

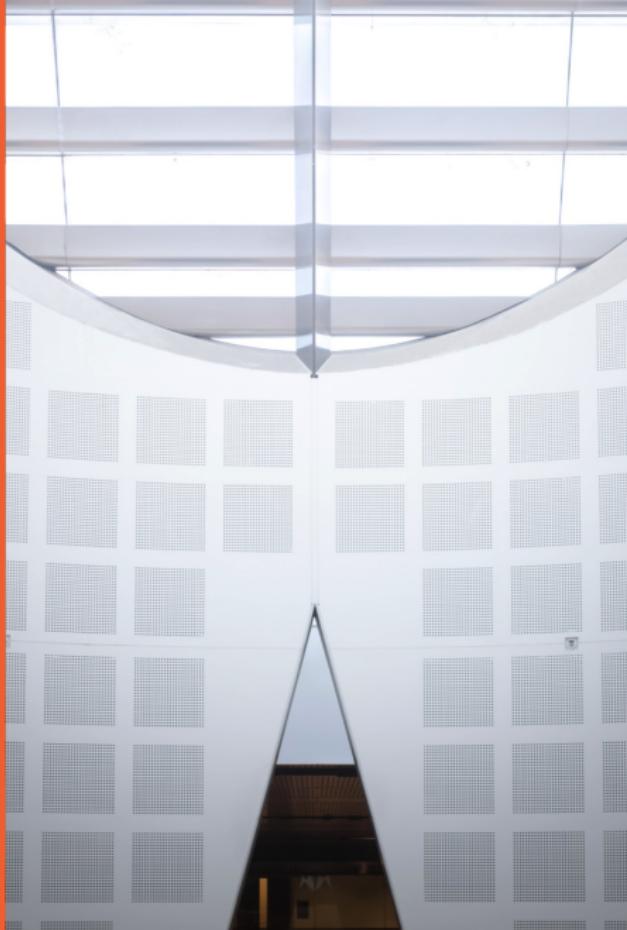
# **COMP2022|2922**

## **Models of Computation**

### **Lesson 1: Propositional Logic**

**Presented by**

Sasha Rubin  
School of Computer Science



# Which logics?

## 1. Propositional logic

A logic for computing with and reasoning about, e.g., statements such as conditionals in programming, digital circuits.

## 2. Predicate logic

A logic for computing with and reasoning about, e.g., the correctness of programs, knowledge-bases in AI, declarative programs, database queries.

# What makes up a logic?

1. **Syntax** tells us the structure of expressions, or the rules for putting symbols together to form an expression.
  - $(1 + (y \times 3))$  is an arithmetic expression/formula
2. **Semantics** refers to the meaning of expressions, or how they are evaluated.
  - It evaluates to 13 if we know that  $y = 4$ .
3. **Deduction** is a syntactic mechanism for deriving new true expressions from existing true expressions.
  - If we know that  $z = x + y$  then we can deduce that  $z = y + x$ .

# Propositional logic

- You can write propositional logic formulas in most programming languages.
- In Python and Java these are called Boolean expressions
- They are usually used as conditions for control flow, e.g., inside an `if` or a `while`.

## Warmup

Do the following two Java Boolean expressions say the same thing?

1. `!(x >= 5 && x != y)`
2. `(x < 5 || x == y)`

## Warmup

Do the following two Java Boolean expressions say the same thing?

1. `!(x >= 5 && x != y)`
2. `(x < 5 || x == y)`

### Note

- They are *syntactically* different (they look different).
- They are *semantically* equivalent (they mean the same thing).
  - In the sense that no matter the values of  $x, y$ , they evaluate to the same thing (i.e., either both true or both false).
- If we know one is true we can *infer* the other is true.
  - Use that `not(A and B)` is equivalent to `(not A or not B)`, no matter what conditions A and B represent.

## Warmup

**Do the following two statements say the same thing?**

1. There are fires outside Sydney and the air in Sydney is polluted.
2. The air in Sydney is polluted and there are fires outside Sydney.

## Warmup

Do the following two statements say the same thing?

1. There are fires outside Sydney and the air in Sydney is polluted.
2. The air in Sydney is polluted and there are fires outside Sydney.

### Note

- We are not asking if the statements are true or not.
- The statements create different images: the first conveys that the fires are causing the pollution, the second that there is already pollution that may be made worse by the fires.

## Warmup

Do the following two statements say the same thing?

1. There are fires outside Sydney and the air in Sydney is polluted.
2. The air in Sydney is polluted and there are fires outside Sydney.

### Note

- We are not asking if the statements are true or not.
- The statements create different images: the first conveys that the fires are causing the pollution, the second that there is already pollution that may be made worse by the fires.
- However, *logically*, the statements are saying the same thing.
- If we know that one is true, we can infer the other is true.
  - The reason for this is that we can use a law that says that " $F$  and  $P$ " means the same as " $P$  and  $F$ ", no matter what  $F$  and  $P$  represent.

## Warmup

**Do the following two statements say the same thing?**

1. If the food is good, then the food is not cheap
2. If the food is cheap, then the food is not good

## Warmup

Do the following two statements say the same thing?

1. If the food is good, then the food is not cheap
2. If the food is cheap, then the food is not good

### Note

- We are not asking if the statements are true or not.
- The statements create different images: Good but not cheap food conveys that it is expensive, while cheap but not good food conveys that it is rotten.

## Warmup

Do the following two statements say the same thing?

1. If the food is good, then the food is not cheap
2. If the food is cheap, then the food is not good

### Note

- We are not asking if the statements are true or not.
- The statements create different images: Good but not cheap food conveys that it is expensive, while cheap but not good food conveys that it is rotten.
- However, *logically*, the statements are saying the same thing, i.e., that this food is not both good and cheap
- If we know that one is true, we can *infer* the other is true.
  - The reason for this is that “if  $G$  then not  $C$ ” means the same as “if  $C$  then not  $G$ ”, no matter what  $G$  and  $C$  represent.

# Propositions

Propositional logic is about statements that can be true or false (although we may not know which). Not every statement can be thought of as a proposition.

**Which of the following are propositions?**

- If this food is good, then this food is not cheap.
- Where are we?
- $2 < 5$

# Propositions

Propositional logic is about statements that can be true or false (although we may not know which). Not every statement can be thought of as a proposition.

## Which of the following are propositions?

Assume that  $i$  is an integer variable, and  $x$  is a Boolean variable.

- $i$
- $i=7$
- $x$
- $x=true$
- $(i < 2 \text{ || } i > 5)$
- $(i < 2 \text{ \&& } x)$

# In a nutshell

## Logic of propositional formulas

### 1. Syntax

$(p \wedge (q \vee p))$  is a propositional logic formula

### 2. Semantics

It evaluates to 0 if  $p = 0, q = 1$ .

### 3. Deduction

If we know the condition  $(p \wedge (q \vee p))$  is true, we can deduce that  $p$  is true.

**Syntax** tells us the structure of expressions, or the rules for putting symbols together to form an expression.

# Syntax

Java and Python each have their own syntax for writing propositional formulas. We will use another syntax which is the standard in computer science and mathematics.

Name	Prop. Logic	Python	Java
conjunction	$\wedge$	and	<code>&amp;&amp;</code>
disjunction	$\vee$	or	<code>  </code>
negation	$\neg$	not	<code>!</code>
implication	$\rightarrow$		
bi-implication	$\leftrightarrow$	<code>==</code>	<code>==</code>
top/verum	$\top$	<code>True</code>	<code>true</code>
bottom/falsum	$\perp$	<code>False</code>	<code>false</code>
atoms	$p, q, r, \dots$	Boolean variables	Boolean variables
formulas	$F, G, H, \dots$	Boolean expressions	Boolean expressions

# Syntax

We now precisely define the syntactic rules for defining formulas of Propositional Logic.

## Definition

- A **propositional atom** (or simply **atom**<sup>1</sup>) is a variable of the form  $p_1, p_2, p_3, \dots$  (or alternatively  $p, q, r, \dots$ )
- A **propositional formula** (or simply **formula**<sup>2</sup>) is defined by the following recursive process:
  1. Every atom is a formula
  2. If  $F$  is a formula then  $\neg F$  is a formula.
  3. If  $F, G$  are formulas then  $(F \vee G)$  as well as  $(F \wedge G)$  are formulas.

---

<sup>1</sup>Also called: propositional variable, atomic proposition, atomic formula.

<sup>2</sup>Also called: Boolean formula.

# Syntax

## Example

Which of the following are formulas?

1.  $\neg p$
2.  $(\neg p)$
3.  $\neg\neg p$
4.  $\neg p \vee q$
5.  $(\neg p \vee q)$
6.  $\neg(p \vee q)$
7.  $(r \wedge \neg(p \vee q))$

# Syntax

Why do we need this mathematical definition of formula?

1. It is unambiguous.
  - If we disagree on whether something is a formula, we just consult the definition
2. It allows one to design algorithms that process formulas using recursion as follows:
  - The base case is rule 1 of the definition
  - The recursive case are rules 2 and 3 of the definition
3. One can prove things about propositional formulas.

# Syntax

A formula which occurs in another formula is called a **subformula**.

## Exercise

Give a recursive process that defines the set **SubForms**( $G$ ) of all subformulas of  $G$ .

# Syntax

## Abbreviations

1.  $(F_1 \rightarrow F_2)$  instead of  $(\neg F_1 \vee F_2)$

The symbol  $\rightarrow$  is called “implication”

$(F_1 \rightarrow F_2)$  is read “ $F_1$  implies  $F_2$ ”

2.  $(F_1 \leftrightarrow F_2)$  instead of  $(F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1)$

The symbol  $\leftrightarrow$  is called “bi-implication”

$(F_1 \leftrightarrow F_2)$  is read “ $F_1$  if and only if  $F_2$ ”

3.  $\perp$  instead of  $(F \wedge \neg F)$

The symbol  $\perp$  is called “bottom” or “falsum”

4.  $\top$  instead of  $(F \vee \neg F)$

The symbol  $\top$  is called “top” or “verum”.

5.  $(\bigvee_{i=1}^n F_i)$  instead of  $(\dots ((F_1 \vee F_2) \vee F_3) \dots \vee F_n)$

6.  $(\bigwedge_{i=1}^n F_i)$  instead of  $(\dots ((F_1 \wedge F_2) \wedge F_3) \dots \wedge F_n)$

# Homework

## Example

Consider the formula  $\neg((p \wedge q) \vee \neg r)$ .

- How do you apply the rules to build this formula? Give the sequence of rules you use to build the following formula.
- This formula has 7 subformulas. What are they?

**Semantics** refers to the meaning of expressions, or how they are evaluated

Semantics of Propositional Logic = Assignments +  
Truth-values of Formulas

## Semantics: truth values

Recall: **Semantics** refers to how you derive the value of a formula based on the values of its atomic subformulas.

- The elements of the set  $\{0, 1\}$  are called **truth values**
- We read 1 as “true”, and 0 as “false”
- After we **assign** truth values to atoms . . .  
(e.g.,  $p = 0, q = 1$ )
- we can **compute** the truth values of formulas.  
(e.g., the truth value of  $(p \vee q)$  is 1).

## Semantics: truth values

Recall: **Semantics** refers to how you derive the value of a formula based on the values of its atomic subformulas.

- The elements of the set  $\{0, 1\}$  are called **truth values**
- We read 1 as “true”, and 0 as “false”
- After we **assign** truth values to atoms . . .  
(e.g.,  $p = 0, q = 1$ )
- we can **compute** the truth values of formulas.  
(e.g., the truth value of  $(p \vee q)$  is 1).

Here are the rules for doing this computation . . .

## Semantics: truth tables

The formula  $\neg F$  is true if and only if the formula  $F$  is false.

$F$	$\neg F$
0	1
1	0

## Semantics: truth tables

The formula  $(F \wedge G)$  is true if and only if both the formulas  $F$  and  $G$  are true.

$F$	$G$	$(F \wedge G)$
0	0	0
0	1	0
1	0	0
1	1	1

## Semantics: truth tables

The formula  $(F \vee G)$  is true if and only if either  $F$  or  $G$  or both are true.

$F$	$G$	$(F \vee G)$
0	0	0
0	1	1
1	0	1
1	1	1

# Semantics

## Example

Evaluate the formula  $F = \neg((p \wedge q) \vee r)$  under the assignment  $p = 1, q = 1, r = 0$ .

$p$	$q$	$r$	$\parallel$	$\neg((p \wedge q) \vee r)$
1	1	0	$\parallel$	

## Semantics

Evaluate the formula  $F = \neg((p \wedge q) \vee r)$  under all assignments of the atoms  $p, q, r$ .

$p$	$q$	$r$	$\neg((p \wedge q) \vee r)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

# Semantics

We now formalise these truth tables with the following definition of the semantics.

## Definition

- An **assignment** is a function  $\alpha$  assigning truth values to atoms.  
So  $\alpha(p)$  is the truth value of  $p$  under assignment  $\alpha$ .
- The **truth value**  $tv(F, \alpha)$  of a formula  $F$  under the assignment  $\alpha$  is given by the following recursive procedure:
  1. The truth value of an atom  $p$  is just  $\alpha(p)$ .
  2. The truth value of a formula of the form  $\neg F$  is 1 if the truth value of  $F$  is 0, and 0 otherwise.
  3. The truth value of a formula of the form  $(F \wedge G)$  is 1 if the truth values of both  $F$  and  $G$  are 1, and 0 otherwise.
  4. The truth value of a formula of the form  $(F \vee G)$  is 1 if the truth values of either  $F$  or  $G$  or both are 1, and 0 otherwise.

# Semantics

We now formalise these truth tables with the following definition of the semantics.

## Definition

- An **assignment** is a function  $\alpha$  assigning truth values to atoms.  
So  $\alpha(p)$  is the truth value of  $p$  under assignment  $\alpha$ .
- The **truth value**  $tv(F, \alpha)$  of a formula  $F$  under the assignment  $\alpha$  is given by the following recursive procedure:

$$1. \ tv(p, \alpha) = \alpha(p) \text{ for every atom } p$$

$$2. \ tv(\neg F, \alpha) = \begin{cases} 0 & \text{if } tv(F, \alpha) = 1 \\ 1 & \text{if } tv(F, \alpha) = 0 \end{cases}$$

$$3. \ tv((F \wedge G), \alpha) = \begin{cases} 1 & \text{if } tv(F, \alpha) = 1 \text{ and } tv(G, \alpha) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$4. \ tv((F \vee G), \alpha) = \begin{cases} 1 & \text{if } tv(F, \alpha) = 1 \text{ or } tv(G, \alpha) = 1 \\ 0 & \text{otherwise} \end{cases}$$

# Semantics

## Example

Suppose  $\alpha(p_1) = 1$  and  $\alpha(p_2) = 0$ . Compute the truth value of  $(\neg p_1 \vee p_2)$  under  $\alpha$ .

# Semantics

Why do we need this precise definition of semantics?

1. It is unambiguous.
  - If we disagree on the truth value of  $F$  under  $\alpha$ , we just consult the definition.
2. This definition is implemented by the runtime environment of Python and Java to compute the values of Boolean expressions.
3. One can prove things about formulas.

# Semantics

## Terminology

- In case the truth value of  $F$  under  $\alpha$  is equal to 1, we say that  $\alpha$  makes  $F$  true or that  $\alpha$  satisfies  $F$ .
- The standard notation in logic for this is different: we say that  $\alpha$  models  $F$ , written:  $\alpha \models F$ .  
The symbol  $\models$  is called the “double-turnstile”.
- A formula  $F$  is satisfiable if at least one assignment satisfies  $F$ . Otherwise  $F$  is unsatisfiable.

## Exercise

Convince yourself that here is an equivalent (and more succinct) way to define the truth value of formulas under an assignment  $\alpha$ :

1.  $\text{tv}(p, \alpha) = \alpha(p)$  for atoms  $p$
2.  $\text{tv}(\neg F, \alpha) = 1 - \text{tv}(F, \alpha)$ .
3.  $\text{tv}((F \wedge G), \alpha) = \min\{\text{tv}(F, \alpha), \text{tv}(G, \alpha)\}$ .
4.  $\text{tv}((F \vee G), \alpha) = \max\{\text{tv}(F, \alpha), \text{tv}(G, \alpha)\}$ .

# Semantics

## Example

Suppose  $\alpha(p_1) = 1$  and  $\alpha(p_2) = 0$ . Compute the truth value of  $(\neg p_1 \vee p_2)$  under  $\alpha$ .

$$\begin{aligned}\text{tv}((\neg p_1 \vee p_2), \alpha) &= \max\{\text{tv}(\neg p_1, \alpha), \text{tv}(p_2, \alpha)\} && \text{Rule 4.} \\ &= \max\{1 - \text{tv}(p_1, \alpha), \text{tv}(p_2, \alpha)\} && \text{Rule 2.} \\ &= \max\{1 - \alpha(p_1), \alpha(p_2)\} && \text{Rule 1.} \\ &= \max\{1 - 1, 0\} = 0\end{aligned}$$

## Semantics: implication

$F$	$G$	$(F \rightarrow G)$
0	0	1
0	1	1
1	0	0
1	1	1

## Semantics: implication

- We now discuss why  $(F \rightarrow G)$  is defined as  $(\neg F \vee G)$ .
- When is the formula  $(F \rightarrow G)$  true?
- It is true whenever, **if**  $F$  is true, **then** also  $G$  is true.
- So, if  $F$  is not true, then it doesn't matter whether  $G$  is true or not, the formula  $(F \rightarrow G)$  will still be true.
- **Note.** Implication is not the same as the programming construct "If condition is true then do instruction". And it does not mean that  $F$  causes  $G$  to be true.

$F$	$G$	$(F \rightarrow G)$
0	0	1
0	1	1
1	0	0
1	1	1

## Semantics: implication

Let's write a function that returns whether or not you are ready to go outside. The constraint is that if it is raining, then, to be ready, you should be carrying\_umbrella.

---

```
1 def ready:  
2     return (not raining or carrying_umbrella)
```

---

If Python had a symbol for implication, say  $\rightarrow$ , then we could have written:

---

```
1 def ready:  
2     return (raining → carrying_umbrella)
```

---

## Connection with natural language

What is the connection between propositional logic and natural language? Although logic can be used to model propositions of natural language, there are important differences.

- In logic, semantics is determined by the assignment and the syntactic structure of the formula.
- In natural language, the semantics also depends on the context.

# Connection with natural language

## Disjunction

- Inclusive or:  $(F \vee G)$
- Exclusive or:  $((F \vee G) \wedge \neg(F \wedge G))$

# Connection with natural language

## Disjunction

- Inclusive or:  $(F \vee G)$
- Exclusive or:  $((F \vee G) \wedge \neg(F \wedge G))$

## Examples

One can reach the airport by taxi or bus	Inclusive
You can choose to save your money or your life	Exclusive
The error is in the program or the sensor data	Exclusive?
The program or the sensor data are erroneous	Inclusive?

# Connection with natural language

## Implication/equivalence

“Eating fast-food is equivalent to aiding the destruction of the world's rainforests”

- This looks like an equivalence,
- but is more likely an implication (it's unlikely that the speaker is claiming that destroying rainforests results in eating fast food).

## For you think about

How does the following important idea apply to syntax and semantics of propositional logic?

*In Computer Science we start with the simplest possible systems, and sets of rules that we haven't necessarily confirmed by experiment, but which we just suppose are true, and then ask what sort of complex systems we can and cannot build... it is a mathematical set of tools, or body of ideas, for understanding just about any system—brain, universe, living organism, or, yes, computer.*

Scott Aaronson — theoretical computer scientist

## References

- Chapter 1 of Schöning covers syntax and semantics of propositional logic
- We use slightly more readable notation:
  - We use  $p, q, r, \dots$  for atoms instead of  $A, B, C, \dots$
  - We use  $\alpha$  for assignments instead of  $\mathcal{A}$
  - We use  $\text{tv}(-)$  for truth values of a formula instead of  $\mathcal{A}$

## Coda: algorithmic issue

**Input:** A propositional formula  $F$

**Output:** Yes if  $F$  is satisfiable, and No otherwise

- Can one check every assignment? There are infinitely many.
- **Fact.** Assignments that agree on the atoms occurring in  $F$  also agree on  $F$ .
- So we only need to check finitely many assignments.
- If  $F$  contains atoms  $p_1, \dots, p_n$ , there are exactly  $2^n$  different assignments. Why?
- Test all of them using truth-tables.

## Coda: algorithmic issue

**Question.** How efficient is this test for satisfiability?

- Suppose  $F$  has  $n$  atoms.
- Then the truth-table has  $2^n$  rows — exponential growth.
- Suppose you can generate a table at the rate of one row per second.
- If  $n = 80$ , you will need  $2^{80}$  seconds to write out the table.
- This is about 2.8 million times the age of the universe.

**Question.** Is there a substantially faster method?

- Probably not.
- This is related to the P vs NP problem. See COMP3027: Algorithm Design.

# **COMP2022|2922**

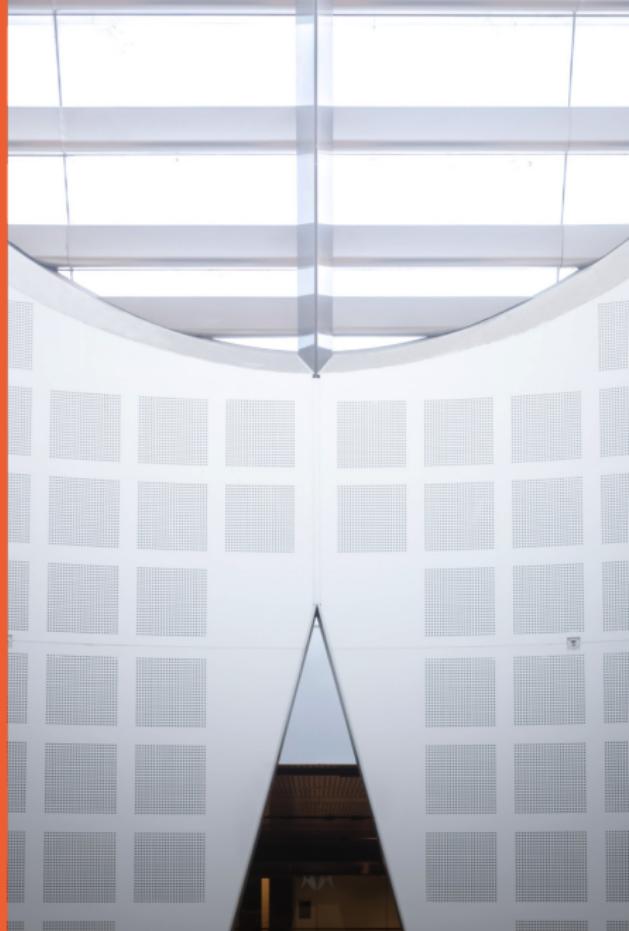
## **Models of Computation**

### **Lesson 2a: Propositional Logic**

### **Equivalences and Consequences**

**Presented by**

Sasha Rubin  
School of Computer Science



Formulas that “mean the same thing” are called **equivalent**. We now study common equivalences, also called **laws**.

# Equivalences

Recall that although the formulas  $(A \wedge B)$  and  $(B \wedge A)$  are syntactically different formulas, they mean the same thing.

Let's make this idea precise.

## Definition

Two formulas  $F$  and  $G$  are **(logically) equivalent** if they are assigned the same truth value under every assignment. This is written  $F \equiv G$ .

# Equivalences

Are the following pairs of formulas equivalent?

1.  $p$  and  $q$ ?
2.  $(p \rightarrow q)$  and  $(q \rightarrow p)$ ?
3.  $(p \rightarrow q)$  and  $(\neg q \rightarrow \neg p)$ ?
4.  $(p \vee \neg p)$  and  $(q \vee \neg q)$ ?

# Some Equivalences

For all formulas  $F, G, H$ :

(Idempotent Laws)

$$F \equiv (F \wedge F)$$

$$F \equiv (F \vee F)$$

(Commutative Laws)

$$(F \wedge G) \equiv (G \wedge F)$$

$$(F \vee G) \equiv (G \vee F)$$

(Associative Laws)

$$(F \wedge (G \wedge H)) \equiv ((F \wedge G) \wedge H)$$

$$(F \vee (G \vee H)) \equiv ((F \vee G) \vee H)$$

$$(F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H))$$

$$(F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H))$$

$$\neg(F \wedge G) \equiv (\neg F \vee \neg G)$$

$$\neg(F \vee G) \equiv (\neg F \wedge \neg G)$$

$$\neg\neg F \equiv F$$

$$(\top \vee F) \equiv \top$$

$$(\top \wedge F) \equiv F$$

$$(\perp \vee F) \equiv F$$

$$(\perp \wedge F) \equiv \perp$$

(de Morgan's Laws)

(Double Negation Law)

(Validity Law)

(Unsatisfiability Law)

## Equivalences

Which equivalence can be used to justify that the following two statements, logically speaking, say the same thing?

1. There are fires outside Sydney and the air in Sydney is polluted.
2. The air in Sydney is polluted and there are fires outside Sydney.

$A$  = “There are fires outside Sydney”

$B$  = “The air in Sydney is polluted”

By the Law of Commutativity of  $\wedge$ , we know that  $(A \wedge B)$  is equivalent to  $(B \wedge A)$ .

# Equivalences

There are a number of ways to verify an equivalence.

1. Use truth tables.
2. Deduce it from other equivalences (using the fact that  $\equiv$  is transitive).

# Equivalences

Verify the following equivalence:

$$(p \rightarrow q) \equiv (\neg q \rightarrow \neg p)$$

... using truth tables

# Equivalences

Verify the following equivalence:

$$(p \rightarrow q) \equiv (\neg q \rightarrow \neg p)$$

... using deduction from other equivalences

$$\begin{aligned} (p \rightarrow q) &\equiv (\neg p \vee q) && \text{Definition of } \rightarrow \\ &\equiv (q \vee \neg p) && \text{Commutative Law for } \vee \\ &\equiv (\neg \neg q \vee \neg p) && \text{Double Negation Law} \\ &\equiv (\neg q \rightarrow \neg p) && \text{Definition of } \rightarrow \end{aligned}$$

**Aside.** What justifies the use of Double Negation inside the formula?

- **Fact.** If  $F$  is a subformula of  $H$ , and  $F \equiv G$ , then  $H$  is equivalent to formulas that result by substituting an occurrence of  $F$  in  $H$  by  $G$ . This is called the Substitution Rule.

