible inputs)
  - *Intractable* problems: there could be programs/algorithms, but too slow to be usable

- Theory will help you understand and recognise these

Course topics

- Lambda Calculus
  - Rewrite rules, reductions
  - Encodings, commutative diagrams
  - y-combinator, functional programming
- Automata Theory
  - Regular Languages
    - Finite automata, regular expressions
  - Context-free Languages
    - Pushdown Automata, C-F grammars, parsers
  - Turing Machines
    - Church-Turing thesis
    - Computability, decidability, tractability
- Logic
  - Propositional and predicate logic
  - Logic formal proofs

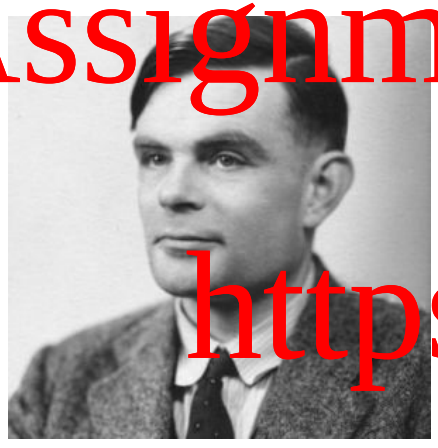
Gottfried Wilhelm Leibniz

- Polymath and Philosopher (17th C)
- Amongst MANY achievements, invents the **Entscheidungsproblem**
  1. Suppose we have a univeral language in which all problems can be stated (e.g. set theory + predicate logic)
  2. Can we find a decision method to solve all the problems?

Alonzo Church

- Mathematician, logician
- Created lambda calculus
- Proved that the Entscheidungsproblem is undecidable
- Proved Peano arithmetic is undeciable
- Church-Turing thesis

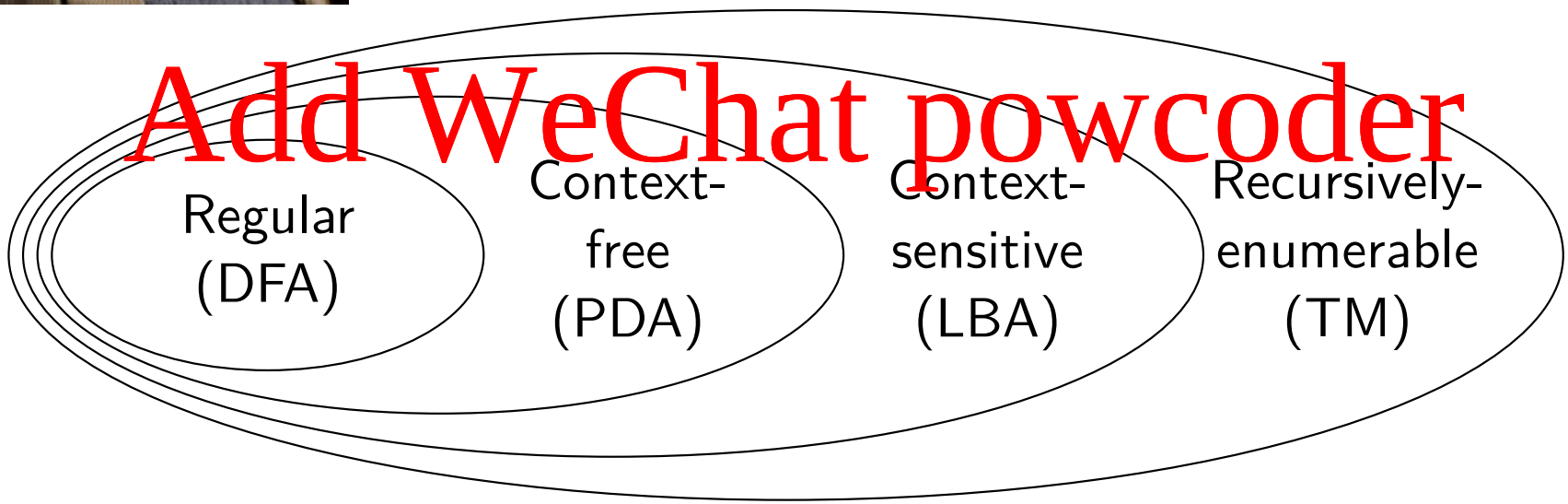
Alan Turing



- Founder of computer science, mathematician, philosopher, code breaker, visionary
- Created abstract machines called *Turing machines* in the 1930s, before computers existed
- Church-Turing thesis
- Turing test
  - Can machines think?
  - The Imitation Game
- Enigma code breaker

