COMP2022: Formal Languages and Logic

2018, Semester 2, Week 1

THE UNIVERSITY OF SYDNEY

Outline

- Why study formal languages and logic?

- Introduction to Functional Programmi

- 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

Stud
speci

-
- Know how to prove your work
- Know when you have, or have not, solved a pro

Theory provides conceptual tools which are used in computer engineering

How this course will help

Most problems in computer science involve answering:

-
-
-
- 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 example uses:
  - implementing programming languag
  - 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
-

-
  - Text processing / pattern matching (e.
  - Model checking (e.g. to verify correcthe protocols and electronic circuits)
  - Agent based game 'AI'
  - Hardware design
  - ...

How this course will help

Formal languages (grammars, especially context-free grammars)

-
-
  - Natural Language Processing (e.g. ma
  - Data storage (e.g. XML)
  - ...

How this course will help

Logic: *Meaning*

-
-
-
- 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

t

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
-
  - Context-free Languages
    - Pushdown Automata, C-F gram
  - Turing Machines
    - Church-Turing thesis
    - Computability, decidability, tractability
- Logic
  - Propositional and predicate logic
  - Logic formal proofs

Gottfried Wilhelm Leibniz

e

which all prob
theory + pred

2. Can we find a de
the problems

Alonzo Church

- Proved Peano arit
- Church-Turing

Alan Turing

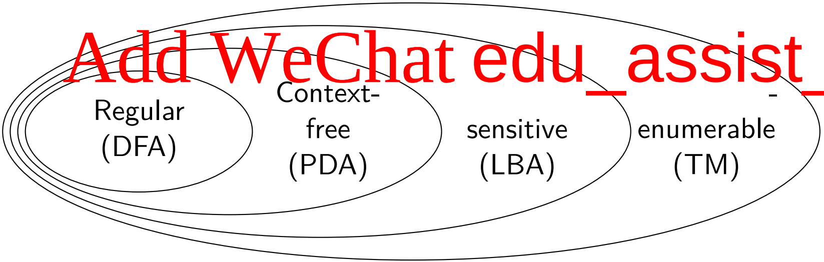
- Founder of computer science, mathematician, philosopher, code breaker, visionary

uring

- Church-Turing t
- Turing test
  - Can machine
  - The Imitatio
- Enigma code breaker

Noam Chomsky

- Linguist, philosopher, cognitive scientist, logician, historian, political critic and activist
- Chomsky Hierarchy: a containment hierarchy



Outline

- Why study formal languages and logic?

- Introduction to Functional Programmi

- Introduction to the lambda calculus

Set

Set: An unordered group of unique objects

- *unordered*: $A = \quad a, b, c \quad = \quad b, c, a \quad = \quad c, b, a$
-
- A
  - $\{b\} \subseteq A$
  - $\{a, d\} \not\subseteq A$
-
- *size*: $|A|$ denotes the number of objects i
- The *empty set* contains no elements (denoted $\emptyset$ or $\{\}$)

2-Tuple

Pair

-
- (a, a)

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B

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

Set Operations

- *Union*
  - Denoted $A \cup B$
  - The set of elements belonging to at least one of A or B

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

- *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.

-

Examples:

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

- 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

- $(x, y) \in A \times B$ $x \in A$ y 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 $D \subseteq A$
- $R \subseteq B$

- R.
  - multiple inputs can produce the same ou
  - i.e. a *many-to-one* function
  - e.g. if $A = \{0, 1, 2\}$, $B = \{a, b$

    $f : A \to B$ as $f(0) = f(1) = a$
- 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?
-
- **Introduction to Functional Program**

- Introduction to the lambda calculus

Historical Origins

- The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene,

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

Historical Origins

Turing's model of computing was the Turing machine, a sort of push

-

  values of variables.
- We will explore this when we cover Automat

  middle part of this unit of study.