## Equivalences: applications (1)

- Equivalences can be used to rewrite a formula into an equivalent one having a special structure, called a **normal form**.
- You will see normal forms in the tutorials/assignments.

## Equivalences: applications (2)

- We defined the syntax of propositional formulas to only use the connectives  $\wedge$ ,  $\vee$  and  $\neg$ .
- Other connectives like  $\rightarrow$  were defined in terms of these ones.
- Could we have made a different choice of connectives when defining the syntax? . . . Yes!

**Fact.** Every propositional formula is equivalent to a formula which only contains the operators  $\neg$  and  $\wedge$ .

- e.g.,  $(\neg p \vee q)$  is equivalent to  $\neg(p \wedge \neg q)$ .

**Fact.** Every propositional formula is equivalent to a formula which only contains the operators  $\neg$  and  $\rightarrow$ .

- e.g.,  $(p \wedge q)$  is equivalent to  $\neg(p \rightarrow \neg q)$ .

Logic allows us to formalise what it means for an argument ("if these facts are true, then that fact must be true") to be correct. This is done with an idea called **logical consequence**.

# Logical consequence

## Definition

Say that  $F$  is a logical consequence of  $\{E_1, \dots, E_k\}$ , if every assignment that makes all of the formulas  $E_i$  (for  $1 \leq i \leq k$ ) true also makes  $F$  true. We write this as

$$\{E_1, \dots, E_k\} \models F$$

# Logical consequence

## Example

Use truth tables to determine whether  $\{p, (p \rightarrow q)\} \models q$ .

# Logical consequence

## Example

Use truth tables to determine whether

$$\{(p \rightarrow r), (q \rightarrow r), (p \vee q)\} \models r.$$

# Logical consequence

## Example

Use truth tables to determine whether  $\{(p \rightarrow q)\} \models (q \rightarrow p)$ .

## Logical consequence: Application

- Logic is used to study correct argumentation and reasoning.
- This is useful for coding, reasoning about the world, mathematics, etc.

### Example

Why is the following argument correct?

1. If  $x = 5$  then  $z = 4$
2.  $x = 5$
3. Conclude  $z = 4$

The reason is that it is a particular instance of the fact that

$$\{(p \rightarrow q), p\} \models q$$

# Logical consequence: Application

## Example

Why is the following argument correct?

1. If  $x = 5$  then  $z = 4$
2. If  $x < 1$  then  $z = 4$
3. Either  $x = 5$  or  $x < 1$
4. Conclude  $z = 4$

The reason is that it is a particular instance of the fact that

$$\{(p \rightarrow r), (q \rightarrow r), (p \vee q)\} \models r$$

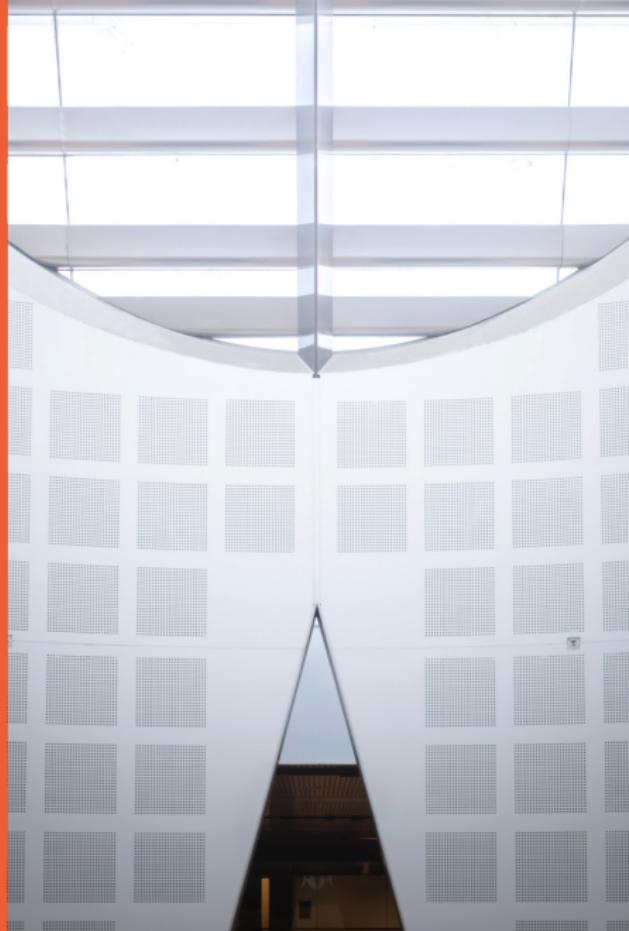
# **COMP2022|2922**

## **Models of Computation**

### **Lesson 2b: Deduction in Propositional Logic**

**Presented by**

Sasha Rubin  
School of Computer Science



**Deduction** is a syntactic mechanism for deriving validities as well as logical consequences from assumptions

# Deduction: motivation

- The most famous deductive system is surely found in Euclid's *Elements* for deducing facts in elementary geometry and number theory.
- Euclid made some **assumptions**  $E_1, \dots, E_k$  about points and lines ...
- ... and produced **formal proofs** that established **logical consequences**  $F$  such as
  1. "the angles in a triangle sum to 180 degrees"
  2. "there are infinitely many prime numbers".

# Deduction: Formal proofs

A deductive system gives you rules for constructing **formal proofs**.

Formal proofs:

1. A highly disciplined way of reasoning (good for computers)
2. A sequence of formulas where each step is a deduction based on earlier steps
3. Based entirely on rewriting formulas – no semantics.

# Deduction: Formal proofs

- A formal proof usually starts with some assumptions.
- Each step creates another formula which is **justified** by applying an **inference rule** to formulas from previous steps.
- If we manage to prove a formula  $F$  from assumptions  $E_1, \dots, E_k$ , then we will write

$$\{E_1, \dots, E_k\} \vdash F$$

which is read  $E_1, \dots, E_k$  proves  $F$ .<sup>1</sup>

---

<sup>1</sup>The name of the symbol  $\vdash$  is called “turnstile”.

$\models$  is a semantic notion

$\vdash$  is a syntactic notion

# Natural deduction

We use a deductive system called **Natural Deduction (ND)**.<sup>2</sup>

1. Every connective has two types of inference rules:
  - **Introduction rules** introduce the connective
  - **Elimination rules** remove the connective
2. There is also a rule to **introduce assumptions**.
3. Some of the connective rules **cancel assumptions**.
4. In this system,  $\rightarrow$  and  $\perp$  are not abbreviations.
5. If we collect all the assumptions  $\{E_1, \dots, E_k\}$  that have not been cancelled at the end of the derivation, and if  $F$  is the last formula in the proof, then we write

$$\{E_1, \dots, E_k\} \vdash F$$

---

<sup>2</sup>Developed by Gerhard Gentzen (20th century German logician), and Stanisław Jaśkowski (20th century Polish logician).

# Natural deduction

Here is one of the rules:  $(\wedge \text{ E}) \quad \frac{\mathcal{S} \vdash (A \wedge B)}{\mathcal{S} \vdash A}$

Let's analyse it:

1. The name of the rule is  $(\wedge \text{ E})$  which is read “Conjunction Elimination”.
2.  $\mathcal{S}$  denotes a **set** of formulas, and  $A, B$  denote formulas.
3. We use this rule as follows: if we have proven  $(A \wedge B)$  using the formulas in  $\mathcal{S}$ , then we can apply this rule to get a proof of  $A$  that uses the formulas in  $\mathcal{S}$ .
4. This rule formalises the following type of reasoning: if we have a proof of  $(A \wedge B)$ , then we have a proof of  $A$ .

# Natural deduction

Let's look at another rule:  $(\wedge I)$  
$$\frac{\mathcal{S}_1 \vdash A \quad \mathcal{S}_2 \vdash B}{\mathcal{S}_1 \cup \mathcal{S}_2 \vdash (A \wedge B)}$$

Let's analyse it:

1. The name of the rule is  $(\wedge I)$  which is read “Conjunction Introduction”.
2.  $\mathcal{S}_1$  and  $\mathcal{S}_2$  denote sets of formulas, and  $A, B$  denote formulas.
3. We use this rule as follows: if we have proven  $A$  using the formulas in  $\mathcal{S}_1$ , and we have proven  $B$  using the formulas in  $\mathcal{S}_2$ , then we can apply this rule to get a proof of  $(A \wedge B)$  that uses the formulas in  $\mathcal{S}_1 \cup \mathcal{S}_2$ .
4. This rule formalises following type of reasoning: if we have a proof of  $A$  and we have a proof of  $B$ , then we have a proof of  $(A \wedge B)$ .

# Natural deduction: formal proofs

Here is how we will write a formal proof in ND.

1. Write the *initial assumptions* (sometimes called *premises*).
2. Devise a sequence of formulas, in which the last one is the desired *conclusion*.
3. Each step in the sequence must be:
  - *line numbered* so that we can refer to it later in the proof,
  - annotated with the line numbers of the *assumptions* which that step depends on,
  - *justified* by the name of inference rule,
  - annotated with the line numbers *referenced* by the justification.

Line	Assumptions	Formula	Justification	References

# Natural deduction: formal proofs

Each line of the proof means the following:

If we know the *assumptions* hold, then we conclude the *formula* holds, because it can be *justified* by applying rule to the *referenced* lines above it.

Line	Assumptions	Formula	Justification	References
:	:	:	:	:
8	1,2,4	$\neg C$	( $\wedge$ E)	4
:	:	:	:	:

# Natural deduction: Rules involving $\wedge$

$(\wedge E)$	$\frac{\mathcal{S} \vdash (A \wedge B)}{\mathcal{S} \vdash A}$
$(\wedge E)$	$\frac{\mathcal{S} \vdash (A \wedge B)}{\mathcal{S} \vdash B}$
$(\wedge I)$	$\frac{\mathcal{S}_1 \vdash A \quad \mathcal{S}_2 \vdash B}{\mathcal{S}_1 \cup \mathcal{S}_2 \vdash (A \wedge B)}$

- The  $\wedge E$  rules formalise the following reasoning: if we have a proof of  $(A \wedge B)$ , then we have a proof of  $A$  and we have a proof of  $B$ .
- The  $\wedge I$  rule formalises the following reasoning: if we have a proof of  $A$  and we have a proof of  $B$ , then we have a proof of  $(A \wedge B)$ .

# Natural deduction: Proof involving $\wedge$

$$\{(F \wedge G)\} \vdash (G \wedge F)$$

Line	Assumptions	Formula	Justification	References
1	1	$(F \wedge G)$	Asmp. I	
2	1	$F$	$\wedge E$	1
3	1	$G$	$\wedge E$	1
4	1	$(G \wedge F)$	$\wedge I$	2,3

# Natural deduction: Introducing an assumption

$$(\text{Asmp. I}) \quad \frac{}{\{F\} \vdash F}$$

- This rule allows one to introduce a formula  $F$  as an assumption, without reference to earlier lines.
- Its only assumption is itself.

## Natural deduction: rules involving $\rightarrow$

$$\begin{array}{c} (\rightarrow E) \frac{\mathcal{S}_1 \vdash A \quad \mathcal{S}_2 \vdash (A \rightarrow B)}{\mathcal{S}_1 \cup \mathcal{S}_2 \vdash B} \\ (\rightarrow I) \frac{\mathcal{S} \cup \{A\} \vdash B}{\mathcal{S} \vdash (A \rightarrow B)} \end{array}$$

- The  $(\rightarrow E)$  rule formalises Modus Ponens.
- The  $(\rightarrow I)$  rule formalises that if we have a proof of  $B$  from  $A$ , then we have a proof of “if  $A$  then  $B$ ”.

# Natural deduction: proofs involving $\rightarrow$

The following consequence is called *Hypothetical Syllogism (HS)*:

$$\{(A \rightarrow B), (B \rightarrow C)\} \vdash (A \rightarrow C)$$

Line	Assumptions	Formula	Justification	References
1	1	$(A \rightarrow B)$	Asmp. I	
2	2	$(B \rightarrow C)$	Asmp. I	
3	3	$A$	Asmp. I	
4	1,3	$B$	$\rightarrow E$	1,3
5	1,2,3	$C$	$\rightarrow E$	2,4
6	1,2	$(A \rightarrow C)$	$\rightarrow I$	3,5

## Natural deduction: rules involving $\vee$

$$(\vee I) \quad \frac{\mathcal{S} \vdash A}{\mathcal{S} \vdash (A \vee B)}$$

$$(\vee I) \quad \frac{\mathcal{S} \vdash A}{\mathcal{S} \vdash (B \vee A)}$$

$$(\vee E) \quad \frac{\mathcal{S}_1 \vdash (A \vee B) \quad \mathcal{S}_2 \cup \{A\} \vdash C \quad \mathcal{S}_3 \cup \{B\} \vdash C}{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3 \vdash C}$$

- The  $\vee I$  rule formalises the following reasoning: if we have a proof of  $A$ , then we have a proof of  $(A \vee B)$ , and similarly we have a proof of  $(B \vee A)$ , no matter what  $B$  is.
- The  $\vee E$  rule formalises “reasoning by cases”: if we have a proof of  $(A \vee B)$ , a proof of  $C$  from  $A$ , and a proof of  $C$  from  $B$ , then we have a proof of  $C$ .

# Natural deduction: proof involving $\vee$

$$\{(A \vee B), (A \rightarrow C), (B \rightarrow C)\} \vdash C$$

Line	Assumptions	Formula	Justification	References
1	1	$(A \vee B)$	Asmp. I	
2	2	$(A \rightarrow C)$	Asmp. I	
3	3	$(B \rightarrow C)$	Asmp. I	
4	4	$A$	Asmp. I	
5	2,4	$C$	$\rightarrow E$	2,4
6	6	$B$	Asmp. I	
7	3,6	$C$	$\rightarrow E$	3,6
8	1,2,3	$C$	$\vee E$	1,4,5,6,7

In line 8, we use  $(\vee E)$  with  $S_1 = \{(A \vee B)\}$ ,  $S_2 = \{(A \rightarrow C)\}$ ,  $S_3 = \{(B \rightarrow C)\}$ .

# Natural deduction: proofs

The following consequence is called *Constructive Dilemma (CD)*:

$$\{((A \rightarrow B) \wedge (C \rightarrow D)), (A \vee C)\} \vdash (B \vee D)$$

Line	Assumptions	Formula	Justification	References
1	1	$(A \rightarrow B)$ $(C \rightarrow D)$	$\wedge$ Asmp. I	
2	2	$(A \vee C)$	Asmp. I	
3	1	$(A \rightarrow B)$	$\wedge E$	1
4	1	$(C \rightarrow D)$	$\wedge E$	1
5	5	$A$	Asmp. I	
6	1,5	$B$	$\rightarrow E$	3,5
7	1,5	$(B \vee D)$	$\vee I$	6
8	8	$C$	Asmp. I	
9	1,8	$D$	$\rightarrow E$	4,8
10	1,8	$(B \vee D)$	$\vee I$	9
11	1,2	$(B \vee D)$	$\vee E$	2,5,7,8,10

## Natural deduction: rules involving $\neg$

$$(\neg E) \quad \frac{\mathcal{S}_1 \vdash A \quad \mathcal{S}_2 \vdash \neg A}{\mathcal{S}_1 \cup \mathcal{S}_2 \vdash \perp}$$

$$(\neg I) \quad \frac{\mathcal{S} \cup \{A\} \vdash \perp}{\mathcal{S} \vdash \neg A}$$

- The  $(\neg E)$  rule formalises that  $\perp$  follows from any contradiction.
- The  $(\neg I)$  rule formalises that we have a proof of  $\perp$  from  $A$ , then we have a proof of  $\neg A$ .

# Natural deduction: proofs

$$\{F\} \vdash \neg\neg F$$

Line	Assumptions	Formula	Justification	References
1	1	$F$	Asmp. I	
2	2	$\neg F$	Asmp. I	
3	1,2	$\perp$	$\neg E$	1,2
4	1	$\neg\neg F$	$\neg I$	2,3

To show

$$\emptyset \vdash F \rightarrow \neg\neg F$$

we can continue the proof above with one more line:

5		$F \rightarrow \neg\neg F$	$\rightarrow I$	1,4
---	--	----------------------------	-----------------	-----

# Natural deduction: proofs

The following consequence is called *Modus Tollens*:

$$\{(F \rightarrow G), \neg G\} \vdash \neg F$$

Line	Assumptions	Formula	Justification	References
1	1	$(F \rightarrow G)$	Asmp. I	
2	2	$\neg G$	Asmp. I	
3	3	$F$	Asmp. I	
4	1,3	$G$	$\rightarrow E$	1,3
5	1,2,3	$\perp$	$\neg E$	2,4
6	1,2	$\neg F$	$\neg I$	3,5

## Natural deduction: another rule

$$(\perp) \frac{\mathcal{S} \vdash \perp}{\mathcal{S} \vdash A}$$

- The  $(\perp)$  rule formalises that from a false assumption, anything can be derived.

# Natural deduction: proofs

$$\{(F \vee G), \neg G\} \vdash F$$

Line	Assumptions	Formula	Justification	References
1	1	$(F \vee G)$	Asmp. I	
2	2	$\neg G$	Asmp. I	
3	3	$G$	Asmp. I	
4	2,3	$\perp$	$\neg E$	2,3
5	2,3	$F$	$\perp$	4
6	6	$F$	Asmp. I	
7	1,2	$F$	$\vee E$	1,3,5,6,6

We apply  $(\vee E)$  using  $S_1 = \{(F \vee G)\}, S_2 = \emptyset, S_3 = \{\neg G\}$

## Natural deduction: last rule

$$(RA) \quad \frac{\mathcal{S} \cup \{\neg A\} \vdash \perp}{\mathcal{S} \vdash A}$$

- The (RA) rule is called *Reductio ad absurdum* or *Reduction to the absurd*. It formalises the method of proof by contradiction: if the assumption that  $A$  is false (i.e., that  $\neg A$  is true) leads to a contradiction, then  $A$  must be true.

# Natural deduction: proofs

$$\{\neg\neg F\} \vdash F$$

Line	Assumptions	Formula	Justification	References
1	1	$\neg\neg F$	Asmp. I	
2	2	$\neg F$	Asmp. I	
3	1,2	$\perp$	$\neg E$	1,2
4	1	$F$	RA	2,3

We apply (RA) using  $\mathcal{S} = \{\neg\neg F\}$ .

# Natural Deduction: wrap-up

We should definitely ask two questions *about* Natural Deduction:

1. Can it prove only logical consequences? Such a system is called **sound**.
2. Can it prove all logical consequences? Such a system is called **complete**.

Roughly speaking:

1. A deductive system that is not sound might give us wrong results.
  - E.g., it might have a proof of  $(B \rightarrow A)$  from  $(A \rightarrow B)$ .
2. A deductive system that is not complete might not give us all the right results.
  - E.g., it might not have a proof of  $\neg\neg A$  from  $A$ .

## Theorem

Natural deduction for propositional logic is sound and complete.

## For you think about

How does the following idea apply to Natural Deduction for propositional logic?

*In Computer Science we start with the simplest possible systems, and sets of rules that we haven't necessarily confirmed by experiment, but which we just suppose are true, and then ask what sort of complex systems we can and cannot build... it is a mathematical set of tools, or body of ideas, for understanding just about any system—brain, universe, living organism, or, yes, computer.*

Scott Aaronson — theoretical computer scientist

# COMP2022|2922

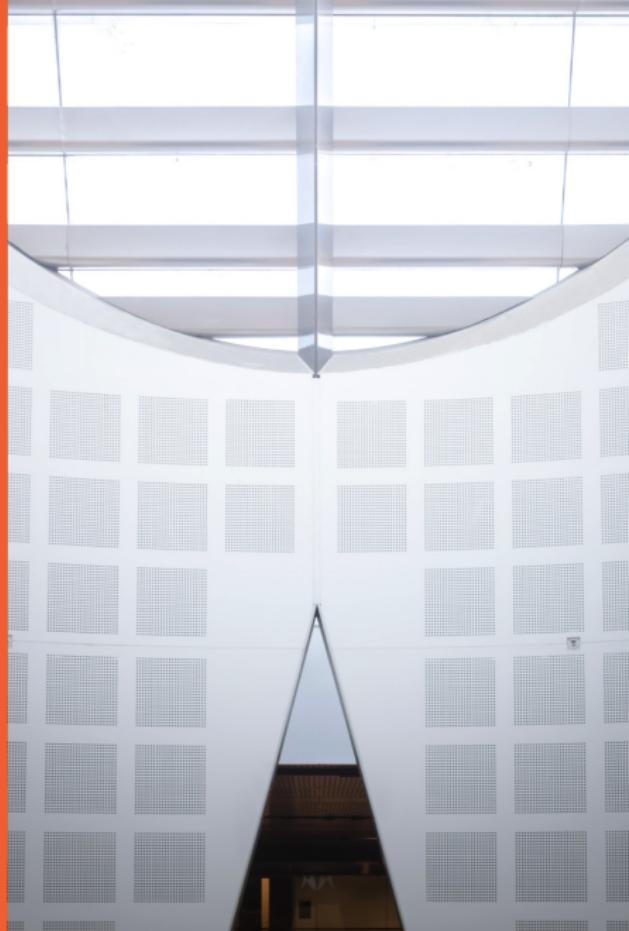
## Models of Computation

### Lesson 3: Predicate Logic

**Presented by**

Sasha Rubin

School of Computer Science



The world has many objects, some of which are related to each other. These can be **modeled** by predicate logic.

In predicate logic we will be able to represent formulas like the following:<sup>1</sup>

$$\forall x \forall y (x + y = y + x)$$

This is not a proposition since whether or not it is true depends on missing information: the type of the variables  $x, y$  and the meaning of the symbol  $+$ .

- If  $x$  and  $y$  are integer variables and  $+$  means ordinary addition, then the formula is true.
- If  $x$  and  $y$  are string variables and  $+$  means concatenation, then the formula is false.

"hello" + "world" != "world" + "hello"

---

<sup>1</sup>The symbol  $\forall$  is read “for all”.

So, in order to give meaning to formulas of predicate logic, we first have to fix the possible values that variables can take, as well as the interpretation of symbols like +.

Structure = Domain + Functions + Predicates + Constants

# Background: domains

When we reason about objects, we have in mind a certain domain of discourse.

1. In programming, the domain may be the integers, or strings, or both, etc.
2. In the world, the domain includes people, animals, etc.

For us, a **domain** is simply a non-empty set  $D$ .

- The set  $\mathbb{Z}$  of integers is a domain.
- The set  $\mathbb{S}$  of binary strings is a domain.
- The set  $\mathbb{H}$  of humans is a domain.

In predicate logic, variables  $x, y, z, \dots$  vary over elements of the domain.

# Background: functions

Mappings from several objects in a domain to a single object of the domain are called **functions**.

1. Here are some functions in the domain  $\mathbb{Z}$ : add, double.
2. Here are some functions in the domain  $\mathbb{S}$ : concat, reverse.
3. Here is a function in the domain  $\mathbb{H}$ : `mother_of`  
So `mother_of(x)` is the mother of  $x$ .

The number of arguments of a function is called its **arity**.

By the way, it is traditional in logic to use notation like  $f(x, y)$  or even  $\text{add}(x, y)$  instead of infix notation like  $x + y$ .

## Background: predicates

Properties of objects in a domain, as well as relations between objects in a domain, are called **predicates** (sometimes also called **relations**).

1. Here are some predicates in the domain  $\mathbb{Z}$ : `is_even`, `is_odd`, `is_bigger_than`.
2. Here is a predicate in the domain  $\mathbb{S}$ : `is_longer_than`.
3. Here is a predicate in the domain of  $\mathbb{H}$ : `loves`  
`loves(x,y)` says that  $x$  loves  $y$ .

## Background: arity

- Predicates (like functions) take some number of arguments, and so we write `is_even(x)` and `is_bigger_than(x,y)`.
- The number of arguments is called the **arity** of the predicate.
- `is_even` takes one argument, so it is called a **unary** predicate.
- `is_bigger_than` takes two arguments, so it is called a **binary** predicate.

## Background: constants

Single elements of the domain are called **constants**.<sup>2</sup>

1. Here are some constants in the domain  $\mathbb{Z}$ : 0, 1, 2, 3 and  $-3$ .
2. Here are some constants in the domain  $\mathbb{S}$ : 0, 1, 10 and 11.
3. Here are some constants in the domain  $\mathbb{H}$ : Sasha and Paul.

---

<sup>2</sup>They may also be seen as functions of arity zero, i.e., functions that take no arguments and always return the same element.

# Background: structures

## Definition

A **structure**  $\mathcal{A}$  consists of a domain  $D$ , functions on  $D$ , predicates on  $D$ , and constants from  $D$ .

## Examples

- $\mathcal{Z} = (\mathbb{Z}, \text{add}, \text{eq}, 0, 1)$  where  $\mathbb{Z}$  is the set of integers,  $\text{add}$  is integer addition (a binary function),  $\text{eq}$  is equality of integers (a binary predicate), and  $0, 1$  are constants.
- $\mathcal{S} = (\mathbb{S}, \text{add}, \text{eq}, 0, 1)$  where  $\mathbb{S}$  is the set of binary strings,  $\text{add}$  is string concatenation (a binary function),  $\text{eq}$  is equality of strings (a binary predicate), and  $0, 1$  are constants.
- $\mathcal{H} = (\mathbb{H}, \text{loves}, \text{mother\_of})$  where  $\text{loves}$  is a binary predicate and  $\text{mother\_of}$  is a unary function.