Noam Chomsky

- Linguist, philosopher, cognitive scientist, logician, historian, political critic and activist
- Chomsky Hierarchy: a containment hierarchy of classes of formal languages (applies to human language and computer theory)



Outline

- Why study formal languages and logic?
- **Mathematical notions and definitions**

- Introduction to Functional Programming

- Introduction to the lambda calculus

Set

- *Set*: An unordered group of unique objects
  - *unordered*: $A = \{a, b, c\} = \{b, c, a\} = \{c, b, a\}$
  - *contains*: $a \in A$ denotes that a is in A
  - *subsets*: if $X \subseteq A$ then every element $x \in X$ is also in A
    - $\{a, c\} \subseteq A$
    - $\{b\} \subseteq A$
    - $\{a, d\} \not\subseteq A$
  - *size*: $|A|$ denotes the number of objects in A
  - The *empty set* contains no elements (denoted $\emptyset$ or $\{\}$)

2-Tuple

Pair or *2-Tuple*: an ordered list of two objects

- *ordered*: $(a, b) \neq (b, a)$
- *duplicates allowed*: (a, a) is allowed

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B
  - $x \in A \cup B$ if and only if $x \in A$ or $x \in B$ (or both)

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B
  - $x \in A \cup B$ if and only if $x \in A$ or $x \in B$ (or both)
- *Intersection*

  - Denoted $A \cap B$
  - The set of elements belonging to both A and B
  - $x \in A \cap B$ if and only if $x \in A$ and $x \in B$

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B
  - $x \in A \cup B$ if and only if $x \in A$ or $x \in B$ (or both)
- *Intersection*
  - Denoted $A \cap B$
  - The set of elements belonging to both A and B
  - $x \in A \cap B$ if and only if $x \in A$ and $x \in B$
- *Subtraction*
  - Denoted $A \setminus B$
  - The set of elements belonging to A which do not belong to B
  - $x \in A \setminus B$ if and only if $x \in A$ and not $x \in B$

Power Set

$\mathcal{P}(A)$ denotes the Power Set of A which is the set of all the subsets of A.

- Including $\emptyset$, the *empty set*

Examples:

- If $A = \{0, 1\}$ then $\mathcal{P}(A) = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$
- If $A = \{a, b, c\}$ then
  $\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

Cartesian product of two sets A and B

- The set of all pairs where the first element is in A and the second element is in B, denoted $A \times B$

- $(x, y) \in A \times B$ if and only if $x \in A$ and $y \in B$

- $\{0, 1\} \times \{a, b\} = \{(0, a), (0, b), (1, a), (1, b)\}$

Function $f : A \to B$

- Defines an input-output relationship from the set A to B
- The set of possible inputs to f is the domain $D \subseteq A$
- The set of possible outputs is the co-domain, or range $R \subseteq B$
- For each $a \in D$, f produces exactly one output $f(a) \in R$.
  - multiple inputs can produce the same output
  - i.e. a *many-to-one* function
  - e.g. if $A = \{0, 1, 2\}$, $B = \{a, b\}$, then we might define $f : A \to B$ as $f(0) = f(1) = a$ and $f(2) = b$
- f can be thought of as a subset of $A \times B$
  - e.g. in the example above, $f = \{(0, a), (1, a), (2, b)\}$

Outline

- Why study formal languages and logic?

- Mathematical notions and definitions
- **Introduction to Functional Programming**

- Introduction to the lambda calculus

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ∼1930's
  - different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation recursive function definitions, and combinatorics

- These results led Church to conjecture that any intuitively appealing model of computing would be equally powerful as well
  - This conjecture is known as Church's thesis

Historical Origins

Turing's model of computing was the Turing machine, a sort of pushdown automaton using an unbounded storage "tape"

- The Turing machine computes in an imperative way, by changing the values written on its tape, analogous to how higher level imperative programs compute by changing the values of variables.
- We will explore this when we cover Automata Theory in the middle part of this unit of study.