Historical Origins

- Church's model of computation is called the **Lambda**
-
-
  programming
- You can use it to compute by substituting par expressions, like passing arguments to fu

Functional Programming Concepts

- Functional languages are an attempt to realize Church's

- The key idea: do everything by function com
  - No mutable state → no side effect

Expressions

- Expressions are compositions of functions
-
-


```
(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**

ith P' by

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**

ith P' by

$$E\,[P]$$

where $P \to P'$ holds according to t

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**

ith P' by

$$E\left[P\right]$$

where $P \to P'$ holds according to t

- This process is repeated until no more reduc 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**

ith P' by

$$E[P]$$

where $P \to P'$ holds according to t

- This process is repeated until no more reduc 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
-

Example

- Consider these rewrite rules (a **reduction system**)
  - $a + b \mapsto c$ where c is the addition of a and b
  - $a \quad b \mapsto c$ where c is the multiplication of a and b
-
-

	E	P	P	
1	$(1+2) \times (3+2)$			$\mapsto$
2	$3 \times (3+2)$			$\mapsto$
3	3×5			$\mapsto$
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
4	15			

Step	E	P		
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?

- Introduction to Functional Programmi

- **Introduction to the lambda calculus**

Application

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

$$(F \quad A)$$

deno

A.

Application

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

$$(F \quad A)$$

deno [illegible] A.

For example, suppose A was simply the nu [illegible] e function $x \mapsto x + 1$, then the normal form of [illegible] 4.

Application

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

$$(F \quad A)$$

deno ... A.

For example, suppose A was simply the nu ... e
function $x \mapsto x + 1$, then the normal form of ... 4.

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

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:

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:

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:

$$\rightarrow (F$$

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:

$\rightarrow (F$

$\rightarrow (F$

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:

$$\rightarrow (F$$

$$\rightarrow (F$$

$$\rightarrow 6$$

Abstraction

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

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

deno

i.e. so x in M

has be

Abstraction

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

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

deno

i.e. so x in M
has be

Examples

- $\lambda x.x$ is the function

Abstraction

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

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

deno

i.e. so x in M
has be

Examples

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

Abstraction

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

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

deno

i.e. so x in M
has be

Examples

- $\lambda x.x$ is the function $x \mapsto x$, i.e. f
- $\lambda x.4$ 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**.

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

deno

i.e. so x in M
has be

Examples

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

-
-

The

Application and abstraction

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

-
-

The

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

Application and abstraction

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

-
-

The

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

Application and abstraction

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

-
-

The

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

Application and abstraction

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

-
-

The

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

Application and abstraction

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

-
-

The

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

$$(F \quad (F \quad (F \quad 3)))$$

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:

$$(F \quad (F \quad (F \quad 3)))$$

No! W $FFF3$

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:

$$(F \quad (F \quad (F \quad 3)))$$

No! W $FFF3$

So far

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

... It quickly gets more complex as we use abstractio

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

- x is a *bound* variable, because the
- y is a *free* variable, because it is not bou

Free and bound variables

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

Free and bound variables

-
- The second occurrence of x is a *f*
  not in the scope of the λx.

- y is a *free* variable.

Free and bound variables

- x λ
- The second occurrence of x is *bo*
- The third occurrence of x is also

β-reduction (abstraction)

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

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

This

ever

β-reduction (abstraction)

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

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

This

ever

Important: Only the *free* occurrences! E

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

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


```
{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


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


would output 2, 1.

- We can apply the same notion of scope to rena 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: $((\lambda x.(y \cdot x)) \cdot x) = ((\lambda z.(y \cdot z)) \cdot x)$

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'

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

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'

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

Don't change the binding of the other variables (us

- 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
-

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
-
-

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
-
-
-

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
-
-
-

Rename the first λ with y:

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

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
-
-
-

Rename the first λ with y:

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

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 the lambda calculus

  - Function application
  - Abstraction (β-reduction)
  - Free and bound variables
  - Renaming