# Background: structures

## Important notation

- We have conflicting notation: e.g., which function is add?  
Integer addition or string concatenation?
- To distinguish these, we may *superscript* with the name of the structure: so  $\text{add}^{\mathbb{Z}}$  is the integer-addition function, and  $\text{add}^{\mathcal{S}}$  is the string-concatenation function.
- So, if we want to be precise, we should write the structures before as:

$$\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathbb{Z}}, \text{eq}^{\mathbb{Z}}, 0^{\mathbb{Z}}, 1^{\mathbb{Z}})$$

and

$$\mathcal{S} = (\mathbb{S}, \text{add}^{\mathcal{S}}, \text{eq}^{\mathcal{S}}, 0^{\mathcal{S}}, 1^{\mathcal{S}})$$

# Background: structures

## Important notation

- So if add is not a function, what is it, and how do we use it?
- It is just a **name**, it is just a symbol.
- Just as different people may have the same name, so too different functions ( $\text{add}^{\mathbb{Z}}$  and  $\text{add}^S$ ) can have the same name (add).
- Why do we need the names at all? We can use names in formulas of predicate logic to talk about multiple structures!
  - Consider the formula  $\forall x \forall y \text{eq}(\text{add}(x, y), \text{add}(y, x))$  that says that addition is commutative.
  - This formula makes sense about integers **and** about strings!
  - It happens to be true about the integers but false about strings.

# Plan

## What makes up a logic?

1. **Syntax** tells us the structure of expressions, or the rules for putting symbols together to form an expression.
2. **Semantics** refers to the meaning of expressions, or how they are evaluated.
3. **Deduction** is a syntactic mechanism for deriving new true expressions from existing true expressions.

# Syntax + Semantics: In a nutshell

The **Predicate-Logic Formulas** for a structure  $\mathcal{A}$  are built from:

1. symbols for the functions, predicates and constants of  $\mathcal{A}$ ;  
called the **vocabulary** of  $\mathcal{A}$ ,
2. the usual connectives  $\neg, \wedge, \vee$ ,
3. variables  $x, y, z, \dots$ , and
4. the quantifier symbols  $\forall$  (read "for all") and  $\exists$  (read "exists").

# Syntax + Semantics: In a nutshell

What is the vocabulary of the following structures?

1. The vocabulary of  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$  consists of add, eq, 0, 1
2. The vocabulary of  $\mathcal{S} = (\mathbb{S}, \text{add}^{\mathcal{S}}, \text{eq}^{\mathcal{S}}, 0^{\mathcal{S}}, 1^{\mathcal{S}})$  consists of add, eq, 0, 1
3. The vocabulary of  $\mathcal{H} = (\mathbb{H}, \text{loves}^{\mathcal{H}}, \text{mother\_of}^{\mathcal{H}})$  consists of loves, mother\_of.

## Syntax + Semantics: In a nutshell

- Variables  $x, y, z, \dots$  vary over elements in the domain  $D$ .
- Quantifier symbols  $\forall, \exists$  allow one to talk about their quantity.
- Universal quantifier:  $\forall$  “for every”, “for all”
- Existential quantifier:  $\exists$  “there exists”, “some”

# Syntax + Semantics: In a nutshell

## Example

Here are some formulas and their meaning in the structure

$$\mathcal{H} = (\mathbb{H}, \text{loves}^{\mathcal{H}}, \text{mother\_of}^{\mathcal{H}}).$$

$\forall x \text{loves}(x, x)$	Every human loves themself
$\forall x \text{loves}(x, \text{mother\_of}(x))$	Everyone loves their mother
$\forall x \text{loves}(\text{mother\_of}(x), x)$	Everyone's mother loves them
$\forall x \forall y \text{loves}(x, y)$	Everyone loves everyone
$\exists x \forall y \text{loves}(x, y)$	Someone loves everyone
$\forall x \exists y \text{loves}(x, y)$	Everyone loves someone
$\exists x \exists y \text{loves}(x, y)$	Someone loves someone
$\forall x \exists y \text{loves}(y, \text{mother\_of}(x))$	Everyone's mother is loved by someone

**Syntax** tells us the structure of expressions, or the rules for putting symbols together to form an expression.

Syntax of Predicate Logic = Symbols + Terms + Formulas

## Syntax: symbols

Suppose structure  $\mathcal{A}$  has functions  $f^{\mathcal{A}}, g^{\mathcal{A}}, h^{\mathcal{A}}, \dots$ , predicates  $P^{\mathcal{A}}, Q^{\mathcal{A}}, R^{\mathcal{A}}, \dots$ , and constants  $a^{\mathcal{A}}, b^{\mathcal{A}}, c^{\mathcal{A}}, \dots$

Then predicate logic formulas for  $\mathcal{A}$  are built from the following **symbols**:

1. The vocabulary of  $\mathcal{A}$ , i.e.,
  - 1.1 Function symbols  $f, g, h, \dots$
  - 1.2 Predicate symbols  $P, Q, R, \dots$
  - 1.3 Constants symbols  $a, b, c, \dots$
2. Connectives  $\neg, \wedge, \vee$  (as in Propositional Logic)
3. Variables  $u, v, w, x, y, z, \dots$
4. Quantifier symbols  $\forall, \exists$

As we have seen, we may use more suggestive symbols like add (instead of  $f$ ) and eq (instead of  $P$ ).

# Syntax: terms

A term refers to an object without naming it

- $f(2)$
- $\text{father}(\text{Alan})$  Alan's father
- $\text{round\_up}(2.5)$  The smallest integer  $\geq 2.5$

Terms may have variables

- $f(x)$
- $\text{father}(x)$   $x$ 's father
- $\text{round\_up}(x)$  The smallest integer  $\geq x$

Terms may be composed

- $f(g(x, y))$
- $\text{father}(\text{father}(\text{Alan}))$  Alan's father's father
- $\text{half}(\text{round\_up}(x))$  Half of the smallest integer  $\geq x$

# Syntax: terms

Here is a precise definition.

## Definition

**Terms** are defined by the following recursive process:

1. Every variable is a term.
2. Every constant symbol is a term.
3. If  $f$  is a  $k$ -ary function symbol, and if  $t_1, \dots, t_k$  are terms, then  $f(t_1, \dots, t_k)$  is a term.

## Examples

- So, if  $f$  is unary,  $g$  is binary, and  $c$  is a constant, then the following are terms:  $x$ ,  $c$ ,  $f(x)$ ,  $g(c, f(y))$ ,  $f(f(y))$ ,  $g(f(x), g(x, y))$ .
- In the language of  $\mathcal{Z}$ , the following are terms:  $\text{add}(x, 1)$ ,  $\text{add}(x, \text{add}(x, y))$ ,  $\text{add}(0, \text{add}(1, \text{add}(1, x)))$ .

# Syntax: formulas

We can finally give the syntax of predicate logic.

## Definition

- An **atomic formula (of predicate logic)** has the form  $P(t_1, \dots, t_k)$  where  $P$  is a  $k$ -ary predicate symbol and  $t_1, \dots, t_k$  are terms.
- A **formula (of predicate logic)** is defined by the following recursive process:
  1. Every atomic formula is a formula.
  2. If  $F$  is a formula then  $\neg F$  is a formula.
  3. If  $F, G$  are formulas then  $(F \wedge G)$  and  $(F \vee G)$  are formulas.
  4. If  $x$  is a variable and  $F$  is a formula then  $\exists x F$  and  $\forall x F$  are formulas.

As for propositional logic, we use the same shorthands, i.e.,  
 $(F \rightarrow G)$ ,  $(F \leftrightarrow G)$ ,  $(\bigwedge_i F_i)$ ,  $(\bigvee_i F_i)$ ,  $\top$ ,  $\perp$ .

If  $F$  is a formula and  $F$  occurs as part of the formula  $G$  then  $F$  is called a **subformula** of  $G$ .

# Syntax: formulas

## Example

If  $f$  is binary,  $P$  is binary, then the following are formulas:

- $P(f(x, y), f(y, x))$
- $\forall x P(f(x, y), f(y, x))$
- $\forall x \forall y (P(f(x, y), f(y, x)) \vee \neg P(x, x))$

# Syntax: free and bound occurrences of variables

- An occurrence of the variable  $x$  in the formula  $F$  is **bound** if  $x$  occurs within a subformula of  $F$  of the form  $\exists xG$  or  $\forall xG$ . Otherwise it is a **free** occurrence.
  - Programming analogy: bound/local, free/global
- A variable may have both free and bound occurrences in a formula  $F$ .
- Let  $\text{Free}(F)$  be the set of all variables that occur free in  $F$ .
- A formula without free variables (i.e.,  $\text{Free}(F) = \emptyset$ ) is called a **sentence**.

## Example

$$\forall x (P(x, y) \rightarrow \exists y Q(x, y, z))$$

**Semantics** refers to the meaning of expressions, or how they are evaluated

Semantics of Predicate Logic = Structures + Assignments +  
Values of Terms + Truth-values of Formulas

# Semantics: assignments

## Definition

Let  $\mathcal{A}$  be a structure with domain  $D$ . An **assignment** is a function  $\alpha$  that maps variables  $x, y, z, \dots$  to elements of the domain  $D$ .

- So an assignment  $\alpha$  gives every variable a value in the domain.
- This means that  $\alpha$  can also be seen to give every term a value in the domain.

# Semantics: values of terms

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ .

- Suppose assignment  $\alpha$  maps  $x$  to 1 and  $y$  to 2, i.e.,  
 $\alpha(x) = 1, \alpha(y) = 2$ .
- Then  $\alpha$  gives the term  $\text{add}(x, y)$  the integer value 3.
- And  $\alpha$  gives the term  $\text{add}(x, \text{add}(x, y))$  the integer value 4.

# Semantics: values of terms

## Definition

The **value** of the term  $t$  under assignment  $\alpha$  in the structure  $\mathcal{A}$  is defined by the following recursive process:

1. If  $t$  is a variable, say  $x$ , then its value is just  $\alpha(x)$ .
2. If  $t$  is a constant symbol, say  $c$ , then its value is  $c^{\mathcal{A}}$ .
3. If  $t$  is a term of the form  $f(t_1, \dots, t_k)$  and  $t_1, \dots, t_k$  are terms with values  $d_1, \dots, d_k$ , then its value is  $f^{\mathcal{A}}(d_1, \dots, d_k)$ .

# Semantics: values of terms

We can write this more concisely:

## Definition

The **value** of the term  $t$  under assignment  $\alpha$  in the structure  $\mathcal{A}$ , denoted  $\text{val}(t, \alpha, \mathcal{A})$ , is defined by the following recursive process:

1.  $\text{val}(x, \alpha, \mathcal{A}) = \alpha(x)$ .
2.  $\text{val}(c, \alpha, \mathcal{A}) = c^{\mathcal{A}}$ .
3.  $\text{val}(f(t_1, \dots, t_k), \alpha, \mathcal{A}) = f^{\mathcal{A}}(\text{val}(t_1, \alpha, \mathcal{A}), \dots, \text{val}(t_k, \alpha, \mathcal{A}))$ .

## Notation

- If the structure  $\mathcal{A}$  is clear from context, write  $\text{val}(t, \alpha)$ .
- If also the assignment  $\alpha$  is clear from context, write  $\text{val}(t)$ .

# Semantics: values of terms

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ .

- Suppose assignment  $\alpha$  maps  $x$  to 1 and  $y$  to 2.
- What is the value of the term  $\text{add}(x, \text{add}(x, y))$ ?

$$\begin{aligned}\text{val}(\text{add}(x, \text{add}(x, y))) &= \text{add}^{\mathcal{Z}}(\text{val}(x), \text{val}(\text{add}(x, y))) && 3. \\ &= \text{add}^{\mathcal{Z}}(\text{val}(x), \text{add}^{\mathcal{Z}}(\text{val}(x), \text{val}(y))) && 3. \\ &= \text{add}^{\mathcal{Z}}(\alpha(x), \text{add}^{\mathcal{Z}}(\alpha(x), \alpha(y))) && 1. \\ &= \text{add}^{\mathcal{Z}}(1, \text{add}^{\mathcal{Z}}(1, 2)) \\ &= 1 + (1 + 2) = 4\end{aligned}$$

# Semantics: truth-values of atomic-formulas

We can now assign truth values to atomic formulas  $P(t_1, \dots, t_k)$ .

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ . Suppose  $\alpha(x) = 2, \alpha(y) = 3$ .

- What is the truth value of the following atomic formula?

$$\text{eq}(\text{add}(y, x), \text{add}(x, y))$$

- To answer this, we must ask if

$$(\text{val}(\text{add}(y, x)), \text{val}(\text{add}(x, y))) \in \text{eq}^{\mathcal{Z}}$$

- But this is just asking if  $3 + 2 = 2 + 3$ , which is true, so the truth value is 1.

# Semantics: truth-values of formulas

## Definition

The **truth value** of a formula  $F$  under the assignment  $\alpha$  in the structure  $\mathcal{A}$ , denoted  $\text{TV}(F, \alpha, \mathcal{A})$  or just  $\text{TV}(F, \alpha)$ , is defined by the following recursive process:

1. For an atomic formula  $P(t_1, \dots, t_k)$ ,

$$\text{TV}(P(t_1, \dots, t_k), \alpha) = \begin{cases} 1 & \text{if } (\text{val}(t_1, \alpha), \dots, \text{val}(t_k, \alpha)) \in P^{\mathcal{A}} \\ 0 & \text{otherwise.} \end{cases}$$

# Semantics: truth-values of formulas

## Definition

The **truth value** of a formula  $F$  under the assignment  $\alpha$  in the structure  $\mathcal{A}$ , denoted  $\text{TV}(F, \alpha, \mathcal{A})$  or just  $\text{TV}(F, \alpha)$ , is defined by the following recursive process:

$$2. \text{ TV}(\neg F, \alpha) = \begin{cases} 0 & \text{if } \text{TV}(F, \alpha) = 1 \\ 1 & \text{if } \text{TV}(F, \alpha) = 0 \end{cases}$$

$$3. \text{ TV}((F \wedge G), \alpha) = \begin{cases} 1 & \text{if } \text{TV}(F, \alpha) = 1 \text{ and } \text{TV}(G, \alpha) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$4. \text{ TV}((F \vee G), \alpha) = \begin{cases} 1 & \text{if } \text{TV}(F, \alpha) = 1 \text{ or } \text{TV}(G, \alpha) = 1 \\ 0 & \text{otherwise} \end{cases}$$

# Semantics: truth-values of formulas

## Definition

The **truth value** of a formula  $F$  under the assignment  $\alpha$  in the structure  $\mathcal{A}$ , denoted  $\text{TV}(F, \alpha, \mathcal{A})$  or just  $\text{TV}(F, \alpha)$ , is defined by the following recursive process:

5. Define  $\text{TV}(\forall xG, \alpha) = 1$  if for every element  $d$  in the domain of  $\mathcal{A}$ ,  $\text{TV}(G, \alpha[x \mapsto d]) = 1$ , and 0 otherwise.
6. Define  $\text{TV}(\exists xG, \alpha) = 1$  if there is an element  $d$  in the domain of  $\mathcal{A}$  such that  $\text{TV}(G, \alpha[x \mapsto d]) = 1$ , and 0 otherwise.

$\alpha[x \mapsto d]$  is the assignment which is identical to  $\alpha$  except that it maps the variable  $x$  to  $d$ . This can be expressed by the equation:

$$\alpha[x \mapsto d](y) = \begin{cases} \alpha(y) & \text{if } y \neq x \\ d & \text{if } y = x \end{cases}$$

# Semantics: truth-values of formulas

We can write the definition more succinctly:

## Definition

The **truth value** of a formula  $F$  under the assignment  $\alpha$  in the structure  $\mathcal{A}$ , denoted  $\text{TV}(F, \alpha, \mathcal{A})$  or just  $\text{TV}(F, \alpha)$ , is defined by the following recursive process:

1.  $\text{TV}(P(t_1, \dots, t_k), \alpha) = 1$  if  $(\text{val}(t_1, \alpha), \dots, \text{val}(t_k, \alpha)) \in P^{\mathcal{A}}$ , and 0 otherwise.
2.  $\text{TV}(\neg F, \alpha) = 1 - \text{TV}(F, \alpha)$
3.  $\text{TV}((F \wedge G), \alpha) = \min\{\text{TV}(F, \alpha), \text{TV}(G, \alpha)\}$
4.  $\text{TV}((F \vee G), \alpha) = \max\{\text{TV}(F, \alpha), \text{TV}(G, \alpha)\}$
5.  $\text{TV}(\forall x G, \alpha) = \min\{\text{TV}(G, \alpha[x \mapsto d]) : d \in D\}$
6.  $\text{TV}(\exists x G, \alpha) = \max\{\text{TV}(G, \alpha[x \mapsto d]) : d \in D\}$

# Semantics

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ . Let  $\alpha$  be *any* assignment. Compute  $\text{TV}(\exists xG, \alpha)$  where  $G$  is the formula  $\text{eq}(x, \text{add}(x, x))$ .

You can think of this as a loop.

# Semantics

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ . Let  $\alpha$  be *any* assignment. Compute  $\text{TV}(\exists xG, \alpha)$  where  $G$  is the formula  $\text{eq}(x, \text{add}(x, x))$ .

- By the rule for  $\exists$ ,  $\text{TV}(\exists xG, \alpha) = 1$  iff there is some  $d \in \mathbb{Z}$ :

$$\text{TV}(G, \alpha[x \mapsto d]) = 1$$

- By the rule for predicates, this means that

$$(\text{val}(x, \alpha[x \mapsto d]), \text{val}(\text{add}(x, x), \alpha[x \mapsto d])) \in \text{eq}^{\mathcal{Z}}$$

- By evaluating the terms, this means that  $(d, d + d) \in \text{eq}^{\mathcal{Z}}$ .
- By evaluating the predicate, this means that  $d = d + d$ .
- Is there such an integer  $d$ ? Yes,  $d = 0$ .
- So  $\text{TV}(\exists xG, \alpha) = 1$ .

# Semantics

## Example

Recall the structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ . Let  $\alpha$  be *any* assignment. Compute  $\text{TV}(\forall x \forall y G, \alpha)$  where  $G$  is the formula  $\text{eq}(\text{add}(x, y), \text{add}(y, x))$ .

- By the rule for  $\forall$  (applied twice),  $\text{TV}(\forall x \forall y G, \alpha) = 1$  iff for all  $d, e \in \mathbb{Z}$ :  $\text{TV}(G, \alpha[x, y \mapsto d, e]) = 1$ .
- By the rule for predicates, this means that

$$(\text{val}(\text{add}(x, y), \alpha[x, y \mapsto d, e]), \text{val}(\text{add}(y, x), \alpha[x, y \mapsto d, e])) \in \text{eq}^{\mathcal{Z}}$$

- By evaluating the terms, this means that  $(d + e, e + d) \in \text{eq}^{\mathcal{Z}}$ .
- By evaluating the predicate, this means that  $d + e = e + d$ .
- Is this true for all integers  $d, e$ ? Yes.
- So  $\text{TV}(\forall x \forall y G, \alpha) = 1$ .

# Semantics

- Intuitively, the truth value of a formula  $F$  does not depend on  $\alpha(z)$  if the variable  $z$  does not occur in  $F$ .
- More is true:  
**Fact.** The truth value  $\text{TV}(F, \alpha)$  only depends on  $\alpha(z)$  if  $z \in \text{Free}(F)$ .
- In the previous examples the formula  $F$  had no free variables. So the truth value of  $F$  does not depend at all on the  $\alpha$  we **start** with.
  - Of course, as we evaluated the formula, we updated the assignment and the formula, and so some variables became free and the current assignment did matter.

## Semantics: sentences

- A formula without free occurrences of variables (i.e.,  $\text{Free}(F) = \emptyset$ ) is called a **sentence**.
- **Fact.** The truth value of a sentence does not depend on the variable assignment.
- So, for a sentence  $F$  and a structure  $\mathcal{A}$ , if the truth-value is 1 under some assignment, then it is 1 under every assignment.
- In this case we say that  **$F$  is true in  $\mathcal{A}$** .

# Semantics

So in the previous examples, we have that the following formulas are true in the integer structure  $\mathcal{Z} = (\mathbb{Z}, \text{add}^{\mathcal{Z}}, \text{eq}^{\mathcal{Z}}, 0^{\mathcal{Z}}, 1^{\mathcal{Z}})$ :

- the formula  $\exists x \text{ eq}(x, \text{add}(x, x))$ , which in ordinary notation is written  $\exists x (x = x + x)$ ;
- the formula  $\forall x \forall y \text{ eq}(\text{add}(x, y), \text{add}(y, x))$  which in ordinary notation is written  $\forall x \forall y (x + y = y + x)$ .

# Semantics: satisfiable sentences

A sentence  $F$  is **satisfiable** if it is true in some structure  $\mathcal{A}$ .

## Example

- The formula  $\forall x \forall y \text{eq}(\text{add}(x, y), \text{add}(y, x))$  is satisfiable (since it is true in the structure  $\mathcal{Z}$ ).
- The formula  $\exists x (\text{eq}(x, x) \wedge \neg \text{eq}(x, x))$  is not satisfiable.

## For you to think about

We saw that we can use truth-tables to decide if a given propositional logic formula is satisfiable. Is there an algorithm that decides if a given predicate logic formula is satisfiable?

# Reference

- Chapter 2 of Schöning covers Predicate Logic
  - NB. We use the same syntax, but slightly different notation for semantics.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 4a: Equivalences of Predicate Logic**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



Formulas that “mean the same thing” are called **equivalent**. We now study common equivalences, also called **laws**.

# Equivalences

## Definition

Two formulas  $F$  and  $G$  are **(logically) equivalent** if they are assigned the same truth value in every structure under every assignment. This is written  $F \equiv G$ .

**Fact.** All equivalences which have been proved for formulas in propositional logic also hold for formulas of predicate logic.

## Example

- De Morgan's Law:  $\neg(F \wedge G) \equiv (\neg F \vee \neg G)$  holds for all formulas  $F, G$  of predicate logic.
- E.g.,  $\neg(\exists x P(x) \wedge Q(y)) \equiv (\neg \exists x P(x) \vee \neg Q(y))$ .

# Equivalences involving quantifiers

For all formulas  $F, G$ :

(Q. Negation)

$$\neg \forall x F \equiv \exists x \neg F$$

$$\neg \exists x F \equiv \forall x \neg F$$

(Q. Unification)

$$(\forall x F \wedge \forall x G) \equiv \forall x (F \wedge G)$$

$$(\exists x F \vee \exists x G) \equiv \exists x (F \vee G)$$

(Q. Transposition)

$$\forall x \forall y F \equiv \forall y \forall x F$$

$$\exists x \exists y F \equiv \exists y \exists x F$$

(Q. Extraction)

if  $x \notin \text{Free}(G)$  :

$$(\forall x F \wedge G) \equiv \forall x (F \wedge G)$$

$$(\forall x F \vee G) \equiv \forall x (F \vee G)$$

$$(\exists x F \wedge G) \equiv \exists x (F \wedge G)$$

$$(\exists x F \vee G) \equiv \exists x (F \vee G)$$

# Equivalences

Here are informal reasons behind some of these equivalences:<sup>1</sup>

1.  $\neg\forall x F \equiv \exists x \neg F$

- the LHS says that not all  $x$  satisfy  $F$ ,
- which means the same thing as some  $x$  doesn't satisfy  $F$ ,
- which means that some  $x$  does satisfy  $\neg F$ ,
- which is what the RHS says.

2.  $(\forall x F \wedge \forall x G) \equiv \forall x(F \wedge G)$

- the LHS says that  $F$  holds for every  $x$  and  $G$  holds for every  $x$ ,
- which is the same as saying both  $F$  and  $G$  hold for every  $x$ ,
- which is what the RHS says.

3.  $\forall x \forall y F \equiv \forall y \forall x F$

- Both sides say that  $F$  holds for all values of the listed variables.

4.  $(\forall x F \wedge G) \equiv \forall x(F \wedge G)$  if  $x \notin \text{Free}(G)$

- LHS says  $F$  holds for every  $x$ , and  $G$  holds.
- RHS says  $F$  and  $G$  hold for every  $x$ ; but  $G$  doesn't depend on the value of  $x$ .

---

<sup>1</sup>To do prove them formally, use the inductive definition of truth-value.

# Equivalences

## Example

Show that  $\neg(\exists x P(x, y) \vee \forall z \neg R(z)) \equiv \forall x \exists z (\neg P(x, y) \wedge R(z))$

$$\begin{aligned}\neg(\exists x P(x, y) \vee \forall z \neg R(z)) &\equiv (\neg \exists x P(x, y) \wedge \neg \forall z \neg R(z)) && \text{DeMorgan's Laws} \\ &\equiv (\forall x \neg P(x, y) \wedge \exists z \neg \neg R(z)) && \text{Quantifier Negation} \\ &\equiv (\forall x \neg P(x, y) \wedge \exists z R(z)) && \text{Double Negation} \\ &\equiv \forall x (\neg P(x, y) \wedge \exists z R(z)) && \text{Quantifier Extraction} \\ &\equiv \forall x (\exists z R(z) \wedge \neg P(x, y)) && \text{Comm. } \wedge \\ &\equiv \forall x \exists z (R(z) \wedge \neg P(x, y)) && \text{Quantifier Extraction} \\ &\equiv \forall x \exists z (\neg P(x, y) \wedge R(z)) && \text{Comm. } \wedge\end{aligned}$$

# Equivalences

To show that  $F_1 \not\equiv F_2$  we should find a counterexample, i.e., a structure  $\mathcal{A}$  and assignment such that

$$\text{TV}(F_1, \alpha, \mathcal{A}) \neq \text{TV}(F_2, \alpha, \mathcal{A}).$$

**Show that**  $(\forall x P(x) \wedge Q(x)) \not\equiv \forall x(P(x) \wedge Q(x))$

Here is a counter-example:

- The structure  $\mathcal{A} = (A, P^{\mathcal{A}}, Q^{\mathcal{A}})$  has domain  $A = \{1, 2\}$ , predicates  $P^{\mathcal{A}} = \{1, 2\}, Q^{\mathcal{A}} = \{2\}$ ,
  - the assignment  $\alpha$  maps variable  $x$  to domain element 2.
1. Then  $\text{TV}((\forall x P(x) \wedge Q(x)), \alpha, \mathcal{A}) = 1$  (since every element of  $A$  is in  $P$  and  $\alpha(x)$  is in  $Q$ ).
  2. And  $\text{TV}(\forall x(P(x) \wedge Q(x)), \alpha, \mathcal{A}) = 0$  (since not every element of  $A$  is in  $P$  and in  $Q$ ).

This also shows why we can't drop  $x \notin \text{Free}(G)$  for Quantifier Extraction.