Historical Origins

- Church's model of computation is called the **Lambda Calculus**
- Based on the notation of parameterised expressions, with each parameter introduced by a letter λ (lambda)
- Lambda calculus was the inspiration for functional programming
- You can use it to compute by substituting parameters into expressions, like passing arguments to functions

Functional Programming Concepts

- Functional languages are an attempt to realize Church's lambda calculus in practical form as a programming language
  - e.g. Lisp, Scheme, FP, ML, Miranda, Haskell, ...

- The key idea: do everything by function composition
  - No mutable state $\rightarrow$ no side effects

Expressions

- Expressions are compositions of functions
- Constants are functions with no domains (e.g. $f() = 1$)
- Examples:


```
append(append("a", "b"), "c")
(if x (if y 1 2) 3)
```

Programs

- A functional program is an expression E, which represents both the program and the input

Programs

- A functional program is an expression E, which represents both the program and the input
- We compute E by reducing it using **rewrite rules**

Programs

- A functional program is an expression E, which represents both the program and the input
- We compute E by reducing it using **rewrite rules**
  - Each reduction replaces some subexpression P of E with P' by applying one of the rewrite rules

Programs

- A functional program is an expression E, which represents both the program and the input
- We compute E by reducing it using **rewrite rules**
  - Each reduction replaces some subexpression P of E with P' by applying one of the rewrite rules
  - Schematic representation:

$$E\,[P] \to E\,[P']$$

where $P \to P'$ holds according to the rewrite rules

Programs

- A functional program is an expression E, which represents both the program and the input
- We compute E by reducing it using **rewrite rules**
  - Each reduction replaces some subexpression P of E with P' by applying one of the rewrite rules
  - Schematic representation:

$$E\,[P] \to E\,[P']$$

  where $P \to P'$ holds according to the rewrite rules

- This process is repeated until no more reductions are applicable

Programs

- A functional program is an expression E, which represents both the program and the input
- We compute E by reducing it using **rewrite rules**
  - Each reduction replaces some subexpression P of E with P' by applying one of the rewrite rules
  - Schematic representation:

$$E[P] \to E[P']$$

where $P \to P'$ holds according to the rewrite rules

- This process is repeated until no more reductions are applicable
- We say the resulting expression is in **Normal Form**

Example

- Consider these rewrite rules (a **reduction system**)
  - $a + b \mapsto c$ where c is the addition of a and b
  - $a \times b \mapsto c$ where c is the multiplication of a and b
- ... and this algebraic expression: $(1 + 2) \times (3 + 2)$

Example

- Consider these rewrite rules (a **reduction system**)
  - $a + b \mapsto c$ where c is the addition of a and b
  - $a \times b \mapsto c$ where c is the multiplication of a and b
- .. and this algebraic expression: $(1 + 2) \times (3 + 2)$
- Reductions into Normal Form:

Step	E	P	P'	rule
1	$(1+2) \times (3+2)$	$1+2$	3	$a + b \mapsto c$
2	$3 \times (3+2)$	$3+2$	5	$a + b \mapsto c$
3	3×5	3×5	15	$a \times b \mapsto c$
4	15			

Church-Rosser Property

Reduction systems are usually designed to satisfy the Church-Rosser property – that an expression's normal form is independent of the order of evaluation of the subexpressions.

Step	E	P	P'	rule
1	$(1+2)\times(3+2)$	$1+2$	3	$a+b\mapsto c$
2	$3\times(3+2)$	$3+2$	5	$a+b\mapsto c$
3	3×5	3×5	15	$a\times b\mapsto c$
4	15			

Step	E	P	P'	rule
1	$(1+2)\times(3+2)$	$3+2$	5	$a+b\mapsto c$
2	$(1+2)\times 5$	$1+2$	3	$a+b\mapsto c$
3	3×5	3×5	15	$a\times b\mapsto c$
4	15			

Outline

- Why study formal languages and logic?