# Equivalences

## Example

Show that  $(\forall x P(x) \wedge Q(x)) \equiv \forall y(P(y) \wedge Q(x))$ .

$$\begin{aligned} (\forall x P(x) \wedge Q(x)) &\equiv (\forall y P(y) \wedge Q(x)) && \text{rename bound variable} \\ &\equiv \forall y(P(y) \wedge Q(x)) && \text{Quantifier Extraction} \end{aligned}$$

- Bound variables are like dummy variables, and so one can rename them as follows.
  - Let  $F$  be a formula in which the variable  $z$  does not occur in  $F$ . The **renaming** of variable  $x$  by  $z$  in  $F$  is the formula  $F'$  that results from  $F$  by simultaneously replacing all occurrences of  $x$ , that are **bound by the same occurrence of a quantifier**, by the variable  $z$ . We say that  $F'$  is a **renaming** of  $F$ .
- **Fact.** Renamings preserve the formula up to logical equivalence.
  - In other words, renamings only change the syntax, not the semantics (truth-value) of a formula.

# Equivalences: renaming

## Examples

1.  $\exists z Q(z, y)$  is a renaming of  $\exists x Q(x, y)$ .
2.  $(\exists x P(x) \vee \exists z Q(z, x))$  is not a renaming of  $(\exists w P(w) \vee \exists z Q(z, x))$  since  $x$  already occurred in the formula.
3.  $(\exists z P(z) \wedge \forall z Q(z))$  is not a renaming of  $(\exists x P(x) \wedge \forall x Q(x))$  since a renaming only applies to variables bound by a single quantifier.

# Application of equivalences

Every formula can be transformed into an equivalent one that has a rigid structure, called a **normal form**.

# Normal forms

We will look at two normal forms.

1. Negation normal form (NNF)
2. Prenex normal form (PNF)

# Normal forms: NNF

## Definition

A formula  $F$  is in **negation normal form (NNF)** if negations only occur immediately in front of atomic formulas.

## Theorem

For every formula  $F$  there is an equivalent formula in NNF.

## Algorithm (“push negations inwards”)

Substitute in  $F$  every occurrence of a subformula of the form  $\neg\neg G$  by  $G$ , and

$$\neg\forall x F \text{ by } \exists x \neg F$$

$$\neg\exists x F \text{ by } \forall x \neg F$$

$$\neg(G \wedge H) \text{ by } (\neg G \vee \neg H)$$

$$\neg(G \vee H) \text{ by } (\neg G \wedge \neg H)$$

until no such subformulas occur, and return the result.

Why is this algorithm correct?

# Normal forms: NNF

## Example

Put  $\neg\exists x(P(x) \wedge \neg\exists yQ(y, x))$  into NNF.

# Normal forms: PNF

## Definition

A formula  $F$  is in **prenex normal form (PNF)** if it has the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n F$$

where each  $Q_i \in \{\exists, \forall\}$  is a quantifier symbol, the  $x_i$ s are variables,  $n \geq 0$  (so, there may be no quantifiers in the prefix), and  $F$  does not contain a quantifier.

## Example

- $\forall x(\neg P(x) \vee \exists y(N(y) \wedge L(x, y)))$  is not in PNF.
- $\forall x \exists y(\neg P(x) \vee (N(y) \wedge L(x, y)))$  is in PNF.

# Normal forms: PNF

## Theorem

For every formula  $F$  there is an equivalent formula in PNF.

## Algorithm (“pull quantifiers out the front”)

1. Put  $F$  in NNF, call the result  $F'$ .
2. Substitute in  $F'$  every occurrence of a subformula of the form

$$(\forall x F \wedge G) \text{ by } \forall x(F \wedge G)$$

$$(\forall x F \vee G) \text{ by } \forall x(F \vee G)$$

$$(\exists x F \wedge G) \text{ by } \exists x(F \wedge G)$$

$$(\exists x F \vee G) \text{ by } \exists x(F \vee G)$$

until no such subformulas occur (use commutativity to handle  $(G \wedge \forall x F)$ , etc.), and return the result.

3. NB. To apply these equivalences we need that  $x \notin \text{Free}(G)$ .  
This can always be achieved by renaming the bound variable  $x$ .

# Normal forms: PNF

Put the following into PNF

- $(Q(x) \vee \forall x R(x, x))$

# Normal forms: PNF

Put the following into PNF

$$- (\forall y P(z, y) \vee \exists w Q(w))$$

$$- (\forall y P(z, y) \vee \exists y Q(y))$$

# Logical consequence

## Definition

A sentence  $F$  is a **logical consequence**<sup>2</sup> of the set  $\{E_1, \dots, E_k\}$  of sentences if every structure  $\mathcal{A}$  in which all of the  $E_1, \dots, E_k$  are true, also  $F$  is true. In this case we write

$$\{E_1, \dots, E_k\} \models F$$

or simply

$$E_1, \dots, E_k \models F$$

## Example

- $\forall x R(x, x)$  is a logical consequence of  $\{\forall x \forall y R(x, y)\}$ .
- $P(c)$  is a logical consequence of  $\{Q(c), \forall x (Q(x) \rightarrow P(x))\}$ .

---

<sup>2</sup>This definition mimics the one for Propositional Logic.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 4b: Deductive system for Predicate Logic**

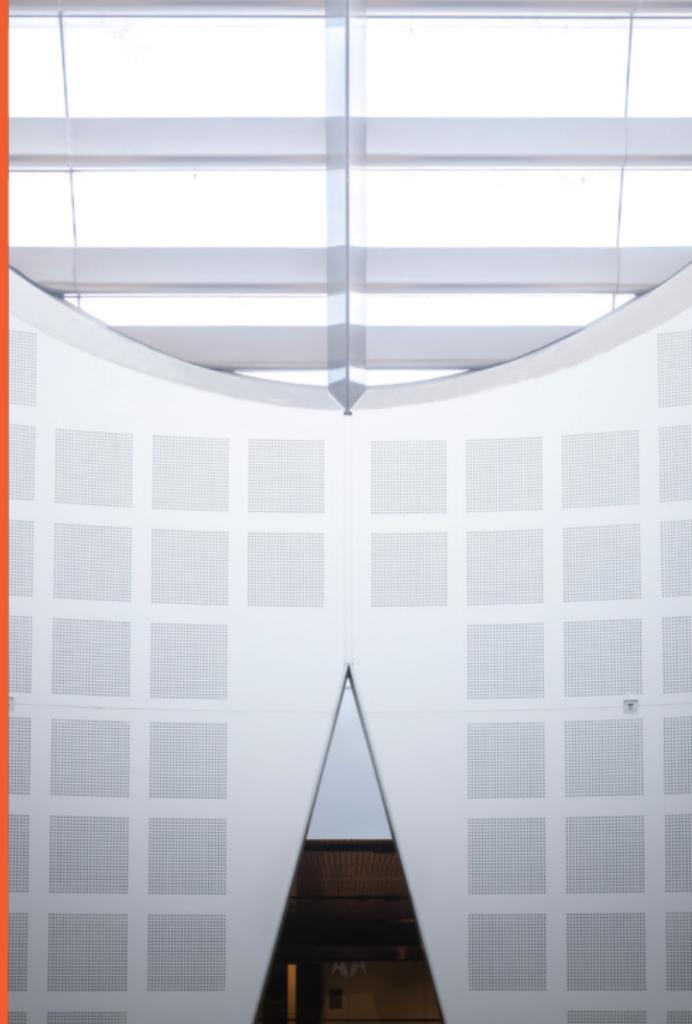
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



Deductive systems are a syntactic mechanism for deriving validities as well as logical consequences from assumptions

# Natural deduction

- We extend ND for propositional logic with rules to handle quantifiers.
- Each quantifier symbol  $\exists, \forall$  has two types of rules:
  1. **Introduction rules** introduce the quantifier
  2. **Elimination rules** remove the quantifier

# Natural Deduction: replacing free variables by terms

## Definition

For a formula  $F$ , variable  $x$ , term  $t$ , we can obtain a formula  $F[t/x]$  by simultaneously replacing all **free** occurrences of  $x$  in  $F$  by  $t$ .

- The idea is that whatever  $F$  said about  $x$ , now  $F[t/x]$  says about  $t$ .
- In a few slides, we will restrict when we are allowed to make such replacements.

# Natural Deduction: inference rules involving $\forall$

$(\forall E)$	$\frac{S \vdash \forall x F}{S \vdash F[t/x]}$ <p>where <math>t</math> is a term that is free to replace <math>x</math> in <math>F</math>.</p>
$(\forall I)$	$\frac{S \vdash F[c/x]}{S \vdash \forall x F}$ <p>where <math>c</math> is a constant symbol, not occurring in <math>F</math>, nor in any formula in <math>S</math>.</p>

$(\forall E)$  formalises the reasoning

*If we know that  $F$  holds for every  $x$ , then it must hold, in particular, taking  $x = t$ .*

# Natural Deduction: inference rules involving $\forall$

$(\forall E)$	$\frac{S \vdash \forall x F}{S \vdash F[t/x]}$ <p>where <math>t</math> is a term that is free to replace <math>x</math> in <math>F</math>.</p>
$(\forall I)$	$\frac{S \vdash F[c/x]}{S \vdash \forall x F}$ <p>where <math>c</math> is a constant symbol, not occurring in <math>F</math>, nor in any formula in <math>S</math>.</p>

$(\forall I)$  formalises the reasoning

*Let  $c$  be any element ... [prove  $F[c/x]$ ]. Since  $c$  was arbitrary, deduce  $F$  holds for all  $x$ .*

That  $c$  is arbitrary is captured by requiring that  $c$  is not in the assumptions used to prove  $F[c/x]$ , and so  $c$  is not constrained in any way.

# Natural Deduction: example using $\forall$

$$\forall x \forall y P(x, y) \vdash \forall y \forall x P(x, y)$$

Line	Assumptions	Formula	Justification	References
1	1	$\forall x \forall y P(x, y)$	Asmp. I	
2	1	$\forall y P(c, y)$	$\forall E$	1
3	1	$P(c, d)$	$\forall E$	2
4	1	$\forall x P(x, d)$	$\forall I^*$	3
5	1	$\forall y \forall x P(x, y)$	$\forall I^{**}$	4

\* the constant  $c$  does not occur in  $F$  (i.e.,  $P(x, d)$ ), nor in the formula of its assumption (in line 1).

\*\* the constant  $d$  does not occur in  $F$  (i.e.,  $\forall x P(x, y)$ ), nor in the formula of its assumption (in line 1).

The conditions on ( $\forall E$ ) that talk about "free to replace" will be explained next.

# Natural Deduction: free to replace

## Example

Say  $F$  is the formula  $\exists y(x < y)$  expressing, about numbers, that there is some number bigger than  $x$ . Then the formula  $F[t/x]$  taking  $t = z + 1$  is  $\exists y(z + 1 < y)$  which says that there is some number bigger than  $z + 1$ .

## Note

The formula  $F[t/x]$  taking  $t = y + 1$  is the formula  $\exists y(y + 1 < y)$  which says that there is some number bigger than its successor.

- What went wrong? We changed  $x$  from being free to  $y + 1$  where  $y$  is bound.
- We disallow such substitutions.

If no variable  $y$  of  $t$  is in the scope of a quantifier  $Qy$  in  $F[t/x]$  then we say that  **$t$  is free to replace  $x$  in  $F$** .

# Natural Deduction: example using $\forall$

$$\forall x \forall y P(x, y) \vdash \forall y \forall x P(x, y)$$

Line	Assumptions	Formula	Justification	References
1	1	$\forall x \forall y P(x, y)$	Asmp. I	
2	1	$\forall y P(c, y)$	$\forall E^*$	1
3	1	$P(c, d)$	$\forall E^{**}$	2
4	1	$\forall x P(x, d)$	$\forall I$	3
5	1	$\forall y \forall x P(x, y)$	$\forall I$	4

\* The term  $c$  is free to replace  $x$  in  $F = \forall y P(y, x)$ .

\*\* The term  $d$  is free to replace  $y$  in  $F = P(c, y)$ .

# Natural Deduction: example using $\forall$

$$\forall x(P(x) \wedge Q(x)) \vdash (\forall xP(x) \wedge \forall xQ(x))$$

Line	Assumptions	Formula	Justification	References
1	1	$\forall x(P(x) \wedge Q(x))$	Asmp. I	
2	1	$(P(c) \wedge Q(c))$	$\forall E^*$	1
3	1	$P(c)$	$\wedge E$	2
4	1	$\forall xP(x)$	$\forall I^*$	3
5	1	$Q(c)$	$\wedge E$	2
6	1	$\forall xQ(x)$	$\forall I^*$	5
7	1	$(\forall xP(x) \wedge \forall xQ(x))$	$\wedge I$	4,6

\* Check that the conditions in lines 2,4,6 are satisfied.

## Natural Deduction: inference rule $\exists I$

$(\exists I)$	$\frac{S \vdash F[t/x]}{S \vdash \exists x F}$ <p style="margin-top: 10px;">where <math>t</math> is a term that is free to replace <math>x</math> in <math>F</math>.</p>
---------------	--

$(\exists I)$  formalises the reasoning

*If we know that  $F$  holds for a specific term  $t$ , then we know it holds for some  $x$ .*

# Natural Deduction: example using $\exists I$

$$\forall x P(x) \vdash \exists x P(x)$$

Line	Assumptions	Formulas	Just.	Ref.
1	1	$\forall x P(x)$	Asmp. I	
2	1	$P(c)$	$\forall E$	1
3	1	$\exists x P(x)$	$\exists I^*$	2

\* the constant  $c$  is free to replace  $x$  in  $P(x)$

## Natural Deduction: inference rule $\exists E$

$(\exists E)$	$\frac{S_1 \vdash \exists x F \quad S_2 \cup \{F[c/x]\} \vdash G}{S_1 \cup S_2 \vdash G}$
	where $c$ is a constant symbol, not occurring in $F$ , nor in $G$ , nor in any formula in $S_2$ .

$(\exists E)$  formalises the reasoning

*If we prove  $G$  from  $F[c/x]$ , but we did not use any other properties of  $c$ , then we can prove  $G$  from the weaker assumption that **some**  $x$  satisfies  $F$ .*

## Natural Deduction: inference rule $\exists E$

$(\exists E)$	$\frac{S_1 \vdash \exists x F \quad S_2 \cup \{F[c/x]\} \vdash G}{S_1 \cup S_2 \vdash G}$
	where $c$ is a constant symbol, not occurring in $F$ , nor in $G$ , nor in any formula in $S_2$ .

How to use  $(\exists E)$ ?

1. **Assume**  $F[c/x]$  ensuring that  $c$  does not occur in  $F$ .
2. Derive  $G$  making sure that  $c$  is not in the assumption set of  $G$  except for  $F[c/x]$ .
3. **Cancel** the assumption  $F[c/x]$ , and conclude  $G$ .

# Natural Deduction: example using $\exists E$

$$\forall x(Q(x) \rightarrow P(y)), \exists x Q(x) \vdash P(y)$$

Line	Assumptions	Formulas	Just.	Ref.
1	1	$\forall x(Q(x) \rightarrow P(y))$	Asmp. I	
2	2	$\exists x Q(x)$	Asmp. I	
3	1	$(Q(c) \rightarrow P(y))$	$\forall E$	1
4	4	$Q(c)$	Asmp. I	
5	1,4	$P(y)$	$\rightarrow E$	3,4
6	1,2	$P(y)$	$\exists E$	2,4,5

# Natural Deduction: incorrect usage of $(\exists E)$

$$\exists x P(x) \vdash \forall x P(x) \quad \dots?$$

Line	Asmp.	Form.	Just.	Ref.
1	1	$\exists x P(x)$	Asmp. I	
2	1	$P(c)$	$\exists E$	? , ?, ?
3	1	$\forall x P(x)$	$\forall I$	2

Here is the faulty argument in natural language:

*We are given that some  $x$  has  $F$ . Let  $c$  be such an  $x$ . Since  $c$  was chosen arbitrarily (?!), conclude that every  $x$  has  $F$ .*

# Natural Deduction: e.g. using $\exists$ and $\forall$

$$\neg \exists x \neg P(x) \vdash \forall x P(x)$$

Line	Asmp.	Form.	Just.	Ref.
1	1	$\neg \exists x \neg P(x)$	Asmp. I	
2	2	$\neg P[c/x]$	Asmp. I	
3	2	$\exists x \neg P(x)$	$\exists I^*$	2
4	1,2	$\perp$	$\neg E$	1,3
5	1	$P[c/x]$	RA	2,4
6	1	$\forall x P(x)$	$\forall I^{**}$	5

\*  $c$  is free to replace  $x$  in  $\neg P(x)$

\*\*  $c$  does not appear in  $P(x)$  nor in the assumption 1

# Deductive system: wrapping up

- We introduced Natural Deduction (ND) for Predicate Logic as a way to derive logical consequences in a way that can be checked by machine.
- Just like for Propositional Logic, ND for Predicate Logic is sound (meaning it can only derive logical consequences) and complete (meaning that it can derive all logical consequences).
- However, unlike Propositional Logic, there is no algorithm that can decide, given  $E_1, \dots, E_k, F$  whether or not  $F$  is a consequence of  $E_1, \dots, E_k$ .
  - We will discuss this when we cover undecidability (Lec 10).
- This means that finding proofs of Predicate Logic inherently require human ingenuity.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 5a: Introduction to Machine Models**

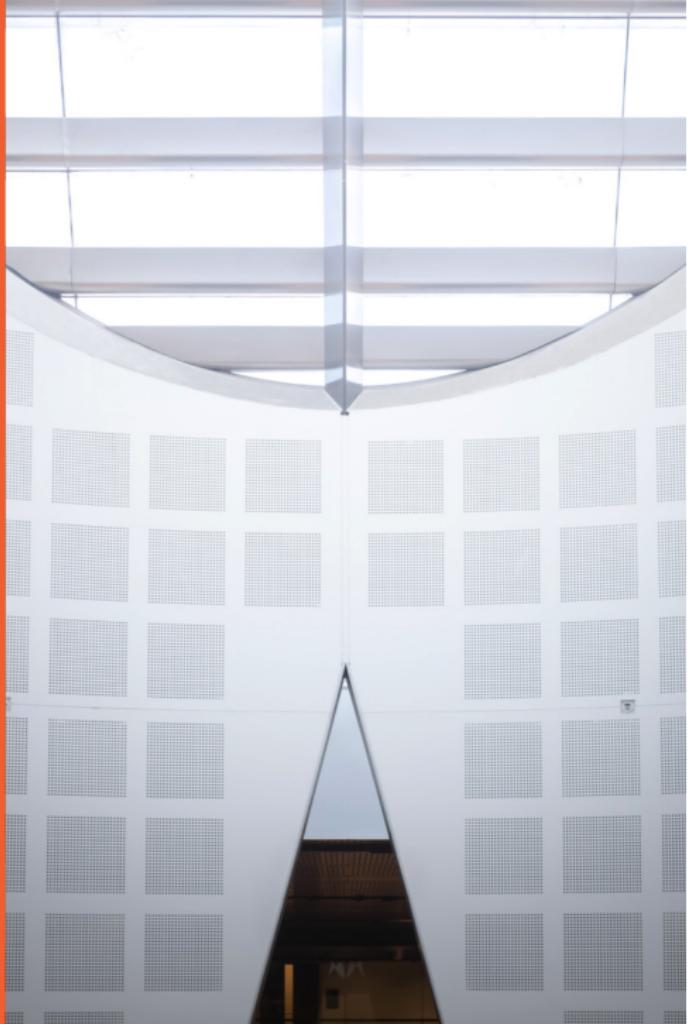
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# What sort of problems are programs meant to solve?

- A **computational problem** specifies what the input can be and what the output should be.
  1. Given numbers  $x, y$ , output  $x + y$ ?
  2. Given numbers  $x, y$ , output  $x \times y$ ?
  3. Given strings  $w, t$ , decide if  $w$  is a substring of  $t$ ?
  4. Given a string  $e$ , decide if  $e$  is well-bracketed?
- The last two are called **decision problems** because their output is either 1 (meaning Yes) or 0 (meaning No).

# Main modeling assumption

We will focus on decision problems where the input is a string.

## For you to think about

1. Is focusing on **decision problems** a serious restriction?

What if we replaced the problem

Given numbers  $x, y$ , output  $x + y$ .

by the decision problem

Given numbers  $x, y, z$ , decide if  $x + y = z$ .

2. Is focusing on input **strings** a serious restriction?

- We can encode almost anything as a string.
- How would you encode an integer, or a set of integers, or a graph?

# Strings

## Definition

An **alphabet**  $\Sigma$  is a nonempty finite set of symbols.

## Example

- $\Sigma = \{0, 1\}$  is the classic **binary alphabet**.
- $\Sigma = \{a, b, c, \dots, z\}$  is the lower-case English alphabet.

# Strings

## Definition

A **string over  $\Sigma$**  is a finite sequence of symbols from  $\Sigma$ . The number of symbols in a string is its **length**.

## Example

- 0110 is a string of length 4 over alphabet  $\{0, 1\}$
- *bob* is a string of length 3 over alphabet  $\{a, b, c, \dots, z\}$ .
- If  $w$  has length  $n$  we write  $w = w_1w_2 \cdots w_n$  where each  $w_i \in \Sigma$ .
- There is only one string of length 0. It is denoted  $\epsilon$ .
- The set of all strings over  $\Sigma$  is denoted  $\Sigma^*$ .
  - The reason for this notation will be made clear later.

# Strings

## Concatenation of strings

- The *concatenation* of strings  $x, y$  is the string  $xy$  formed by appending  $y$  to the end of  $x$ .
  - The concatenation of  $x = 010$  and  $y = 01$  is  $01001$ .
- $w\epsilon = \epsilon w = w$  for all strings  $w$ .
  - $01\epsilon = 01$

# Decision problems

## Definition

A **decision problem** is a function  $P : \Sigma^* \rightarrow \{0, 1\}$ .

## Example

$$P(s) = \begin{cases} 1 & \text{if the length of the string } s \in \{0, 1\}^* \text{ is odd,} \\ 0 & \text{otherwise.} \end{cases}$$

We will also write the decision problem like this:

**Input:** String  $s$  over alphabet  $\{0, 1\}$ .

**Output:** 1 if the length of the string  $s$  is odd, and 0 otherwise.

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

## Example

The program

---

```
1 def L(s): # s binary-string
2     # a % b returns the remainder of a divided by b
3     return len(s)%2
```

---

solves the decision problem

**Input:** A string  $s$ .

**Output:** 1 if the length of the string  $s$  is odd, and 0 otherwise.

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

## Example

The program

---

```
1 def Q(x): # binary string x encoding natural number
2         # least significant digit first
3     return x[0]
```

---

solves the decision problem

**Input:** A string encoding a natural number.

**Output:** 1 if the integer is odd, and 0 otherwise.

# What decision problems can be solved by programs?

**Input:** Strings encoding integers  $x, y, z$ .

**Output:** 1 if  $z = x + y$ , and 0 otherwise.

**Input:** Strings  $w, t$ .

**Output:** 1 if  $w$  is a substring of  $t$ , and 0 otherwise.

**Input:** Strings encoding integers  $x, y, z$ .

**Output:** 1 if  $z = x \times y$ , and 0 otherwise.

**Input:** String encoding an arithmetic expression  $e$ .

**Output:** 1 if  $e$  is well-bracketed, and 0 otherwise.

**Input:** String encoding a (Python) program  $p$ .

**Output:** 1 if  $p$  enters an infinite loop, and 0 otherwise.

# What decision problems can be solved by programs?

We will see that, in a very precise sense:

1. “addition” and “substring” can be decided with limited memory,
2. “multiplication” and “well-bracketed” require recursion,
3. “infinite loop” cannot be decided by any program.

To do this, we introduce **mathematical models** of programs that solve decision problems.

1. Limited memory: regular expressions and finite automata (L5-L7).
2. Plus recursion: context-free grammars and pushdown automata (L8-L9).
3. Universal: Turing machines and Lambda-calculus (L10-L11).

# One more modeling choice

We will think of decision problems as deciding if the input is in some **set of strings**.

## Example

Here are two ways of describing the same problem.

**Input:** A binary string encoding a natural number.  
**Output:** 1 if the integer is odd, and 0 otherwise.

**Input:** A binary string encoding a natural number.  
**Output:** 1 if the string is in the set  $\{w \in \{0, 1\}^* : w[0] = 1\}$ , and 0 otherwise.

Sets of strings are called **languages**. They are so important, they have their own field of study called **Formal Language Theory**.

# Languages

## Definition

A set  $L \subseteq \Sigma^*$  of strings is called a (formal) language over  $\Sigma$ .

## Example

Let  $\Sigma = \{a, b\}$ .

- $L_1 = \{x \in \Sigma^* : x \text{ starts with a } b\}$
- $L_2 = \{x \in \Sigma^* : x \text{ has an even number of } bs\}.$
- $L_3 = \{x \in \Sigma^* : x \text{ has the same number of } as \text{ as } bs\}.$
- The empty set  $\emptyset$ .
- **Note.** The empty set  $\emptyset$  has no elements in it, but the set  $\{\epsilon\}$  has one element in it, the empty string.

## For you to think about after class

1. Is HTML a formal language? If so, what is the alphabet and what are the strings?
2. Is music a formal language? If so, what is the alphabet and what are the strings?

# Important operations on languages

## Definition

Let  $A, B$  be languages over  $\Sigma$ .

- The **union** of  $A$  and  $B$ , written  $A \cup B$ , is the language  $\{x \in \Sigma^* : x \in A \text{ or } x \in B\}$ .
- The **concatenation** of  $A$  and  $B$ , written  $A \circ B$ , is the language  $\{xy \in \Sigma^* : x \in A, y \in B\}$ .
- The **star** of  $A$ , written  $A^*$ , is the language  $\{x_1x_2 \cdots x_k \in \Sigma^* : k \geq 0, k \in \mathbb{Z}, \text{ each } x_i \in A\}$ .
  - So  $A^* = \{\epsilon\} \cup \overbrace{A}^1 \cup \overbrace{A \circ A}^2 \cup \overbrace{A \circ A \circ A}^3 \cup \dots$

# Important operations on languages

## Examples

$$1. \{a, ab\} \cup \{ab, aab\} =$$

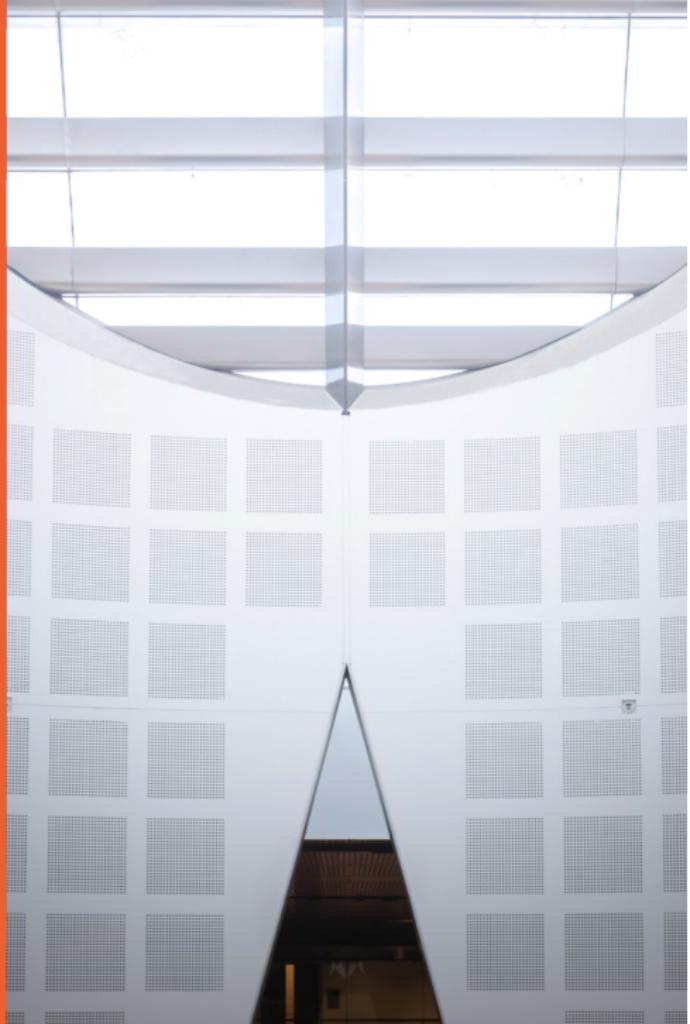
$$2. \{a, ab\} \circ \{ab, aab\} =$$

$$3. \{a, ab\}^* =$$

**COMP2022|2922**  
**Models of Computation**  
**Lesson 5b: Regular Languages**  
**Chapter 1 of Sipser (eReserve)**

**Presented by**

Sasha Rubin  
School of Computer Science



# Regular expressions in a nutshell

- Expressions that describe languages
- Extremely useful
  - Pattern matching (Control-F)
  - Specification of data formats
  - Lexical analysis
- Based on three operations:
  - Language Union  $L_1 \cup L_2$
  - Language Concatenation  $L_1 \circ L_2$
  - Language Iteration/Kleene-star  $L^*$

# Regular expressions: Syntax

## Definition

Let  $\Sigma$  be an alphabet. The regular expressions over  $\Sigma$  are defined by the following recursive process:

- R1. The symbols  $\emptyset$  and  $\epsilon$  are regular expressions.
- R2. Each symbol  $a \in \Sigma$  is a regular expression.
- R3. If  $R_1, R_2$  are regular expressions then so is  $(R_1 \cup R_2)$
- R4. If  $R_1, R_2$  are regular expressions then so is  $(R_1 \circ R_2)$ , also written  $(R_1 R_2)$
- R5. If  $R$  is a regular expressions then so is  $R^*$

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ .

- $(a \cup \emptyset)$
- $(a \circ \epsilon)$
- $b^*$
- $(a^* \cup b^*)$
- $(a \cup b)^*$
- $(a \circ b)^*$

# Regular expressions: Semantics

A regular expression  $R$  *matches* certain strings, collectively called  $L(R)$ .

## Definition

Let  $\Sigma$  be an alphabet. The language  $L(R)$  **represented** by a regular expression  $R$  is defined by the following recursive procedure:

1.  $L(\emptyset) = \emptyset$  and  $L(\epsilon) = \{\epsilon\}$ .
2.  $L(a) = \{a\}$  for  $a \in \Sigma$ .
3.  $L((R_1 \cup R_2)) = L(R_1) \cup L(R_2)$ .
4.  $L((R_1 \circ R_2)) = L(R_1) \circ L(R_2)$ .
5.  $L(R^*) = L(R)^* =$   
 $\{\epsilon\} \cup L(R) \cup L(R) \circ L(R) \cup L(R) \circ L(R) \circ L(R) \cup \dots$

# Regular expressions

## Notation.

- The symbol  $\cup$  in the regular expression  $(a^* \cup \epsilon)$  is just a **symbol** (in some texts it is denoted  $+$  or  $|$ ), and is different from the union **operation** on languages.
- So, strictly speaking, we should have defined the syntax of regular expressions using different notation, e.g.,  $(a^* \cup \epsilon)$ .
- But this is inconvenient to write, and so we overload and use the same symbol in the syntax and the semantics.

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ .

- $L((a \cup \emptyset)) =$
- $L((a \circ \epsilon)) =$
- $L(b^*) =$
- $L((a^* \cup b^*)) =$
- $L((a \cup b)^*) =$
- $L((a \circ b)^*) =$

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ . Write regular expressions for the following languages:

1. The set of strings ending in  $a$ .
  - $((a \cup b)^* \circ a)$ , also written  $((a \cup b)^* a)$ .
2. The set of strings whose second last symbol is an  $a$ .
  - $((a \cup b)^* \circ a \circ (a \cup b))$
3. The set of strings that have  $aba$  as a substring.
  - $((a \cup b)^* \circ a \circ b \circ a \circ (a \cup b)^*)$
4. The set of strings of even length.
5. The set of strings with an even number of  $a$ 's.

# Regular expressions

## Important questions

1. Which languages can be described by regular expressions? All languages?
2. There are natural decision problems associated with regular expressions. Are there programs that solve them?

**Input:** Regular expression  $R$  and string  $s$

**Output:** 1 if  $s \in L(R)$ , and 0 otherwise.

**Input:** Regular expressions  $R_1, R_2$

**Output:** 1 if  $L(R_1) = L(R_2)$ , and 0 otherwise.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 5c: Finite Automata**

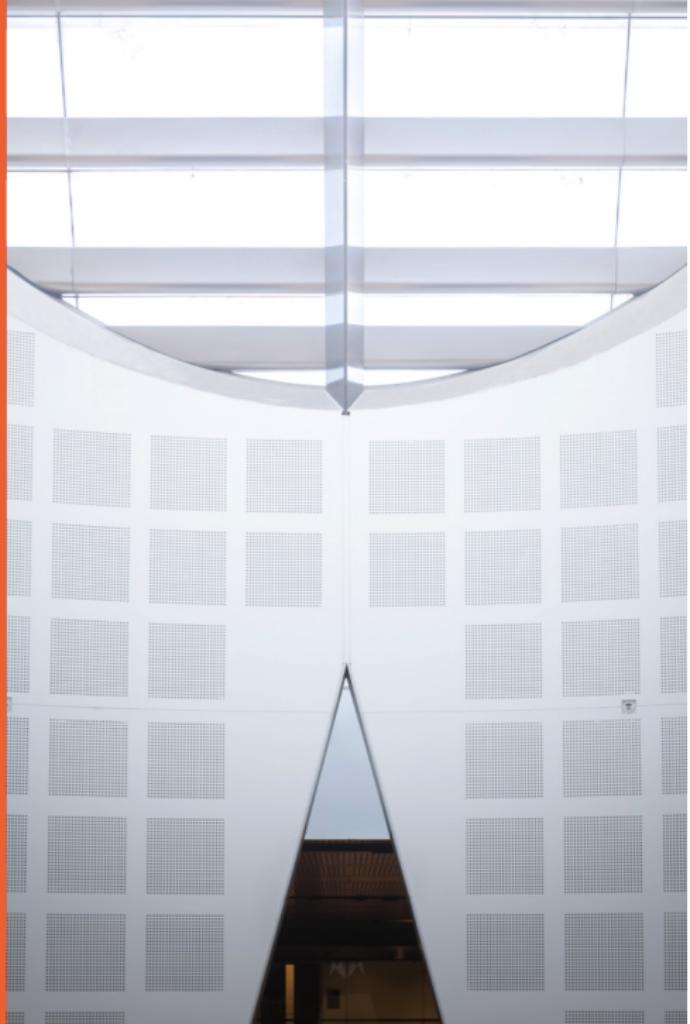
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# Finite automata in a nutshell

A deterministic finite automaton is like a program that can only scan its input string once, and cannot write anything. At the end of reading the input string, it must decide if the string is in the language (accept it) or not (reject it).

# Why study such a simple model of computation?

1. It is part of computing culture.
  - first appeared in McCulloch and Pitt's model of a neural network (1943)
  - then formalised by Kleene (American mathematician) as a model of stimulus and response
2. It has numerous practical applications.
  - lexical analysis (tokeniser)
  - pattern matching
  - communication protocols with bounded memory
  - circuits with feedback
  - finite-state reactive systems
  - finite-state controllers
  - non-player characters in computer games
  - ...
3. It is simple to implement/code.

# Drawing deterministic finite automata (DFA)

# Definition of a deterministic finite automaton (DFA)

## Definition

A deterministic finite automaton (DFA)  $M$  is a tuple  $(Q, \Sigma, \delta, q_0, F)$  where

1.  $Q$  is a finite set of states,
2.  $\Sigma$  is a finite set called the alphabet,
3.  $\delta : Q \times \Sigma \rightarrow Q$  is the transition function,
4.  $q_0 \in Q$  is the start state, and
5.  $F \subseteq Q$  is the set of accept states, sometimes called final states.

If  $q' = \delta(q, a)$  we write  $q \xrightarrow{a} q'$ , called a transition. The source of the transition is  $q$ , the target is  $q'$ , and the label is  $a$ .

# Language described by a DFA

A DFA  $M$  describes a language  $L(M)$ : all strings that label paths from the start state to an accepting state.

## Examples

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1.  $\{aba\}$
2. all strings over  $\Sigma$
3. the set of strings ending in  $a$
4.  $L((a \circ b)^*)$
5. the set of strings with an even number of  $a$ 's
6. the set of strings with an odd number of  $b$ 's

# Designing DFAs

Draw a DFA for the language  $\{aba\}$

# Designing DFAs

Draw a DFA for the set of all strings over  $\{a, b\}$

# Designing DFAs

Draw a DFA for the set of all strings ending in  $a$

# Designing DFAs

Draw a DFA for the set of all strings matching  $(ab)^*$

# Designing DFAs

Draw a DFA for the set of all strings with an even number of  $a$ 's

# Designing DFAs

Draw a DFA for the set of all strings with an odd number of  $b$ 's

# Definition of the language recognised by a DFA

## Definition

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a DFA, and let  $w = w_1w_2 \cdots w_n$  be a string over  $\Sigma$ .

- A **run** of  $M$  on  $w$  is a sequence  $r = r_0r_1 \cdots r_n$  of states of  $M$  such that
  1.  $r_0 = q_0$ , and
  2.  $\delta(r_i, w_{i+1}) = r_{i+1}$  for  $i = 0, \dots, n - 1$ .
- The run  $r$  is **accepting** if also
  3.  $r_n \in F$ .
- If  $w$  has an accepting run then we say that  $M$  **accepts**  $w$ .

The set  $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$  is the **language recognised by  $M$** .

# DFAs

## Important questions

1. Which languages can be described by DFAs? All languages?
2. There are natural decision problems associated with DFAs. Are there programs that solve them?

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

## The problem

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

is solved by the program

---

```
1 def run(M,w):
2     cur = q_0
3     for i = 1 to len(w):
4         cur = δ(cur,w_i)
5     if cur in F:
6         return 1
7     else:
8         return 0
```

---

## Definition

A language  $L$  is called **regular** if  $L = L(M)$  for some DFA  $M$ .

## Theorem

*The regular languages are closed under complementation. That is, if  $A$  is regular, then  $\Sigma^* \setminus A$  is regular.*

## Example

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1. strings that contain  $aba$  as a substring
2. strings that do not contain  $aba$  as a substring

## Theorem

*The regular languages are closed under complementation. That is, if  $A$  is regular, then  $\Sigma^* \setminus A$  is regular.*

### Construction ("swap accept and reject states")

1. Given  $A = L(M)$  for some DFA  $M = (Q, \Sigma, \delta, q_0, F)$ .
2. We will construct a DFA  $M'$  recognising  $\Sigma^* \setminus L(M)$ .
3. Define  $M' = (Q, \Sigma, \delta, q_0, F')$  where  $F' = Q \setminus F$ .

## Theorem

*The regular languages are closed under union. That is, if A and B are regular, then  $A \cup B$  is regular.*

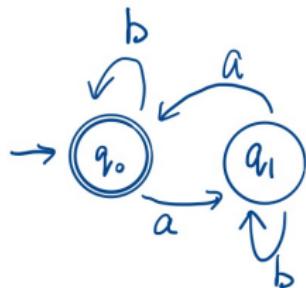
## Example

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1. the strings consisting of an even number of a's
2. the strings consisting of an odd number of b's
3. the strings consisting of an even number of a's or an odd number of b's.

## Designing DFAs

Draw a DFA for the set of all strings with an even number of a's

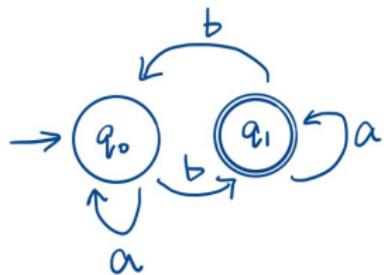


$q_0$  represents strings that contain even number of a's  
and arbitrary number of b's.

$q_1$  represents strings that contain odd number of a's  
and arbitrary number of b's.

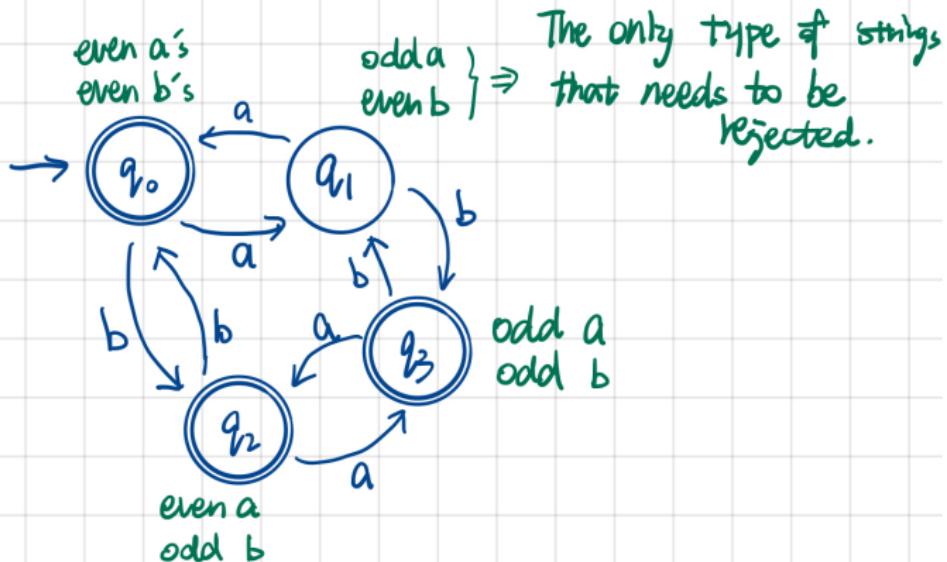
## Designing DFAs

Draw a DFA for the set of all strings with an odd number of b's



$q_0$  represents strings that contain even number of b's  
and arbitrary number of a's.

$q_1$  represents strings that contains odd number of b's  
and arbitrary number of a's.



## Theorem

*The regular languages are closed under union. That is, if  $A_1$  and  $A_2$  are regular, then  $A_1 \cup A_2$  is regular.*

## Construction ("product")

1. Given DFA  $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  recognising  $A_i$ ,  $i = 1, 2$ .
2. We will construct a DFA  $M$  recognising  $L(A_1) \cup L(A_2)$ .
3. Define  $M = (Q, \Sigma, \delta, q_0, F)$  where
  - $Q = Q_1 \times Q_2$ ,
  - $\delta$  maps state  $(s_1, s_2)$  on input  $a \in \Sigma$  to state  $(\delta_1(s_1, a), \delta_2(s_2, a))$ ,
  - $q_0 = (q_1, q_2)$ ,
  - $F = \{(s_1, s_2) : s_1 \in F_1 \text{ or } s_2 \in F_2\}$ .

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 6a: Nondeterministic finite automata**

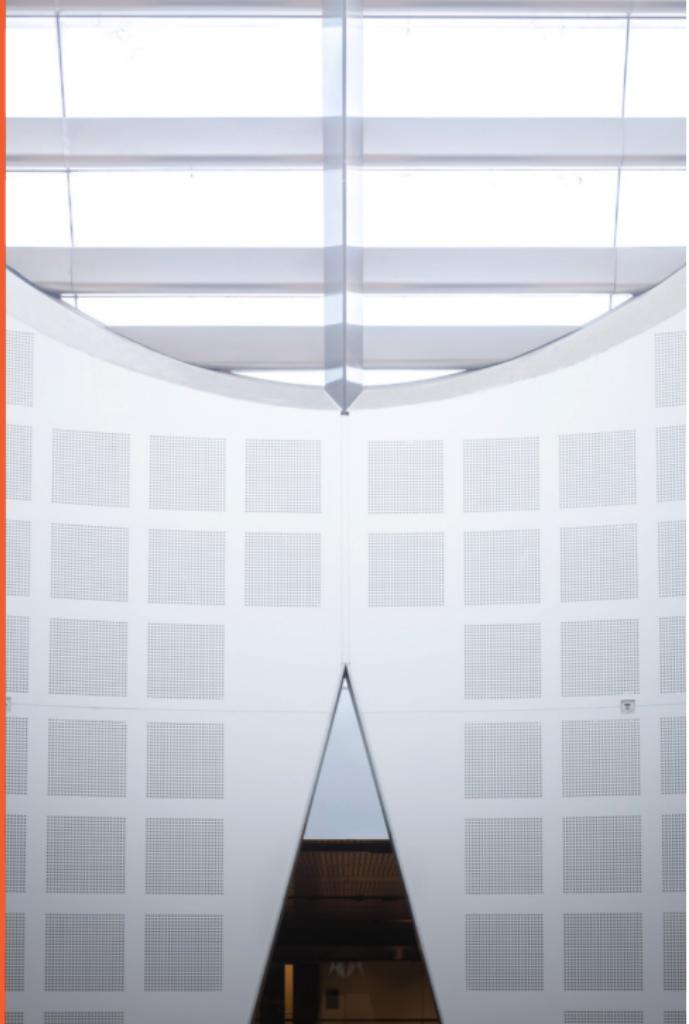
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



## Theorem

*The regular languages are closed under concatenation. That is, if A and B are regular, then  $A \circ B$  is regular.*

Idea:

- Let  $M_A$  be a DFA recognising A, let  $M_B$  be a DFA recognising B.
- We want to build a DFA  $M$  for  $A \circ B$ .
- One way would be for  $M$  to simulate  $M_A$ , and at some point, as long as the state of  $M_A$  is an accepting state, “guess” that it is time to switch, and simulate  $M_B$  on the remaining input.

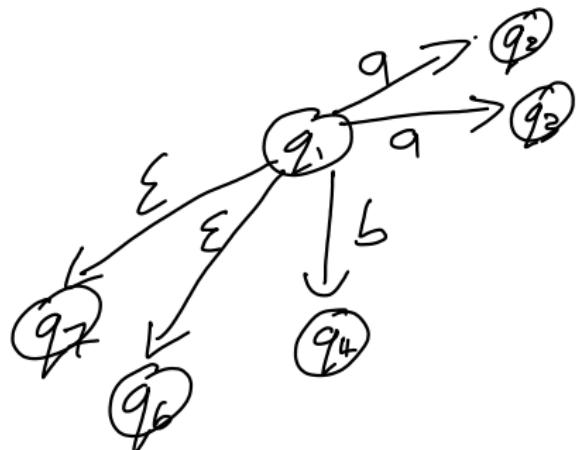
**Question.** How to we capture this “guess”?

We will introduce a model of computation, called a **nondeterministic finite automaton (NFA)**, which is like a DFA except that states can have more than one outgoing transition on the same input symbol. Later we will show how DFAs can simulate NFAs.

# Definition of NFA

## Definition

A **nondeterministic finite automaton**  $M$  is a tuple  $(Q, \Sigma, \delta, q_0, F)$  the same as a DFA except the transition function is  $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ . If  $q' \in \delta(q, a)$  we write  $q \xrightarrow{a} q'$ .<sup>1</sup>



$$\delta(q_1, a) = \{q_2, q_3\}$$

$$\delta(q_1, b) = \{q_4\}$$

$$\delta(q_1, \epsilon) = \{q_5, q_6\}$$

---

<sup>1</sup>Recall:  $\mathcal{P}(Q)$  is the set of subsets of  $Q$ . So  $\mathcal{P}(\{q_1, q_2, q_3\})$  has 8 elements.

# Definition of NFA

The NFA  $M$  **accepts**  $w$  if there is a path

$$q_0 \xrightarrow{y_0} q_1 \xrightarrow{y_1} q_2 \dots \xrightarrow{y_m} q_m$$

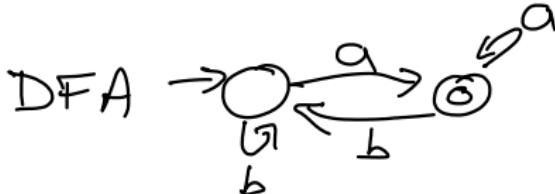
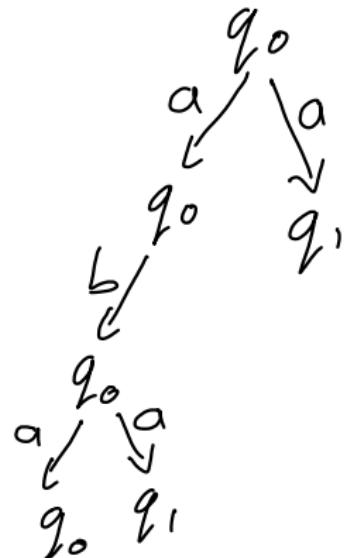
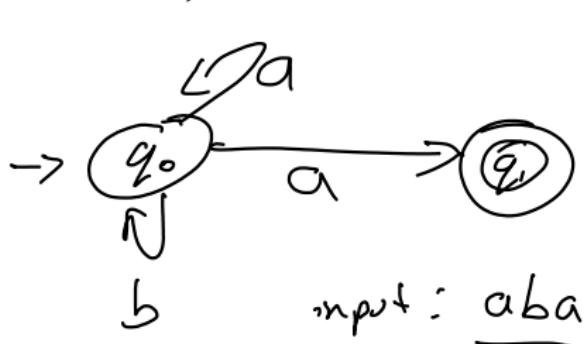
in  $M$  such that  $w = y_1 y_2 \dots y_m$  and  $q_m \in F$ .

That is: an NFA  $M$  describes the language  $L(M)$  of all strings that label paths from the start state to an accepting state.

# Language described by NFAs

## Examples

Let  $\Sigma = \{a, b\}$ . Draw an NFA for the language consisting of the set of strings whose last letter is an  $a$  (also, draw a DFA, for comparison).



# Comparing NFAs and DFAs

1. In a DFA, every input has exactly one run. The input string is accepted if this run is accepting.
2. In an NFA, an input may have zero, one, or more runs. The input string is accepted if at least one of its runs is accepting.
3. For every DFA  $M$  there is an NFA  $N$  such that  $L(M) = L(N)$ .
  - Given DFA  $M = (Q, \Sigma, \delta, q_0, F)$  with  $\delta : Q \times \Sigma \rightarrow Q$ , define the NFA  $N = (Q, \Sigma, \delta', q_0, F)$  where  $\delta'(q, a) = \{\delta(q, a)\}$ .
  - To see that  $L(M) = L(N)$  observe that the diagrams of  $M$  and  $N$  are the same.



DFA

$$\delta(q, a) = q'$$

NFA

$$\delta(q, a) = \{q'\}$$

# From Regular Expressions to DFA

## Theorem

For every regular expression  $R$  there is an NFA  $M_R$  such that  $L(R) = L(M_R)$ .

The construction is by recursion on the structure of  $R$ . The base cases are  $R = \emptyset, R = \epsilon, R = a$  for  $a \in \Sigma$ . The recursive cases are  $R = R_1 \circ R_2, R = R_1 \cup R_2$ , and  $R = (R_1)^*$ .

Base cases:



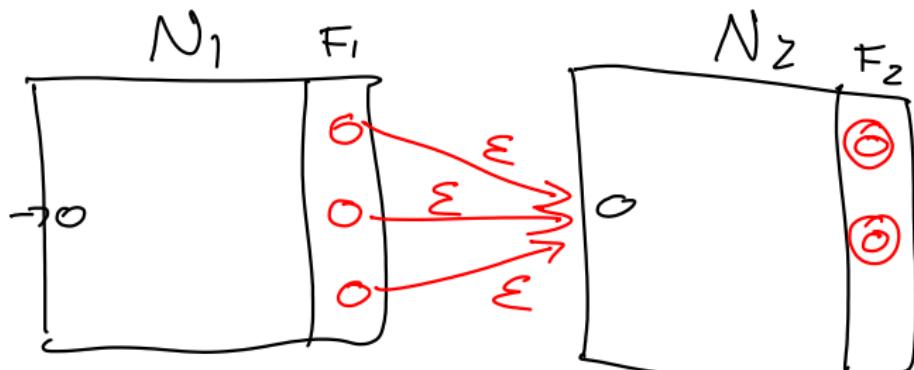
# NFAs are closed under $\circ$

## Lemma

If  $N_1, N_2$  are NFAs, there is an NFA  $N$  recognising  $L(N_1) \circ L(N_2)$ .

## Construction ("Simulate $N_1$ followed by $N_2$ ")

Given NFAs  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct NFA  $N$  with states  $Q_1 \cup Q_2$  as in the figure. The idea is that  $N$  guesses how to break the input into two pieces, the first accepted by  $N_1$ , and the second by  $N_2$ .