- Mathematical notions and definitions

- Introduction to Functional Programming

- **Introduction to the lambda calculus**

Application

The first basic operation of the λ-calculus is **application**

$$(F \cdot A)$$

denotes the **application** of some algorithm F to some input A.

Application

The first basic operation of the λ-calculus is **application**

$$(F \cdot A)$$

denotes the **application** of some algorithm F to some input A.

For example, suppose A was simply the number 3, and F was the function $x \mapsto x + 1$, then the normal form of $(F \cdot A)$ would be 4.

Application

The first basic operation of the λ-calculus is **application**

$$(F \cdot A)$$

denotes the **application** of some algorithm F to some input A.

For example, suppose A was simply the number 3, and F was the function $x \mapsto x + 1$, then the normal form of $(F \cdot A)$ would be 4.

Often, we omit the $\cdot$ and simply write FA.

Application

Note that E and A can be arbitrary expressions.

So, continuing with our example (A is 3, F is $x \mapsto x + 1$), we can also compute recursive expressions like:

$$(F \cdot (F \cdot (F \cdot A)))$$

Application

Note that E and A can be arbitrary expressions.

So, continuing with our example (A is 3, F is $x \mapsto x + 1$), we can also compute recursive expressions like:

$$(F \cdot (F \cdot (F \cdot A))) \to (F \cdot (F \cdot (F \cdot 3)))$$

Application

Note that E and A can be arbitrary expressions.

So, continuing with our example (A is 3, F is $x \mapsto x + 1$), we can also compute recursive expressions like:

$$\begin{aligned}(F \cdot (F \cdot (F \cdot A))) &\to (F \cdot (F \cdot (F \cdot 3))) \\ &\to (F \cdot (F \cdot 4))\end{aligned}$$

Application

Note that E and A can be arbitrary expressions.

So, continuing with our example (A is 3, F is $x \mapsto x + 1$), we can also compute recursive expressions like:

$$\begin{aligned}(F \cdot (F \cdot (F \cdot A))) &\to (F \cdot (F \cdot (F \cdot 3))) \\ &\to (F \cdot (F \cdot 4)) \\ &\to (F \cdot 5)\end{aligned}$$

Application

Note that F and A can be arbitrary expressions.

So, continuing with our example (A is 3, F is $x \mapsto x + 1$), we can also compute recursive expressions like:

$$\begin{aligned}(F \cdot (F \cdot (F \cdot A))) &\to (F \cdot (F \cdot (F \cdot 3))) \\ &\to (F \cdot (F \cdot 4)) \\ &\to (F \cdot 5) \\ &\to 6\end{aligned}$$

Abstraction

The second basic operation of the λ-calculus is **abstraction**.

$$\lambda x.M[x]$$

denotes the function $x \mapsto M[x]$.

i.e. some expression M, where any instance of the variable x in M has been replaced with the input.

Abstraction

The second basic operation of the λ-calculus is **abstraction**.

$$\lambda x.M[x]$$

denotes the function $x \mapsto M[x]$.

i.e. some expression M, where any instance of the variable x in M has been replaced with the input.

Examples

- $\lambda x.x$ is the function

Abstraction

The second basic operation of the λ-calculus is **abstraction**.

$$\lambda x.M[x]$$

denotes the function $x \mapsto M[x]$.

i.e. some expression M, where any instance of the variable x in M has been replaced with the input.

Examples

- $\lambda x.x$ is the function $x \mapsto x$, i.e. $f(x) = x$
- $\lambda x.4$ is the function

Abstraction

The second basic operation of the λ-calculus is **abstraction**.

$$\lambda x.M[x]$$

denotes the function $x \mapsto M[x]$.

i.e. some expression M, where any instance of the variable x in M has been replaced with the input.

Examples

- $\lambda x.x$ is the function $x \mapsto x$, i.e. $f(x) = x$
- $\lambda x.4$ is the function $x \mapsto 4$, i.e. $f(x) = 4$
- $\lambda x.(square \cdot x)$ is the function

Abstraction

The second basic operation of the λ-calculus is **abstraction**.

$$\lambda x.M[x]$$

denotes the function $x \mapsto M[x]$.

i.e. some expression M, where any instance of the variable x in M has been replaced with the input.