# NFAs are closed under $\circ$ (reference slide)

Here is the construction. From  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct  $N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \setminus F_1 \\ \delta_1(q, a) & q \in F_1, a \neq \epsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1, a = \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

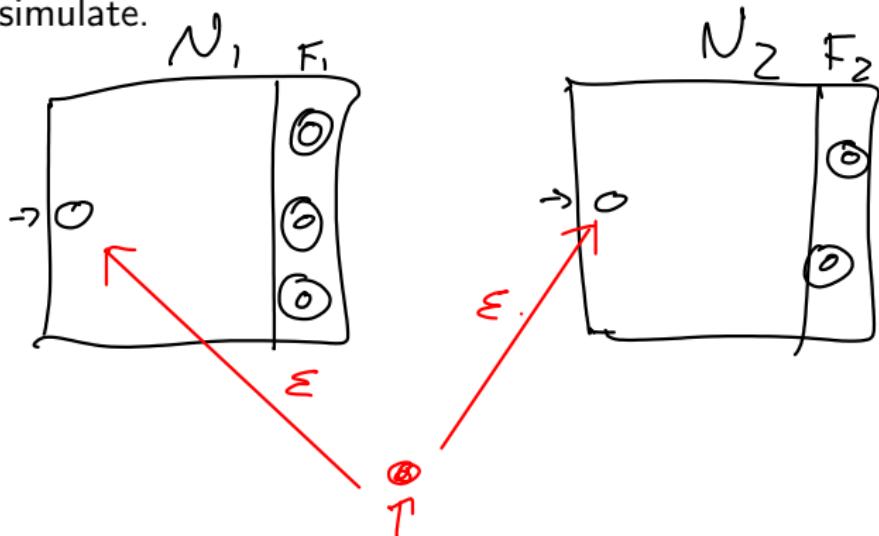
# NFAs are closed under $\cup$

## Lemma

If  $N_1, N_2$  are NFAs, there is an NFA  $N$  recognising  $L(N_1) \cup L(N_2)$ .

## Construction ("Simulate $N_1$ or $N_2$ ")

Given NFAs  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct NFA  $N$  with states  $Q_1 \cup Q_2 \cup \{q_0\}$  as in the figure. The idea is that  $N$  guesses which of  $N_1$  or  $N_2$  to simulate.



# NFAs are closed under $\cup$ (reference slide)

Here is the construction. From  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct  $N = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup F_2)$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon \end{cases}$$

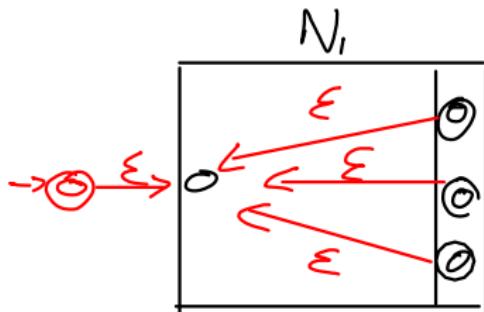
# NFAs are closed under \*

## Lemma

If  $N_1$  is an NFA, there is an NFA  $N$  recognising  $L(N_1)^*$ .

**Construction ("Simulate  $N$  and go back to the initial state")**

Given NFAs  $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  construct NFA  $N$  with states  $Q \cup \{q_0\}$  as in the figure. The idea is that  $N$  guesses how to break the input into pieces, each of which is accepted by  $N_1$ .



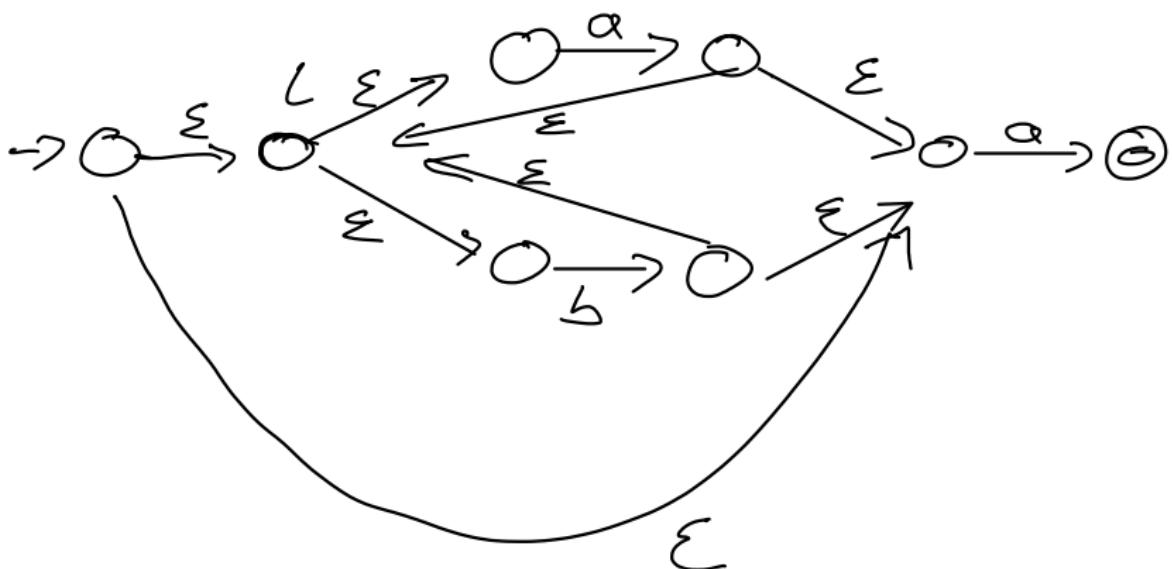
# NFAs are closed under \* (reference slide)

Here is the construction. From  $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  construct  $N = (Q_1 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup \{q_0\})$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & (q \in Q_1 \setminus F_1) \text{ or } (q \in F_1, a \neq \epsilon) \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1, a = \epsilon \\ \{q_1\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon \end{cases}$$

## From RE to NFA: example

Convert the regular expression  $((a \cup b)^* \circ a)$  to an NFA (using the construction just given).



# **COMP2022|2922**

## **Models of Computation**

### **Lesson 6b: NFAs, DFAs, REs have the same expressive power**

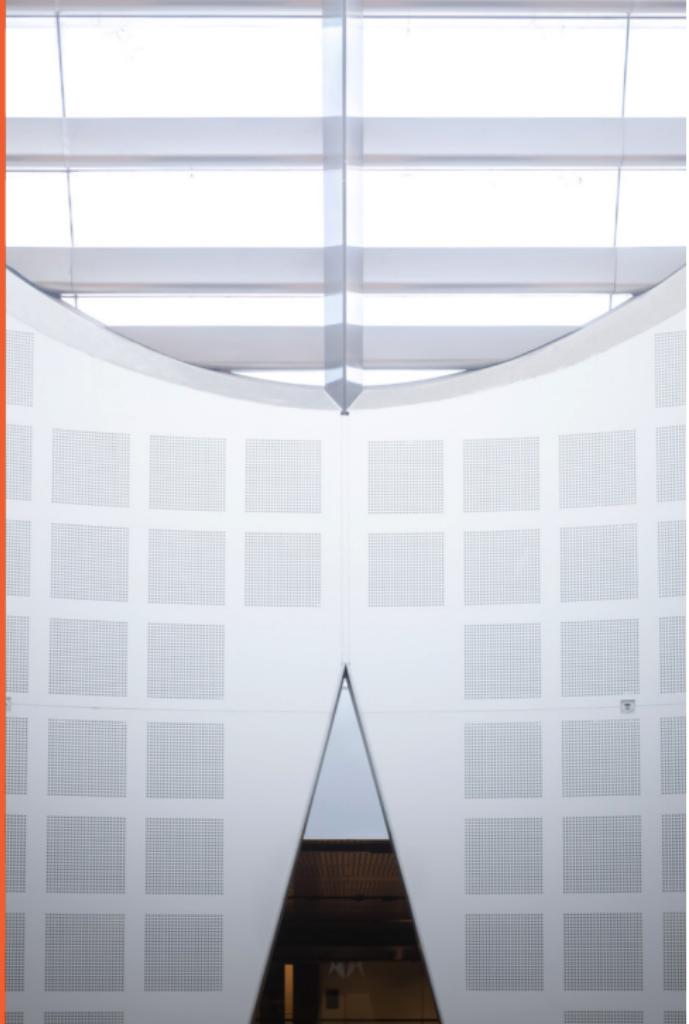
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# Where are we going?

We are going to show that DFA, NFA and regular expressions have the same expressive power. I.e., they recognise the same languages.

1. From Regular Expressions to NFAs
2. From NFAs to NFAs without  $\epsilon$ -transitions
3. From NFAs without  $\epsilon$ -transitions to DFAs
4. From DFAs to Regular Expressions

# From NFAs to NFAs without $\epsilon$ -transitions

## Theorem

For every NFA  $N$  there is an NFA  $N'$  without  $\epsilon$ -transitions such that  $L(N) = L(N')$ .

## Construction ("skip $\epsilon$ -transitions")

Given NFA  $N = (Q, \Sigma, \delta, q_0, F)$  write  $s \xrightarrow{\epsilon^*} t$  if there is a path from  $s$  to  $t$  labelled only by  $\epsilon$ . Construct  $N' = (Q, \Sigma, \delta', q_0, F')$  where

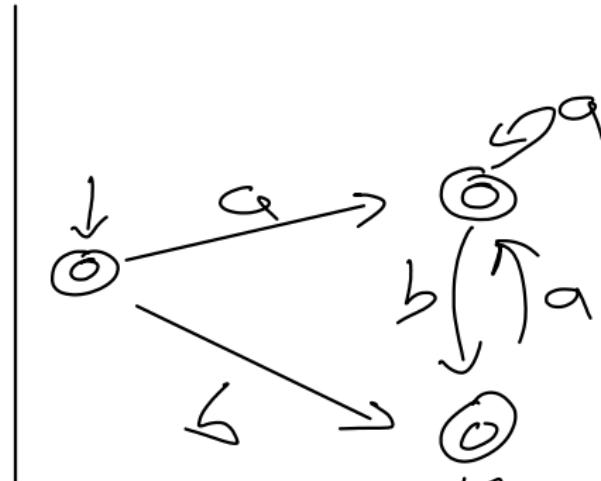
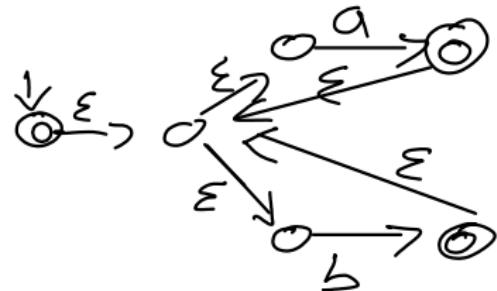
- $t \in \delta'(s, a)$  if, in  $N$ , there is a state  $q$  such that  $s \xrightarrow{\epsilon^*} q \xrightarrow{a} t$ .
- $s \in F'$  if, in  $N$ , there is a state  $q \in F$  such that  $s \xrightarrow{\epsilon^*} q$ .

The idea is that  $N'$  skips over the  $\epsilon$ -transitions of  $N$ .

# From NFAs to NFAs without $\epsilon$ -transitions

Example

$$(a \cup b)^*$$



We don't draw states that are not reachable from the initial state.

# From NFAs without $\epsilon$ -transitions to DFAs

## Theorem

For every NFA  $N$  without  $\epsilon$ -transitions there is a DFA  $M$  such that  $L(M) = L(N)$ .

## Construction ("subset construction")

Intuitively,  $M$  keeps track of all possible states that  $N$  can be in.

Given NFA  $N = (Q, \Sigma, \delta, q_0, F)$  construct  $M = (Q', \Sigma, \delta', \{q_0\}, F')$  where

- $Q' = \mathcal{P}(Q)$ , i.e., every subset  $X \subseteq Q$  is a state of  $Q'$ ,
- for  $X \subseteq Q$ ,  $\delta'(X, a) = \bigcup_{q \in X} \delta(q, a)$ ,
- $F' = \{X \subseteq Q : X \cap F \neq \emptyset\}$ .

# From NFAs without $\epsilon$ -transitions to DFAs

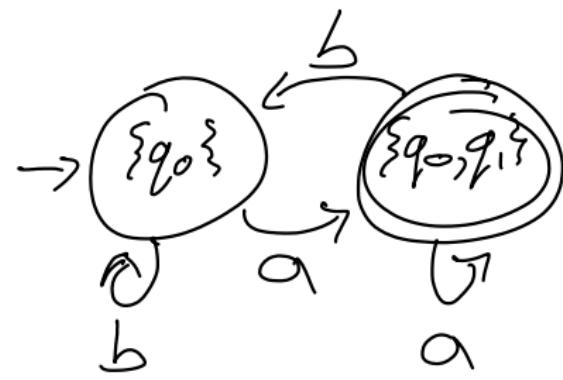
Example



DFA

$$\mathcal{Q}' = \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$$

$\delta'$	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0\}$



# From DFAs to Regular Expressions

## Theorem

For every DFA  $M$  there is a regular expression  $R$  such that  $L(M) = L(R)$ .

Idea:

1. We convert  $M$  into a **generalised nondeterministic finite automaton (GNFA)** which is like an NFA except that it can have regular expressions label the transitions.
2. We **successively remove states** from the automaton, until there is just one transition (from the start state to the final state).
3. We return the regular expression on this last transition.

# GNFAs

A GNFA  $N$  accepts  $w$  if there is a path

$$q_0 \xrightarrow{R_0} q_1 \xrightarrow{R_1} q_2 \dots \xrightarrow{R_m} q_m$$

in  $M$  such that  $w$  matches the regular expression  $R_1 R_2 \dots R_m$  and  $q_m \in F$ .

**Example**



caaba is accepted.

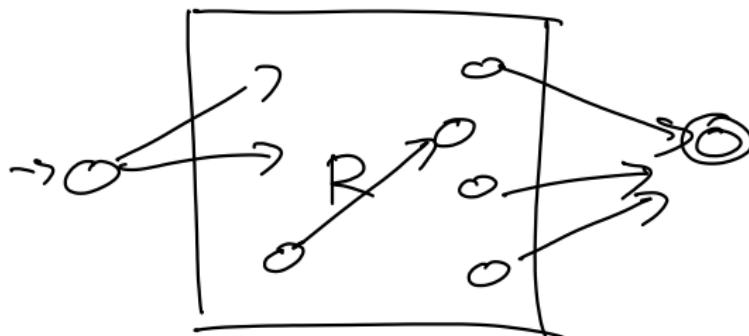
cab is not.

# From DFAs to Regular Expressions

Generalised NFAs. We make sure that:

1. the initial state has no incoming edges.
2. there is one final state, and it has no outgoing edges.
3. there are edges from every state (that is not final) to every state (that is not initial).

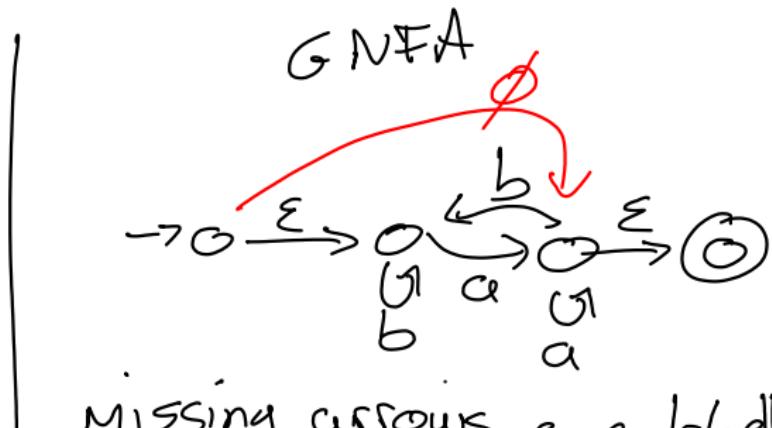
We show how to translate DFAs into GNFs, and then successively remove states from the GNFA until there is just a single edge left.



# From DFAs to GNFAs

For every DFA  $M$  there is a GNFA  $N$  such that  $L(M) = L(N)$ .

1. Add an initial state and  $\epsilon$ -transition to the old initial state.
2. Add a final state and  $\epsilon$ -transitions to it from all the old final states.
3. Add transitions for every pair of states (except from the new final to the new initial).



MISSING arrows are labelled  
by  $\emptyset$

# Removing states from GNFA

For every GNFA  $N$  with  $> 2$  states there is a GNFA  $N'$  with one less state such that  $L(N) = L(N')$ .

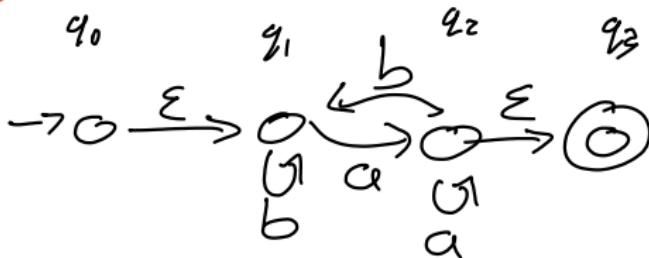
1. Pick a state  $q$  to eliminate.
2. For every pair of remaining states  $(s, t)$ , replace the label from  $s$  to  $t$  by the regular expression

$$(R_{s,t} \cup (R_{s,q} \circ (R_{q,q})^* \circ R_{q,t})).$$

where  $R_{x,y}$  is the regular expression labeling the transition from state  $x$  to state  $y$ .

# Removing states from GNFA

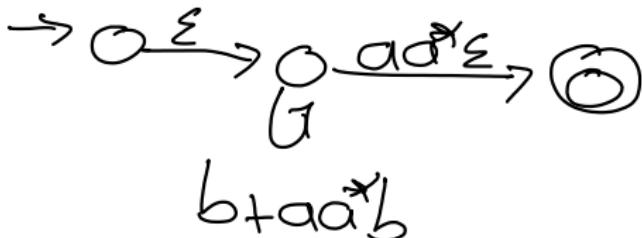
Example



Simplify

$$\begin{aligned} & ((\epsilon + a^*b)a^*)^* \\ & (a^*b)^*a^*\end{aligned}$$

Remove  $q_2$ :



Remove  $q_1$ :



Reg Exp:  $\epsilon(b+aa^*b)^*aa^*\epsilon$ .

# Summary

1. The following models of computation describe the same languages: DFAs, NFAs, Regular Expressions.
2. The regular languages are closed under the Boolean operations union, intersection, complementation, as well as concatenation and Kleene star.

# Decision Problems about DFAs

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

# Emptiness problem for DFAs

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

## Algorithm ("mark reachable states")

1. Mark the states using the following recursive procedure.
  - 1.1 Mark the initial state.
  - 1.2 If  $q$  is marked and  $p \xrightarrow{a} q$  (for some  $a \in \Sigma$ ), then mark  $p$ .
  - 1.3 (Stop when no new states are marked).
2. Return 0 if a final state is marked, else return 1.

# Equivalence problem for DFAs

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

## Algorithm

1. Form DFA  $M$  recognising
$$(L(M_1) \setminus L(M_2)) \cup (L(M_2) \setminus L(M_1)).$$
  - Note that  $L(M_1) = L(M_2)$  iff  $L(M) = \emptyset$ .
2. So we just need a way to check if  $L(M) = \emptyset$ . Use the algorithm for the Emptiness problem for DFAs.

# Decision Problems about Regular Expressions

**Input:** Regular expression  $R$  and string  $w$

**Output:** 1 if  $w \in L(R)$ , and 0 otherwise.

**Input:** Regular expressions  $R_1, R_2$

**Output:** 1 if  $L(R_1) = L(R_2)$ , and 0 otherwise.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 6c: Non-regularity**

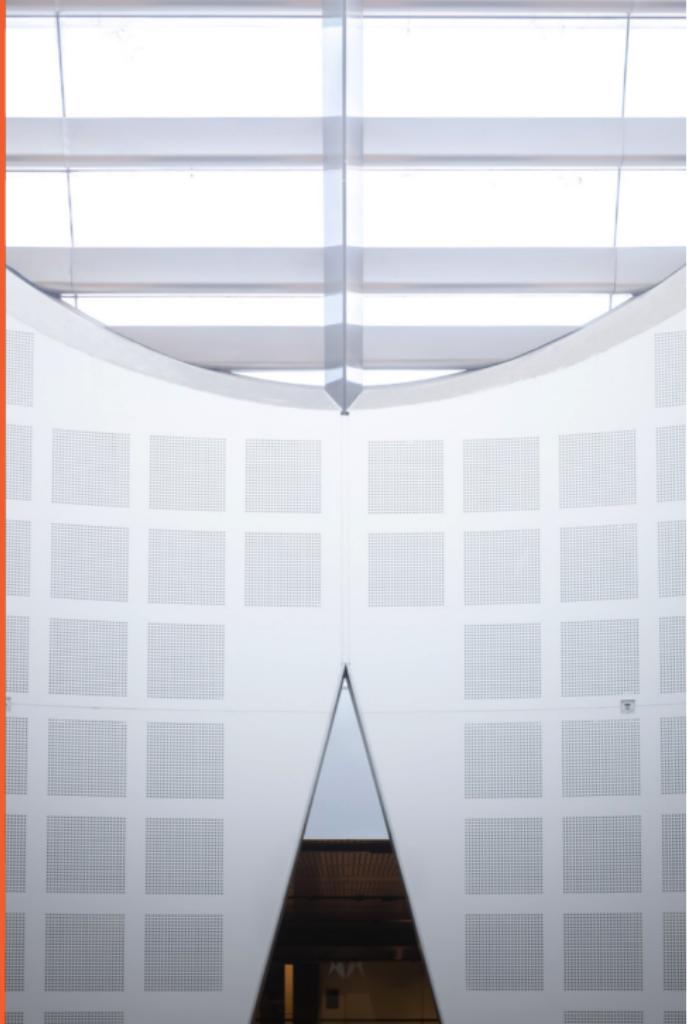
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



To show that a language is regular, it is sufficient to find a DFA, NFA, or Regular Expression for it. To show that a language  $L$  is not regular, one must show that there is **no DFA** that recognises it. This can be done using a **fooling argument**.

1. Assume there is a DFA  $M$  recognising the language  $L$ .
2. Reason about the transitions of this DFA and find a string not in the language which it accepts (or a string in the language which it rejects).
3. This contradicts the assumption that  $M$  recognises  $L$ .
4. Conclude that there is no such DFA.

# Pigeonhole principle (PHP)

If there are  $n$  holes and  $> n$  objects to put in the holes, then at least one hole must get at least two objects.

If there are finitely many holes and infinitely many objects to put in the holes, then at least one hole must get infinitely many objects.

$L = \{ww : w \in \{a,b\}^*\}$  is not regular

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $u \neq v$  such that  $f(u) = f(v)$ .
4. But  $uu \in L$  means that the path from state  $f(u)$  labeled  $u$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $vu$ . So  $vu$  is accepted by  $M$ .
6. But since  $u \neq v$ ,  $M$  accepts a word not in  $L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

$L = \{a^i b^i : i \geq 0\}$  is not regular

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $n \neq m$  such that  $f(a^n) = f(a^m)$ .
4. But  $a^n b^n \in L$  means that the path from state  $f(a^n)$  labeled  $b^n$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $a^m b^n$ . So  $a^m b^n$  is accepted by  $M$ .
6. But since  $m \neq n$ ,  $M$  accepts a word not in  $L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

$$L = \{u \in \{a, b\}^*: |u| \text{ is a power of } 2\}$$

Use the fact that  $2^{(n+1)} = 2^n + 2^n$ .

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $n < m$  such that  $f(u) = f(v)$  and  $|u| = 2^n, |v| = 2^m$ .
4. But  $uu \in L$  means that the path from state  $f(u)$  labeled  $u$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $vu$ . So  $vu$  is accepted by  $M$ .
6. But  $vu$  is not in  $L$  since  $2^m < |vu| = 2^m + 2^n < 2^m + 2^m = 2^{m+1}$ . So  $M$  accepts a word not in  $L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

## Other techniques

Once we know that a language is not regular, we can deduce that other languages are not regular using the closure properties of regular languages.

# Showing a language is not regular

## Example

We have seen that  $L = \{a^i b^i : i \geq 0\}$  is not regular. Use this fact to prove that the set  $L'$  of strings with the same number of *as* as *bs* is not regular.

1. Assume  $L'$  is regular.
2. Then  $L' \cap L(a^*b^*)$  is regular since the intersection of regular languages is regular.
3. But  $L = L' \cap L(a^*b^*)$ , which we know is not regular.
4. This contradicts the assumption that  $L'$  is regular.
5. So  $L'$  is not regular.

# Showing a language is not regular

## Example

We have seen that the language  $L'$  consisting of strings over  $\{a, b\}$  with the same number of  $a$ s as  $b$ s is not regular. Use this fact to prove that the language  $L''$  consisting of strings over  $\{a, b\}$  with a different number of  $a$ s as  $b$ s is not regular.

1. Assume  $L''$  is regular.
2. Then  $L((a \cup b)^*) \setminus L''$  is regular since the complement of regular languages is regular.
3. But  $L' = L((a \cup b)^*) \setminus L''$ , which we know is not regular.
4. This contradicts the assumption that  $L''$  is regular.
5. So  $L''$  is not regular.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 5a: Introduction to Machine Models**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# What sort of problems are programs meant to solve?

- A **computational problem** specifies what the input can be and what the output should be.
  1. Given numbers  $x, y$ , output  $x + y$ ?
  2. Given numbers  $x, y$ , output  $x \times y$ ?
  3. Given strings  $w, t$ , decide if  $w$  is a substring of  $t$ ?
  4. Given a string  $e$ , decide if  $e$  is well-bracketed?
- The last two are called **decision problems** because their output is either 1 (meaning Yes) or 0 (meaning No).

# Main modeling assumption

We will focus on decision problems where the input is a string.

## For you to think about

1. Is focusing on **decision problems** a serious restriction?

What if we replaced the problem

Given numbers  $x, y$ , output  $x + y$ .

by the decision problem

Given numbers  $x, y, z$ , decide if  $x + y = z$ .

2. Is focusing on input **strings** a serious restriction?

- We can encode almost anything as a string.
- How would you encode an integer, or a set of integers, or a graph?

# Strings

## Definition

An **alphabet**  $\Sigma$  is a nonempty finite set of symbols.