Examples

- $\lambda x.x$ is the function $x \mapsto x$, i.e. $f(x) = x$
- $\lambda x.4$ is the function $x \mapsto 4$, i.e. $f(x) = 4$
- $\lambda x.(square \cdot x)$ is the function $x \mapsto (square \cdot x)$, i.e. $f(x) = x^2$

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

- $((\lambda y.(f \cdot y)) \cdot 3) \to (f \cdot 3) \to 4$

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

- $((\lambda y.(f \cdot y)) \cdot 3) \to (f \cdot 3) \to 4$
- $((\lambda y.(g \cdot y)) \cdot 3) \to (g \cdot 3) \to 9$

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

- $((\lambda y.(f \cdot y)) \cdot 3) \to (f \cdot 3) \to 4$
- $((\lambda y.(g \cdot y)) \cdot 3) \to (g \cdot 3) \to 9$
- $(\lambda z.((\lambda y.(g \cdot y)) \cdot z) \cdot 5) \to$

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

- $((\lambda y.(f \cdot y)) \cdot 3) \to (f \cdot 3) \to 4$
- $((\lambda y.(g \cdot y)) \cdot 3) \to (g \cdot 3) \to 9$
- $(\lambda z.((\lambda y.(g \cdot y)) \cdot z) \cdot 5) \to ((\lambda y.(g \cdot y)) \cdot 5) \to$

Application and abstraction

We can easily combine the rules, for example, suppose we have

- $f : x \mapsto x + 1$
- $g : x \mapsto x^2$

Then the following expressions reduce like this:

- $((\lambda y.(f \cdot y)) \cdot 3) \to (f \cdot 3) \to 4$
- $((\lambda y.(g \cdot y)) \cdot 3) \to (g \cdot 3) \to 9$
- $(\lambda z.((\lambda y.(g \cdot y)) \cdot z) \cdot 5) \to ((\lambda y.(g \cdot y)) \cdot 5) \to (g \cdot 5) \to 25$

Parentheses, parentheses everywhere...

You might've noticed by now that I've been writing a *lot* of parentheses. for example, do we *really* need to write:

$$(F \cdot (F \cdot (F \cdot 3)))$$

Parentheses, parentheses everywhere...

You might've noticed by now that I've been writing a *lot* of parentheses. for example, do we *really* need to write:

$$(F \cdot (F \cdot (F \cdot 3)))$$

No! We could write this trivial expression unambiguously as $FFF3$

Parentheses, parentheses everywhere...

You might've noticed by now that I've been writing a *lot* of parentheses. for example, do we *really* need to write:

$$(F \cdot (F \cdot (F \cdot 3)))$$

No! We could write this trivial expression unambiguously as $FFF3$

So far we've only considered functions with:

- 1 parameter (e.g. "square", "increment")
- 0 parameters (constants, e.g. 1, 2, 3...)

... It quickly gets more complex as we use abstraction (λ).

Next week we'll learn about when it is – or isn't – safe to simplify the notation.

Until then, we'll keep writing everything out in full.

Free and bound variables

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

- x is a *bound* variable, because the λ binds it.
- y is a *free* variable, because it is not bound by any λ.

Free and bound variables

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

- The first occurrence of x is a *free* variable.
- The second occurrence of x is a *bound* variable.
- y is a *free* variable.

Free and bound variables

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

- The first occurrence of x is a *bound* variable.
- The second occurrence of x is a *free* variable, because it is not in the scope of the λx.
- y is a *free* variable.

Free and bound variables

$$(\lambda x.(x \cdot (\lambda x.(x \cdot x))))$$

- The first occurrence of x is *bound* to the first λ.
- The second occurrence of x is *bound* to the second λ.
- The third occurrence of x is also *bound* to the second λ.

β-reduction (abstraction)

We can now define the **abstraction** operation more formally. It is the axiom:

$$(\lambda x.M) \cdot N = M[x := N]$$

This means the output of $(\lambda x.M) \cdot N$ is the result of replacing every free occurrence of x in M with N.