## Example

- $\Sigma = \{0, 1\}$  is the classic **binary alphabet**.
- $\Sigma = \{a, b, c, \dots, z\}$  is the lower-case English alphabet.

# Strings

## Definition

A **string over  $\Sigma$**  is a finite sequence of symbols from  $\Sigma$ . The number of symbols in a string is its **length**.

## Example

- 0110 is a string of length 4 over alphabet  $\{0, 1\}$
- *bob* is a string of length 3 over alphabet  $\{a, b, c, \dots, z\}$ .
- If  $w$  has length  $n$  we write  $w = w_1w_2 \cdots w_n$  where each  $w_i \in \Sigma$ .
- There is only one string of length 0. It is denoted  $\epsilon$ .
- The set of all strings over  $\Sigma$  is denoted  $\Sigma^*$ .
  - The reason for this notation will be made clear later.

# Strings

## Concatenation of strings

- The *concatenation* of strings  $x, y$  is the string  $xy$  formed by appending  $y$  to the end of  $x$ .
  - The concatenation of  $x = 010$  and  $y = 01$  is  $01001$ .
- $w\epsilon = \epsilon w = w$  for all strings  $w$ .
  - $01\epsilon = 01$

# Decision problems

## Definition

A **decision problem** is a function  $P : \Sigma^* \rightarrow \{0, 1\}$ .

## Example

$$P(s) = \begin{cases} 1 & \text{if the length of the string } s \in \{0, 1\}^* \text{ is odd,} \\ 0 & \text{otherwise.} \end{cases}$$

We will also write the decision problem like this:

**Input:** String  $s$  over alphabet  $\{0, 1\}$ .

**Output:** 1 if the length of the string  $s$  is odd, and 0 otherwise.

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

## Example

The program

---

```
1 def L(s): # s binary-string
2     # a % b returns the remainder of a divided by b
3     return len(s)%2
```

---

solves the decision problem

**Input:** A string  $s$ .

**Output:** 1 if the length of the string  $s$  is odd, and 0 otherwise.

# Programs that solve problems

## Definition

A program **solves** a decision problem  $P : \Sigma^* \rightarrow \{0, 1\}$  if for every input string  $x \in \Sigma^*$ , the program on input  $x$  outputs  $P(x)$ .

## Example

The program

---

```
1 def Q(x): # binary string x encoding natural number
2         # least significant digit first
3     return x[0]
```

---

solves the decision problem

**Input:** A string encoding a natural number.

**Output:** 1 if the integer is odd, and 0 otherwise.

# What decision problems can be solved by programs?

**Input:** Strings encoding integers  $x, y, z$ .

**Output:** 1 if  $z = x + y$ , and 0 otherwise.

**Input:** Strings  $w, t$ .

**Output:** 1 if  $w$  is a substring of  $t$ , and 0 otherwise.

**Input:** Strings encoding integers  $x, y, z$ .

**Output:** 1 if  $z = x \times y$ , and 0 otherwise.

**Input:** String encoding an arithmetic expression  $e$ .

**Output:** 1 if  $e$  is well-bracketed, and 0 otherwise.

**Input:** String encoding a (Python) program  $p$ .

**Output:** 1 if  $p$  enters an infinite loop, and 0 otherwise.

# What decision problems can be solved by programs?

We will see that, in a very precise sense:

1. “addition” and “substring” can be decided with limited memory,
2. “multiplication” and “well-bracketed” require recursion,
3. “infinite loop” cannot be decided by any program.

To do this, we introduce **mathematical models** of programs that solve decision problems.

1. Limited memory: regular expressions and finite automata (L5-L7).
2. Plus recursion: context-free grammars and pushdown automata (L8-L9).
3. Universal: Turing machines and Lambda-calculus (L10-L11).

# One more modeling choice

We will think of decision problems as deciding if the input is in some **set of strings**.

## Example

Here are two ways of describing the same problem.

**Input:** A binary string encoding a natural number.

**Output:** 1 if the integer is odd, and 0 otherwise.

**Input:** A binary string encoding a natural number.

**Output:** 1 if the string is in the set  $\{w \in \{0, 1\}^* : w[0] = 1\}$ , and 0 otherwise.

Sets of strings are called **languages**. They are so important, they have their own field of study called **Formal Language Theory**.

# Languages

## Definition

A set  $L \subseteq \Sigma^*$  of strings is called a (formal) language over  $\Sigma$ .

## Example

Let  $\Sigma = \{a, b\}$ .

- $L_1 = \{x \in \Sigma^* : x \text{ starts with a } b\}$
- $L_2 = \{x \in \Sigma^* : x \text{ has an even number of } bs\}.$
- $L_3 = \{x \in \Sigma^* : x \text{ has the same number of } as \text{ as } bs\}.$
- The empty set  $\emptyset$ .
- **Note.** The empty set  $\emptyset$  has no elements in it, but the set  $\{\epsilon\}$  has one element in it, the empty string.

## For you to think about after class

1. Is HTML a formal language? If so, what is the alphabet and what are the strings?
2. Is music a formal language? If so, what is the alphabet and what are the strings?

# Important operations on languages

## Definition

Let  $A, B$  be languages over  $\Sigma$ .

- The **union** of  $A$  and  $B$ , written  $A \cup B$ , is the language  $\{x \in \Sigma^* : x \in A \text{ or } x \in B\}$ .
- The **concatenation** of  $A$  and  $B$ , written  $A \circ B$ , is the language  $\{xy \in \Sigma^* : x \in A, y \in B\}$ .
- Write  $A^k$  for  $\overbrace{A \circ A \circ A \cdots A}^k$
- The **star** of  $A$ , written  $A^*$ , is the language

$$\{\epsilon\} \cup A^1 \cup A^2 \cup A^3 \cup \dots$$

i.e.,  $A^* = \{x_1 x_2 \cdots x_k \in \Sigma^* : k \geq 0, k \in \mathbb{Z}, \text{each } x_i \in A\}$ .

# Important operations on languages

## Examples

$$1. \{a, ab\} \cup \{ab, aab\} =$$

$$2. \{a, ab\} \circ \{ab, aab\} =$$

$$3. \{a, ab\}^* =$$

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 5b: Regular Languages**

### **Chapter 1 of Sipser (eReserve)**

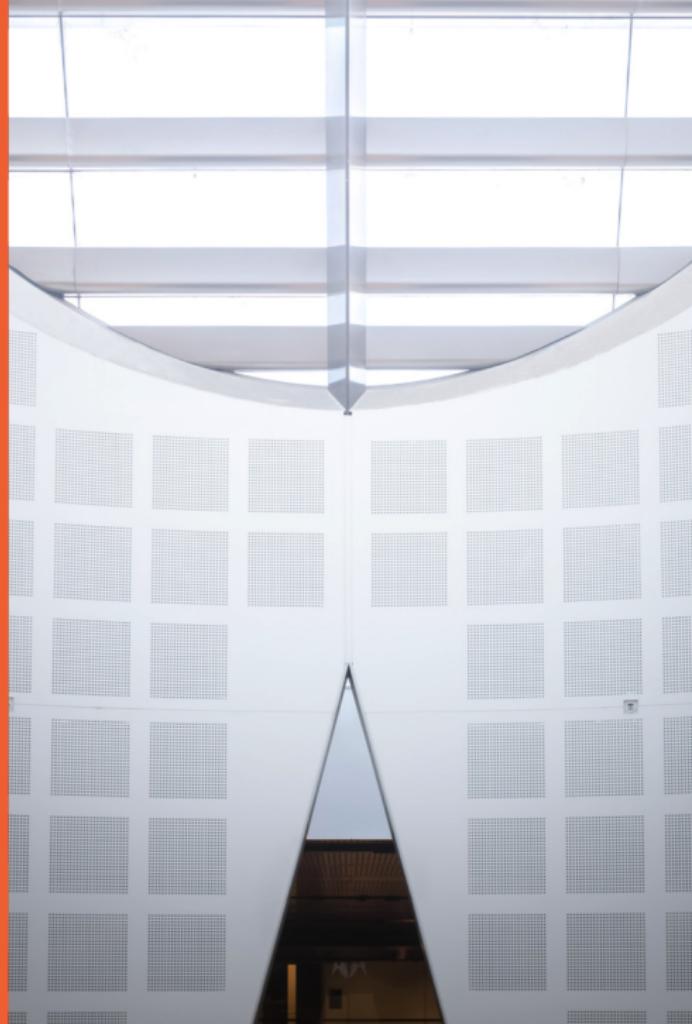
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# Regular expressions in a nutshell

- Expressions that describe languages
- Extremely useful
  - Pattern matching (Control-F)
  - Specification of data formats
  - Lexical analysis
- Based on three operations:
  - Language Union  $L_1 \cup L_2$
  - Language Concatenation  $L_1 \circ L_2$
  - Language Iteration/Kleene-star  $L^*$

# Regular expressions: Syntax

## Definition

Let  $\Sigma$  be an alphabet. The regular expressions over  $\Sigma$  are defined by the following recursive process:

- R1. The symbols  $\emptyset$  and  $\epsilon$  are regular expressions.
- R2. Each symbol  $a \in \Sigma$  is a regular expression.
- R3. If  $R_1, R_2$  are regular expressions then so is  $(R_1 \cup R_2)$
- R4. If  $R_1, R_2$  are regular expressions then so is  $(R_1 \circ R_2)$ , also written  $(R_1 R_2)$
- R5. If  $R$  is a regular expressions then so is  $R^*$ .<sup>1</sup>

---

<sup>1</sup>Note that Sipser writes  $(R^*)$ , although we don't need the parentheses in this case.

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ .

- $(a \cup \emptyset)$
- $(a \circ \epsilon)$
- $b^*$
- $(a^* \cup b^*)$
- $(a \cup b)^*$
- $(a \circ b)^*$

# Regular expressions: Semantics

A regular expression  $R$  *matches* certain strings, collectively called  $L(R)$ .

## Definition

Let  $\Sigma$  be an alphabet. The language  $L(R)$  **represented** by a regular expression  $R$  is defined by the following recursive procedure:

1.  $L(\emptyset) = \emptyset$  and  $L(\epsilon) = \{\epsilon\}$ .
2.  $L(a) = \{a\}$  for  $a \in \Sigma$ .
3.  $L((R_1 \cup R_2)) = L(R_1) \cup L(R_2)$ .
4.  $L((R_1 \circ R_2)) = L(R_1) \circ L(R_2)$ .
5.  $L(R^*) = L(R)^* = \{\epsilon\} \cup L(R)^1 \cup L(R)^2 \cup L(R)^3 \cup \dots$

# Regular expressions

## Notation.

- We **overload** the symbol  $\cup$ , i.e., it is a *symbol* used in regular expressions, and it is an *operation* on sets of strings. The same is true of other symbols like  $a$  and  $\epsilon$ .
  - We do this for convenience (otherwise we would have to use different notation for the symbols in the regular expressions and the operations on languages, e.g., some texts use  $+$  or  $|$  in regular expressions instead of  $\cup$ ).
- We may drop the outermost parentheses to improve readability.
  - E.g., we may write  $a \cup \emptyset$  instead of  $(a \cup \emptyset)$ , and  $a \circ b^*$  instead of  $(a \circ b^*)$ , which is different from  $(a \circ b)^*$ .
- We may write  $(R_1 \cup R_2 \cup R_3)$  instead of  $((R_1 \cup R_2) \cup R_3)$ , and similarly  $(R_1 \circ R_2 \circ R_3)$  instead of  $((R_1 \circ R_2) \circ R_3)$ .
  - The reason we can do this is, as we will see,  $\circ$  and  $\cup$  are associative.

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ .

- $L((a \cup \emptyset)) =$
- $L((a \circ \epsilon)) =$
- $L(b^*) =$
- $L((a^* \cup b^*)) =$
- $L((a \cup b)^*) =$
- $L((a \circ b)^*) =$

# Regular expressions

## Examples

Let  $\Sigma = \{a, b\}$ . Write regular expressions for the following languages:

1. The set of strings ending in  $a$ .
  - $((a \cup b)^* \circ a)$ , also written  $((a \cup b)^* a)$ .
2. The set of strings whose second last symbol is an  $a$ .
  - $((a \cup b)^* \circ a \circ (a \cup b))$
3. The set of strings that have  $aba$  as a substring.
  - $((a \cup b)^* \circ a \circ b \circ a \circ (a \cup b)^*)$
4. The set of strings of even length.
5. The set of strings with an even number of  $a$ 's.

# Regular expressions

## Important questions

1. Which languages can be described by regular expressions? All languages?
2. There are natural decision problems associated with regular expressions. Are there programs that solve them?

**Input:** Regular expression  $R$  and string  $s$

**Output:** 1 if  $s \in L(R)$ , and 0 otherwise.

**Input:** Regular expressions  $R_1, R_2$

**Output:** 1 if  $L(R_1) = L(R_2)$ , and 0 otherwise.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 5c: Finite Automata**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# Finite automata in a nutshell

A deterministic finite automaton is like a program that can only scan its input string once, and cannot write anything. At the end of reading the input string, it must decide to accept or reject the input string.

# Why study such a simple model of computation?

1. It is part of computing culture.
  - first appeared in McCulloch and Pitt's model of a neural network (1943)
  - then formalised by Kleene (American mathematician) as a model of stimulus and response
2. It has numerous practical applications.
  - lexical analysis (tokeniser)
  - pattern matching
  - communication protocols with bounded memory
  - circuits with feedback
  - finite-state reactive systems
  - finite-state controllers
  - non-player characters in computer games
  - ...
3. It is simple to implement/code.

# Drawing deterministic finite automata (DFA)

# Definition of a deterministic finite automaton (DFA)

## Definition

A deterministic finite automaton (DFA)  $M$  is a tuple  $(Q, \Sigma, \delta, q_0, F)$  where

1.  $Q$  is a finite set of states,
2.  $\Sigma$  is a finite set called the alphabet,
3.  $\delta : Q \times \Sigma \rightarrow Q$  is the transition function,
4.  $q_0 \in Q$  is the start state, and
5.  $F \subseteq Q$  is the set of accept states, sometimes called final states.

If  $q' = \delta(q, a)$  we write  $q \xrightarrow{a} q'$ , called a transition.

# Language described by a DFA

A DFA  $M$  describes a language  $L(M)$ : all strings that label paths from the start state to an accepting state.

## Examples

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1.  $\{aba\}$
2. all strings over  $\Sigma$
3. the set of strings ending in  $a$
4.  $L((a \circ b)^*)$
5. the set of strings with an even number of  $a$ 's
6. the set of strings with an odd number of  $b$ 's

# Designing DFAs

Draw a DFA for the language  $\{aba\}$

# Designing DFAs

Draw a DFA for the set of all strings over  $\{a, b\}$

# Designing DFAs

Draw a DFA for the set of all strings ending in  $a$

# Designing DFAs

Draw a DFA for the set of all strings matching the regular expression  $(ab)^*$

# Designing DFAs

Draw a DFA for the set of all strings with an even number of  $a$ 's

# Designing DFAs

Draw a DFA for the set of all strings with an odd number of  $b$ 's

# Definition of the language recognised by a DFA

## Definition

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a DFA, and let  $w = w_1 w_2 \cdots w_n$  be a string over  $\Sigma$ .

- A **run** of  $M$  on  $w$  is a sequence of transitions

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} q_2 \xrightarrow{w_3} \cdots \xrightarrow{w_n} q_n$$

- The run  $r$  is **accepting** if  $r_n \in F$ .
- If  $w$  has an accepting run then we say that  $M$  **accepts**  $w$ .

The set  $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$  is the **language recognised by  $M$** .

# DFAs

## Important questions

1. Which languages can be described by DFAs? All languages?
2. There are natural decision problems associated with DFAs. Are there programs that solve them?

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

## The problem

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

is solved by the program

---

```
1 def run(M,w):
2     cur = q_0
3     for i = 1 to len(w):
4         cur = δ(cur,w_i)
5     if cur in F:
6         return 1
7     else:
8         return 0
```

---

## Definition

A language  $L$  is called **regular** if  $L = L(M)$  for some DFA  $M$ .

## Theorem

*The regular languages are closed under complementation. That is, if  $A$  is regular, then  $\Sigma^* \setminus A$  is regular.*

## Example

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1. strings that contain  $aba$  as a substring
2. strings that do not contain  $aba$  as a substring

## Theorem

*The regular languages are closed under complementation. That is, if  $A$  is regular, then  $\Sigma^* \setminus A$  is regular.*

### Construction ("swap accept and reject states")

1. Given  $A = L(M)$  for some DFA  $M = (Q, \Sigma, \delta, q_0, F)$ .
2. We will construct a DFA  $M'$  recognising  $\Sigma^* \setminus L(M)$ .
3. Define  $M' = (Q, \Sigma, \delta, q_0, F')$  where  $F' = Q \setminus F$ .

## Theorem

*The regular languages are closed under union. That is, if A and B are regular, then  $A \cup B$  is regular.*

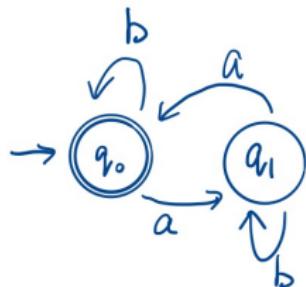
## Example

Let  $\Sigma = \{a, b\}$ . Draw DFAs for the following languages:

1. the strings consisting of an even number of a's
2. the strings consisting of an odd number of b's
3. the strings consisting of an even number of a's or an odd number of b's.

## Designing DFAs

Draw a DFA for the set of all strings with an even number of a's

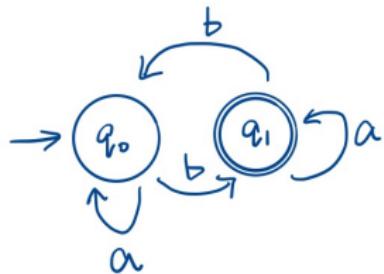


$q_0$  represents strings that contain even number of a's  
and arbitrary number of b's.

$q_1$  represents strings that contain odd number of a's  
and arbitrary number of b's.

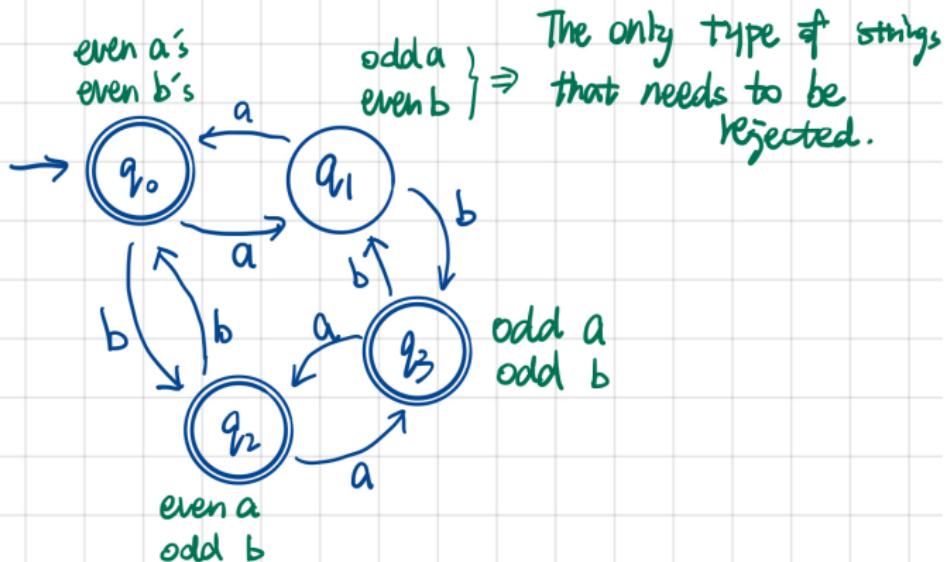
## Designing DFAs

Draw a DFA for the set of all strings with an odd number of b's



$q_0$  represents strings that contain even number of b's  
and arbitrary number of a's.

$q_1$  represents strings that contains odd number of b's  
and arbitrary number of a's.



## Theorem

The regular languages are closed under union. That is, if  $A_1$  and  $A_2$  are regular, then  $A_1 \cup A_2$  is regular.

## Construction ("product")

1. Given DFA  $M_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  recognising  $A_i$ ,  $i = 1, 2$ .
2. We will construct a DFA  $M$  recognising  $L(A_1) \cup L(A_2)$ .
3. Define  $M = (Q, \Sigma, \delta, q_0, F)$  where
  - $Q = Q_1 \times Q_2$ ,<sup>2</sup>
  - $\delta$  maps state  $(s_1, s_2)$  on input  $a \in \Sigma$  to state  $(\delta_1(s_1, a), \delta_2(s_2, a))$ ,
  - $q_0 = (q_1, q_2)$ ,
  - $F = \{(s_1, s_2) : s_1 \in F_1 \text{ or } s_2 \in F_2\}$ .

---

<sup>2</sup>Recall that  $Q_1 \times Q_2 = \{(s, t) : s \in Q_1, t \in Q_2\}$

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 6: Nondeterministic finite automata**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



## Theorem

*The regular languages are closed under concatenation. That is, if A and B are regular, then  $A \circ B$  is regular.*

Idea:

- Let  $M_A$  be a DFA recognising A, let  $M_B$  be a DFA recognising B.
- We want to build a DFA  $M$  for  $A \circ B$ .
- One way would be for  $M$  to simulate  $M_A$ , and at some point, as long as the state of  $M_A$  is an accepting state, “guess” that it is time to switch, and simulate  $M_B$  on the remaining input.

**Question.** How to we capture this “guess”?

We will introduce a model of computation, called a **nondeterministic finite automaton (NFA)**, which is like a DFA except that states can have more than one outgoing transition on the same input symbol. Later we will show how DFAs can simulate NFAs.

# Definition of NFA

## Definition

A **nondeterministic finite automaton**  $M$  is a tuple  $(Q, \Sigma, \delta, q_0, F)$  the same as a DFA except the transition function is<sup>1</sup>  $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ . If  $q' \in \delta(q, a)$  we write  $q \xrightarrow{a} q'$ .

---

<sup>1</sup> $\mathcal{P}(Q)$  is the set of subsets of  $Q$ . E.g.,  $\mathcal{P}(\{q_0, q_1\}) = \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}\}$ .

# Definition of NFA

- A **run** of an NFA  $M$  on string  $w$  is a sequence of transitions

$$q_0 \xrightarrow{y_1} q_1 \xrightarrow{y_2} q_2 \dots \xrightarrow{y_m} q_m$$

such that each  $y_i \in \Sigma \cup \{\epsilon\}$  and  $w = y_1 y_2 \dots y_m$ .<sup>2</sup>

- The run is **accepting** if  $q_m \in F$ .
- If  $w$  has at least one accepting run, then we say that  $w$  is **accepted** by  $M$ .
- The language **recognised** by  $M$  is  
$$L(M) = \{w \in \Sigma^* : w \text{ is accepted by } M\}.$$

That is: an NFA  $M$  describes the language  $L(M)$  of all strings that label paths from the start state to an accepting state.

---

<sup>2</sup>Recall that  $\epsilon x = x\epsilon = x$  for all strings  $x$ .

# Language described by NFAs

## Examples

Let  $\Sigma = \{a, b\}$ . Draw an NFA for the languages consisting of the set of strings whose last letter is an  $a$  (also, draw a DFA, for comparison).

# Comparing NFAs and DFAs

1. In a DFA, every input has exactly one run. The input string is accepted if this run is accepting.
2. In an NFA, an input may have zero, one, or more runs. The input string is accepted if at least one of its runs is accepting.
3. For every DFA  $M$  there is an NFA  $N$  such that  $L(M) = L(N)$ .
  - Given DFA  $M = (Q, \Sigma, \delta, q_0, F)$  with  $\delta : Q \times \Sigma \rightarrow Q$ , define the NFA  $N = (Q, \Sigma, \delta', q_0, F)$  where  $\delta'(q, a) = \{\delta(q, a)\}$ .
  - To see that  $L(M) = L(N)$  observe that the diagrams of  $M$  and  $N$  are the same.

# From Regular Expressions to NFAs

## Theorem

*For every regular expression  $R$  there is an NFA  $M_R$  such that  $L(R) = L(M_R)$ .*

The construction is by recursion on the structure of  $R$ . The base cases are  $R = \emptyset, R = \epsilon, R = a$  for  $a \in \Sigma$ . The recursive cases are  $R = (R_1 \circ R_2), R = (R_1 \cup R_2)$ , and  $R = R_1^*$ .

Base cases:

# NFAs are closed under $\circ$

## Lemma

If  $N_1, N_2$  are NFAs, there is an NFA  $N$  recognising  $L(N_1) \circ L(N_2)$ .

## Construction ("Simulate $N_1$ followed by $N_2$ ")

Given NFAs  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct NFA  $N$  with states  $Q_1 \cup Q_2$  as in the figure. The idea is that  $N$  guesses how to break the input into two pieces, the first accepted by  $N_1$ , and the second by  $N_2$ .

# NFAs are closed under $\circ$ (reference slide)

Here is the construction. From  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct  $N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \setminus F_1 \\ \delta_1(q, a) & q \in F_1, a \neq \epsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1, a = \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

# NFAs are closed under $\cup$

## Lemma

If  $N_1, N_2$  are NFAs, there is an NFA  $N$  recognising  $L(N_1) \cup L(N_2)$ .

## Construction ("Simulate $N_1$ or $N_2$ ")

Given NFAs  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct NFA  $N$  with states  $Q_1 \cup Q_2 \cup \{q_0\}$  as in the figure. The idea is that  $N$  guesses which of  $N_1$  or  $N_2$  to simulate.

## NFAs are closed under $\cup$ (reference slide)

Here is the construction. From  $N_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$  construct  $N = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup F_2)$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon \end{cases}$$

# NFAs are closed under \*

## Lemma

If  $N_1$  is an NFA, there is an NFA  $N$  recognising  $L(N_1)^*$ .

## Construction ("Simulate $N$ and go back to the initial state")

Given NFAs  $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  construct NFA  $N$  with states  $Q \cup \{q_0\}$  as in the figure. The idea is that  $N$  guesses how to break the input into pieces, each of which is accepted by  $N_1$ .