β-reduction (abstraction)

We can now define the **abstraction** operation more formally. It is the axiom:

$$(\lambda x.M) \cdot N = M[x := N]$$

This means the output of $(\lambda x.M) \cdot N$ is the result of replacing every free occurrence of x in M with N.

Important: Only the *free* occurrences! Example:

$$(yx(\lambda x.x))[x := N] = yN(\lambda x.x)$$

Renaming bound variables

- $(\lambda x.x) \cdot x$ is confusing because the first occurrence of x is bound, but the second is free
- Bound variables have a similar scope to variable scope in imperative programming languages
  - e.g. in Java

```
int x = 1;
{int x = 2; System.out.println(x);}
System.out.println(x);
```

would output 2, 1.

Renaming bound variables

- $(\lambda x.x) \cdot x$ is confusing because the first occurrence of x is bound, but the second is free.
- Bound variables have a similar scope to variable scope in imperative programming languages
  - e.g. in Java

```
int x = 1;
{int x = 2; System.out.println(x);}
System.out.println(x);
```

would output 2, 1.

- We can apply the same notion of scope to rename the occurences of a variable bound by a particular λ.
- Example: $(\lambda x.x) \cdot x = (\lambda y.y) \cdot x$

Don't relabel any occurences that weren't bound to that λ:

- Mistake: $((\lambda x.(y \cdot x)) \cdot x) \neq ((\lambda z.(y \cdot z)) \cdot z)$
- Correct: $((\lambda x.(y \cdot x)) \cdot x) \equiv ((\lambda z.(y \cdot z)) \cdot x)$

Don't relabel any occurences that weren't bound to that λ:

- Mistake: $((\lambda x.(y \cdot x)) \cdot x) \neq ((\lambda z.(y \cdot z)) \cdot z)$
- Correct: $((\lambda x.(y \cdot x)) \cdot x) = ((\lambda z.(y \cdot z)) \cdot x)$

Don't skip any of the occurrences bound to that λ:

- Mistake: $(\lambda x.(x \cdot x)) \neq (\lambda y.(y \cdot x))$
- Correct: $(\lambda x.(x \cdot x)) = (\lambda y.(y \cdot y))$

Don't relabel any occurences that weren't bound to that λ:

- Mistake: $((\lambda x.(y \cdot x)) \cdot x) \neq ((\lambda z.(y \cdot z)) \cdot z)$
- Correct: $((\lambda x.(y \cdot x)) \cdot x) = ((\lambda z.(y \cdot z)) \cdot x)$

Don't skip any of the occurrences bound to that λ:

- Mistake: $(\lambda x.(x \cdot x)) \neq (\lambda y.(y \cdot x))$
- Correct: $(\lambda x.(x \cdot x)) = (\lambda y.(y \cdot y))$

Don't change the binding of the other variables (use a new label)

- Mistake: $(\lambda x.(x \cdot y)) \neq (\lambda y.(y \cdot y))$
- Correct: $(\lambda x.(x \cdot y)) = (\lambda z.(z \cdot y))$

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is bound to the second λ
- the third occurrence of x is

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is bound to the second λ
- the third occurrence of x is bound to the first λ
- the fourth occurrence of x is

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is bound to the second λ
- the third occurrence of x is bound to the first λ
- the fourth occurrence of x is a free variable
- ... so confusing!

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is bound to the second λ
- the third occurrence of x is bound to the first λ
- the fourth occurrence of x is a free variable
- ... so confusing!

Rename the first λ with y:

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

Make sure you know which variables are bound to which λ!

$$(\lambda x.(x \cdot (\lambda x.x)) \cdot x) \cdot x$$

- the first occurrence of x is bound to the first λ
- the second occurrence of x is bound to the second λ
- the third occurrence of x is bound to the first λ
- the fourth occurrence of x is a free variable
- ... so confusing!

Rename the first λ with y:

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

Rename the second λ with z:

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

Review

- Motivation for studying theory
- Mathematical notions and notation
- Introduction to functional programming
  - Functions
  - Expressions
  - Rewrite rules
- Introduction to the lambda calculus
  - Function application
  - Abstraction (β-reduction)
  - Free and bound variables
  - Renaming