# NFAs are closed under \* (reference slide)

Here is the construction. From  $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$  construct  $N = (Q_1 \cup \{q_0\}, \Sigma, \delta, q_0, F_1 \cup \{q_0\})$  where

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & (q \in Q_1 \setminus F_1) \text{ or } (q \in F_1, a \neq \epsilon) \\ \delta_1(q, a) \cup \{q_1\} & q \in F_1, a = \epsilon \\ \{q_1\} & q = q_0, a = \epsilon \\ \emptyset & q = q_0, a \neq \epsilon \end{cases}$$

## From RE to NFA: example

Convert the regular expression  $((a \cup b)^* \circ a)$  to an NFA (using the construction just given).

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 7a: NFAs, DFAs, REs have the same expressive power**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# Where are we going?

We are going to show that DFA, NFA and regular expressions have the same expressive power. I.e., they recognise the same languages.

1. From Regular Expressions to NFAs
2. From NFAs to NFAs without  $\epsilon$ -transitions
3. From NFAs without  $\epsilon$ -transitions to DFAs
4. From DFAs to Regular Expressions

# From NFAs to NFAs without $\epsilon$ -transitions

## Theorem

For every NFA  $N$  there is an NFA  $N'$  without  $\epsilon$ -transitions such that  $L(N) = L(N')$ .

## Construction ("skip over $\epsilon$ -transitions")

Given NFA  $N = (Q, \Sigma, \delta, q_0, F)$  write  $s \xrightarrow{\epsilon^*} t$  if there is a path from  $s$  to  $t$  labelled by a string from  $\epsilon^*$ . Construct

$N' = (Q, \Sigma, \delta', q_0, F')$  where

- $t \in \delta'(s, a)$  if, in  $N$ , there is a state  $q$  such that  $s \xrightarrow{\epsilon^*} q \xrightarrow{a} t$ .
- $s \in F'$  if, in  $N$ , there is a state  $q \in F$  such that  $s \xrightarrow{\epsilon^*} q$ .

The idea is that  $N'$  skips over the  $\epsilon$ -transitions of  $N$ .

# From NFAs to NFAs without $\epsilon$ -transitions

## Example

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 7a: NFAs, DFAs, REs have the same expressive power**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# Where are we going?

We are going to show that DFA, NFA and regular expressions have the same expressive power. I.e., they recognise the same languages.

1. From Regular Expressions to NFAs
2. From NFAs to NFAs without  $\epsilon$ -transitions
3. From NFAs without  $\epsilon$ -transitions to DFAs
4. From DFAs to Regular Expressions

# From NFAs without $\epsilon$ -transitions to DFAs

## Theorem

*For every NFA  $N$  without  $\epsilon$ -transitions there is a DFA  $M$  such that  $L(M) = L(N)$ .*

## Construction ("subset construction")

Intuitively,  $M$  keeps track of all possible states that  $N$  can be in, and  $M$  accepts if at least one of these states is final.

# From NFAs without $\epsilon$ -transitions to DFAs

## Example

# From NFAs without $\epsilon$ -transitions to DFAs (Reference Slide)

Given NFA  $N = (Q, \Sigma, \delta, q_0, F)$  the subset construction builds a DFA  $M = (\mathcal{P}(Q), \Sigma, \delta', \{q_0\}, F')$  where

- for  $X \subseteq Q$ ,  $\delta'(X, a) = \cup_{q \in X} \delta(q, a)$ ,
- $F' = \{X \subseteq Q : X \cap F \neq \emptyset\}$ .

# From DFAs to Regular Expressions

## Theorem

*For every DFA  $M$  there is a regular expression  $R$  such that  $L(M) = L(R)$ .*

Idea:

1. We convert  $M$  into a **generalised nondeterministic finite automaton (GNFA)** which is like an NFA except that it can have regular expressions label the transitions.
2. We **successively remove states** from the automaton, until there is just one transition (from the start state to the final state).
3. We return the regular expression on this last transition.

# GNFAs

- A **run** of a GNFA  $N$  on string  $w$  is a sequence of transitions

$$q_0 \xrightarrow{R_1} q_1 \xrightarrow{R_2} q_2 \dots \xrightarrow{R_m} q_m$$

in  $M$  such that  $w$  matches the reg exp  $R_1 R_2 \dots R_m$ .

- The run is **accepting** if  $q_m \in F$ . The language **recognised** by  $N$  is  $L(N) = \{w \in \Sigma^* : w \text{ is accepted by } N\}$ .

## Example

# From DFAs to Regular Expressions

Generalised NFAs. We make sure that:

1. the initial state has no incoming edges.
2. there is one final state, and it has no outgoing edges.
3. there are edges from every state (that is not final) to every state (that is not initial).

We show how to translate DFAs into GNFAs, and then successively remove states from the GNFA until there is just a single edge left.

# From DFAs to GNFAs

For every DFA  $M$  there is a GNFA  $N$  such that  $L(M) = L(N)$ .

1. Add an initial state and  $\epsilon$ -transition to the old intial state.
2. Add a final state and  $\epsilon$ -transitions to it from all the old final states.
3. Add transitions for every pair of states, including from a state to itself, (except leaving the final, or entering the new).

# From DFAs to GNFAs

## Example

# Removing states from GNFA

For every GNFA  $N$  with  $> 2$  states there is a GNFA  $N'$  with one less state such that  $L(N) = L(N')$ .

1. Pick a state  $q$  to eliminate.
2. For every pair of remaining states  $(s, t)$ , replace the label from  $s$  to  $t$  by the regular expression

$$(R_{s,t} \cup (R_{s,q} \circ R_{q,q}^* \circ R_{q,t})).$$

where  $R_{x,y}$  is the regular expression labeling the transition from state  $x$  to state  $y$ .

# Removing states from GNFAs

## Example

# Summary

1. The following models of computation describe the same languages: DFAs, NFAs, Regular Expressions.
2. The regular languages are closed under the Boolean operations union, intersection, complementation, as well as concatenation and Kleene star.

# Decision problems about regular languages

We now look at some decision problems which take automata/regular-expressions as input.

# Decision Problems about DFAs

**Input:** DFA  $M$  and string  $w$

**Output:** 1 if  $w \in L(M)$ , and 0 otherwise.

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

# Emptiness problem for DFAs

**Input:** DFA  $M$

**Output:** 1 if  $L(M) = \emptyset$ , and 0 otherwise.

## Algorithm ("mark reachable states")

1. Mark the states using the following procedure.
  - Mark the initial state.
  - If  $q$  is marked and  $q \xrightarrow{a} p$  (for some  $a \in \Sigma$ ), then mark  $p$ .
  - Stop when no new states are marked.
2. Return 0 if a final state is marked, else return 1.

Marking a state can be thought of as putting it in a set called "marked".

# Equivalence problem for DFAs

**Input:** DFAs  $M_1, M_2$

**Output:** 1 if  $L(M_1) = L(M_2)$ , and 0 otherwise.

## Algorithm

1. Form DFA  $M$  recognising
$$(L(M_1) \setminus L(M_2)) \cup (L(M_2) \setminus L(M_1)).$$
  - Note that  $L(M_1) = L(M_2)$  iff  $L(M) = \emptyset$ .
2. So we just need a way to check if  $L(M) = \emptyset$ . Use the algorithm for the Emptiness problem for DFAs.

# Decision Problems about Regular Expressions

**Input:** Regular expression  $R$  and string  $w$

**Output:** 1 if  $w \in L(R)$ , and 0 otherwise.

**Input:** Regular expressions  $R_1, R_2$

**Output:** 1 if  $L(R_1) = L(R_2)$ , and 0 otherwise.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 7b: Non-regularity**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



To show that a language is regular, it is sufficient to find a DFA, NFA, or Regular Expression for it. To show that a language  $L$  is not regular, one must show that there is **no DFA** that recognises it. This can be done using a **fooling argument**.

1. Assume there is a DFA  $M$  recognising the language  $L$ .
2. Reason about the transitions of this DFA and find a string not in the language which it accepts (or a string in the language which it rejects).
3. This contradicts the assumption that  $M$  recognises  $L$ .
4. Conclude that there is no such DFA.

# Pigeonhole principle (PHP)

If there are  $n$  holes and  $> n$  objects to put in the holes, then at least one hole must get at least two objects.

How will we apply this?

$L = \{a^i b^i : i \geq 0\}$  is not regular

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $n \neq m$  such that  $f(a^n) = f(a^m)$ .
4. But  $a^n b^n \in L$  means that the path from state  $f(a^n)$  labeled  $b^n$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $a^m b^n$ . So  $a^m b^n$  is accepted by  $M$ .
6. But since  $m \neq n$ ,  $M$  accepts a word not in  $L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

$L = \{ww : w \in \{a,b\}^*\}$  is not regular

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $u \neq v$  such that  $|u| = |v|$  and  $f(u) = f(v)$ .
4. But  $uu \in L$  means that the path from state  $f(u)$  labeled  $u$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $vu$ . So  $vu$  is accepted by  $M$ .
6. But since  $u \neq v$  and  $|u| = |v|$ ,  $vu \notin L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

$$L = \{u \in \{a, b\}^* : |u| \text{ is a power of } 2\}$$

## Proof

1. Assume there is a DFA  $M$  that recognises  $L$ .
2. Write  $f(w)$  for the state that  $M$  reaches after reading input  $w$ .
3. By PHP, there exists  $u, v$  such that  $f(u) = f(v)$  and  $|u| = 2^n, |v| = 2^m, n < m$ .
4. But  $uu \in L$  since  $2^n + 2^n = 2^{n+1}$ , so the path from state  $f(u)$  labeled  $u$  ends in a final state.
5. So there is a path from the initial state to a final state labeled  $vu$ . So  $vu$  is accepted by  $M$ .
6. But  $vu$  is not in  $L$  since  $2^m < |vu| = 2^m + 2^n < 2^m + 2^m = 2^{m+1}$ . So  $M$  accepts a word not in  $L$ .
7. This contradicts the assumption that  $M$  recognises  $L$ .

## Other techniques

Once we know that a language is not regular, we can deduce that other languages are not regular using the closure properties of regular languages.

# Showing a language is not regular

## Example

We have seen that  $L = \{a^i b^i : i \geq 0\}$  is not regular. Use this fact to prove that the set  $L'$  of strings with the same number of *as* as *bs* is not regular.

1. Assume  $L'$  is regular.
2. Then  $L' \cap L(a^*b^*)$  is regular since the intersection of regular languages is regular.
3. But  $L = L' \cap L(a^*b^*)$ , which we know is not regular.
4. This contradicts the assumption that  $L'$  is regular.
5. So  $L'$  is not regular.

# Showing a language is not regular

## Example

We have seen that the language  $L'$  consisting of strings over  $\{a, b\}$  with the same number of  $a$ s as  $b$ s is not regular. Use this fact to prove that the language  $L''$  consisting of strings over  $\{a, b\}$  with a different number of  $a$ s as  $b$ s is not regular.

1. Assume  $L''$  is regular.
2. Then  $\{a, b\}^* \setminus L''$  is regular since the complement of a regular language is regular.
3. But  $L' = \{a, b\}^* \setminus L''$ , which we know is not regular.
4. This contradicts the assumption that  $L''$  is regular.
5. So  $L''$  is not regular.

# Where are we going?

Next week we start learning about more powerful models of computation that can recognise non-regular languages. We start with context-free grammars.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 8: Context-free Grammars**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# Course topics

Here are some important mathematical models of computation:

1. Propositional Logic, Predicate Logic
2. Finite Automata, Regular Expressions, Regular Grammars
3. Pushdown Automata, Context-free grammars
4. Turing Machines, Lambda Calculus

What questions about computation can be answered with mathematical models?

# What problems can be solved by programs with very limited memory?

- E.g., Basic pattern matching, lexical analysis
- COMP3109: Programming Languages and Paradigms
- Model: Finite automata, regular grammars

# What about parentheses matching?

$\{0^n 1^n : n \geq 0\}$  is not regular.

```
<html>
  <head>
    <title>
      My webpage!
    </title>
  </head>
</html>
```

# What problems can be solved by programs with very limited memory but allowing recursion?

- E.g., Sophisicated string processing, parsing
- COMP3109:Programming Languages and Paradigms
- Model: Context-free grammars, Pushdown automata.

# Theme

- State-machines recognise languages.
- Grammars generate languages.

# Grammars in a nutshell

- A grammar is a set of rules which can be used to **generate/derive** strings
- The language of the grammar is all strings that can be derived by the grammar

# Context-free grammars

## Program Syntax

statements: statement+

statement : compound\_stmt | simple\_stmt

## Document Description Definition

<!ELEMENT NEWSPAPER (ARTICLE+)>

<!ELEMENT ARTICLE (STORY | ADVERT) >

## Our Syntax

$$S \rightarrow TS$$

$$T \rightarrow C|D$$

# Context-free grammars

$$S \rightarrow 0S1$$

$$S \rightarrow T$$

$$T \rightarrow \epsilon$$

This grammar generates the language  $L = \{0^n 1^n \mid n \geq 0\}$ .

How does it derive 000111?

# Context-free grammars

$S \rightarrow NounPhrase\ VerbPhrase$

$NounPhrase \rightarrow \text{the}\ Noun$

$VerbPhrase \rightarrow Verb\ NounPhrase$

$Noun \rightarrow \text{girl} \mid \text{ball}$

$Verb \rightarrow \text{likes} \mid \text{sees}$

This grammar generates the language:

- {    the girl likes the girl,    the girl likes the ball,  
      the girl sees the girl,    the girl sees the ball,  
      the ball likes the girl,    the ball likes the ball,  
      the ball sees the girl,    the ball sees the ball    }

# Context-free grammars

Anatomy of a grammar:

- **Variables** aka **Non-terminals**: used to generate strings
- **Start variable**: variable used to start every derivation
- **Terminals**: alphabet of symbols making up strings in the language
- **Rules** aka **Production rules**:

$$A \rightarrow u$$

where  $A$  is a variable and  $u$  is string of variables and terminals

A variable can have many rules:      They can be written together:

$$\text{Noun} \rightarrow \text{girl}$$

$$\text{Noun} \rightarrow \text{ball}$$

$$\text{Noun} \rightarrow \text{girl} \mid \text{ball}$$

# Context-free grammars

$$S \rightarrow a \mid b \mid \epsilon \mid \emptyset \mid (S \cup S) \mid (S \circ S) \mid S^*$$

- Variable  $S$
- Terminals  $a, b, \epsilon, \emptyset, (, ), \cup, \circ, ^*$

# Context-free grammars

$$S \rightarrow p \mid q \mid (S \wedge S) \mid (S \vee S) \mid \neg S$$

- Variable  $S$
- Terminals  $p, q, (,), \wedge, \vee, \neg$

# Context-free grammars

## Notation

- $A, B, C, \dots$  and  $S$  are variables
- $S$  is the start variable
- $a, b, c, \dots$  are terminals
- $u, v, w, \dots$  are strings of terminals and variables

# Context-free grammars: Syntax

## Definition

A *context-free grammar*  $G$  is a 4-tuple  $(V, \Sigma, R, S)$  where:

- $V$  is a finite set of *variables*
- $\Sigma$  is a finite set of *terminals*
- $R$  is a finite set of *rules* in the form  $A \rightarrow v$  where  $A \in V$  and  $v \in (V \cup \Sigma \cup \{\varepsilon\})^*$
- $S \in V$  is a special variable called the *start variable*

# Context-free grammars: Semantics

## Definition

- If  $A \rightarrow w$  is a rule and  $u, v$  are strings of terminals and variables then  $uAv \Rightarrow uwv$ .  
The symbol  $\Rightarrow$  is read **yields**.
- $u \Rightarrow^* v$  means  $u$  yields  $v$  in zero or more steps  
The symbol  $\Rightarrow^*$  is read **derives**.
- The language **generated** by  $G$  is the set of strings over  $\Sigma$  that are derived from the start variable  $S$ .

$$L(G) = \{u \in \Sigma^* : S \Rightarrow^* u\}$$

Sometimes we will use  $u \Rightarrow^+ v$  to mean that  $u$  yields  $v$  in at least one step.

# Derivation of a string

- Begin with the start variable
- Repeatedly replace one variable with the right hand side of one of its productions
- ... until the string is composed only of terminal symbols

Example, derivation of 000111 from this grammar:

$$S \rightarrow 0S1 \mid \varepsilon$$

$$S \Rightarrow 0S1$$

$$\Rightarrow 00S11$$

$$\Rightarrow 000S111$$

$$\Rightarrow 000111$$

# Leftmost and Rightmost Derivations

*Leftmost derivation:* always derive the leftmost variable first

*Rightmost derivation:* always derive the rightmost variable first

Example: “the girl sees the ball”

$$\begin{aligned} S &\Rightarrow NounPhrase\ VerbPhrase \\ &\Rightarrow \text{the } Noun\ VerbPhrase \\ &\Rightarrow \text{the girl } VerbPhrase \\ &\Rightarrow \text{the girl } Verb\ NounPhrase \\ &\Rightarrow \text{the girl sees } NounPhrase \\ &\Rightarrow \text{the girl sees the } Noun \\ &\Rightarrow \text{the girl sees the ball} \end{aligned}$$
$$\begin{aligned} S &\Rightarrow NounPhrase\ VerbPhrase \\ &\Rightarrow NounPhrase\ Verb\ NounPhrase \\ &\Rightarrow NounPhrase\ Verb\ the\ Noun \\ &\Rightarrow NounPhrase\ Verb\ the\ ball \\ &\Rightarrow NounPhrase\ sees\ the\ ball \\ &\Rightarrow \text{the } Noun\ sees\ the\ ball \\ &\Rightarrow \text{the girl sees the ball} \end{aligned}$$

# Constructing grammars

Let  $M$  and  $N$  be two languages whose grammars have disjoint sets of non-terminals (rename them if necessary). Let  $S_M$  and  $S_N$  be their start variables. Then we can construct a grammar recognising the following languages, with a new start variable  $S$ :

- Union: the grammar for  $M \cup N$  starts with  $S \rightarrow S_M \mid S_N$
- Concatenation: the grammar for  $MN$  starts with  $S \rightarrow S_MS_N$
- Star closure: the grammar for  $M^*$  starts with  $S \rightarrow S_MS \mid \varepsilon$

All other productions remain unchanged (aside for renaming of variables as needed)

# Using the union rule

Let  $L = \{\varepsilon, a, b, aa, bb, \dots, a^n, b^n, \dots\}$

Then  $L = M \cup N$  where  $M = \{a^n \mid n \geq 0\}, N = \{b^n \mid n \geq 0\}$

So a grammar  $G_M$  of  $M$  is  $S_M \rightarrow \varepsilon \mid aS_M$   
and a grammar  $G_N$  of  $N$  is  $S_N \rightarrow \varepsilon \mid bS_N$

Using the union rule we get:

$$S \rightarrow S_M \mid S_N$$

$$S_M \rightarrow \varepsilon \mid aS_M$$

$$S_N \rightarrow \varepsilon \mid bS_N$$

# Using the concatenation rule

Let  $L = \{a^m b^n \mid m \geq 0, n \geq 0\}$

Then  $L = MN$  where  $M = \{a^m \mid m \geq 0\}$ ,  $N = \{b^n \mid n \geq 0\}$

So a grammar  $G_M$  of  $M$  is  $S_M \rightarrow \varepsilon \mid aS_M$   
and a grammar  $G_N$  of  $N$  is  $S_N \rightarrow \varepsilon \mid bS_N$

Using the concatenation rule we get:

$$S \rightarrow S_M S_N$$

$$S_M \rightarrow \varepsilon \mid aS_M$$

$$S_N \rightarrow \varepsilon \mid bS_N$$

## Using the star closure rule

Let  $L$  be strings consisting of 0 or more occurrences of  $aa$  or  $bb$ ,  
i.e.  $(aa \mid bb)^*$

Then  $L = M^*$  where  $M = \{aa, bb\}$

So a grammar  $G_M$  of  $M$  is  $S_M \rightarrow aa \mid bb$

Using the star closure rule we get:

$$\begin{aligned}S &\rightarrow S_M S \mid \epsilon \\S_M &\rightarrow aa \mid bb\end{aligned}$$

# Context-Free Languages

## Definition

A language is *context-free* if it is generated by a CFG.

## Facts.

- Every regular language is context-free (why?)
- The *union* of two CFL is also context-free
- The *concatenation* of two CFL is also context-free
- The *star closure* of a CFL is also context-free

# Example

Consider the grammar  $G$ :

$$\begin{array}{l} S \rightarrow AB \\ A \rightarrow \varepsilon \mid aA \\ B \rightarrow \varepsilon \mid bB \end{array}$$

What is  $L(G)$ ?

$$\begin{aligned} S &\Rightarrow AB \\ &\Rightarrow aAB \\ &\Rightarrow^* aaaaAB \\ &\Rightarrow aaaaB \\ &\Rightarrow aaaabB \\ &\Rightarrow^* aaaabbbbbB \\ &\Rightarrow aaaabbbbb \end{aligned}$$

i.e.  $L(G) = L(a^*b^*) = \{a^n b^m \mid n \geq 0, m \geq 0\}$

# Parsing

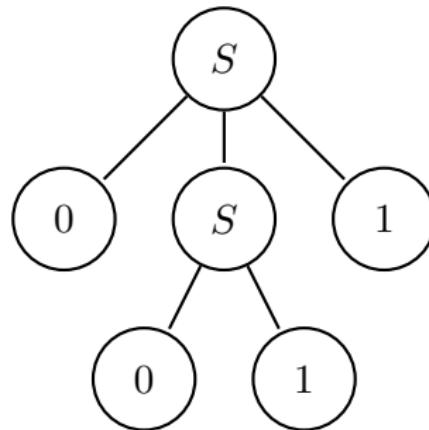
The problem of *parsing* is determining *how* the grammar generates a given string.

# Parse Tree

A *parse tree* is a tree labelled by symbols from the CFG

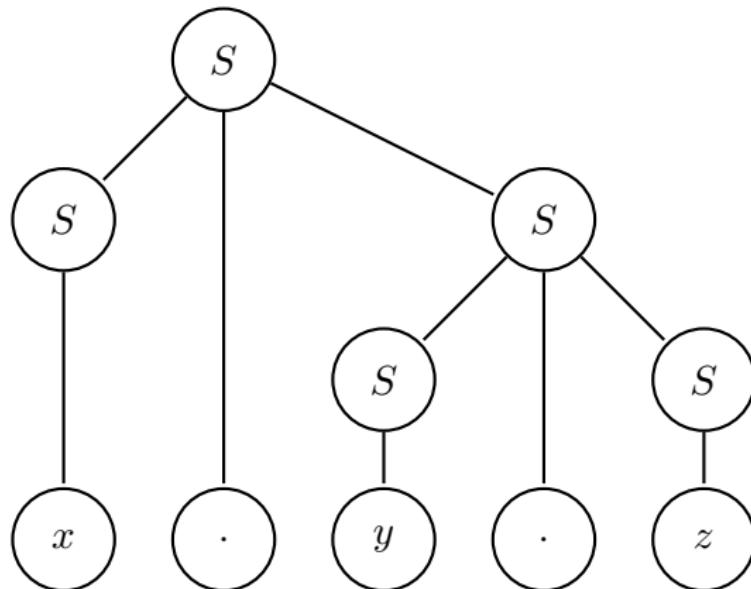
- root = the start variable
- interior node = a variable
- leaf node = a terminal or  $\epsilon$
- children of  $X$  = the right hand side of a production rule for  $X$ , in order

- Example parse tree for “0011” in  $S \rightarrow 0S1 \mid 01$
- An in-order traversal of the leaf nodes retrieves the string



# Parse Tree or Derivation Tree

The parse tree defines the *meaning* of a string in the grammar's language.



$$S \rightarrow S \cdot S$$

$$S \rightarrow x \mid y \mid z$$

This parse tree says that the expression means  $x \cdot (y \cdot z)$  rather than  $((x \cdot y) \cdot z)$ .

# Natural Language Processing (NLP) example

$S \rightarrow NounPhrase\ VerbPhrase$

$NounPhrase \rightarrow ComplexNoun \mid ComplexNoun\ PrepPhrase$

$VerbPhrase \rightarrow ComplexVerb \mid ComplexVerb\ PrepPhrase$

$PrepPhrase \rightarrow Prep\ ComplexNoun$

$ComplexNoun \rightarrow Article\ Noun$

$ComplexVerb \rightarrow Verb \mid Verb\ NounPhrase$

$Article \rightarrow a \mid the$

$Noun \rightarrow girl \mid dog \mid stick \mid ball$

$Verb \rightarrow chases \mid sees$

$Prep \rightarrow with$

# Ambiguity: example

Ambiguity: several meanings for the same sentence.

"The girl chases the dog with a stick" has *two leftmost derivations*

*Sentence*  $\Rightarrow$  *NounPhrase VerbPhrase*

$\Rightarrow$  *ComplexNoun VerbPhrase*

$\Rightarrow$  *Article Noun VerbPhrase*

$\Rightarrow$  *the Noun VerbPhrase*

$\Rightarrow$  *the girl VerbPhrase*

$\Rightarrow$  *the girl ComplexVerb*

$\Rightarrow$  *the girl Verb NounPhrase*

$\Rightarrow$  *the girl chases NounPhrase*

$\Rightarrow$  *the girl chases ComplexNoun PrepPhrase*

$\Rightarrow$  *the girl chases Article Noun PrepPhrase*

$\Rightarrow$  *the girl chases the Noun PrepPhrase*

$\Rightarrow$  *the girl chases the dog PrepPhrase*

$\Rightarrow$  *the girl chases the dog Prep ComplexNoun*

$\Rightarrow$  *the girl chases the dog with ComplexNoun*

$\Rightarrow$  *the girl chases the dog with Article Noun*

$\Rightarrow$  *the girl chases the dog with a Noun*

$\Rightarrow$  *the girl chases the dog with a stick*

*Sentence*  $\Rightarrow$  *NounPhrase VerbPhrase*

$\Rightarrow$  *ComplexNoun VerbPhrase*

$\Rightarrow$  *Article Noun VerbPhrase*

$\Rightarrow$  *the Noun VerbPhrase*

$\Rightarrow$  *the girl VerbPhrase*

$\Rightarrow$  *the girl ComplexVerb PrepPhrase*

$\Rightarrow$  *the girl Verb NounPhrase PrepPhrase*

$\Rightarrow$  *the girl chases NounPhrase PrepPhrase*

$\Rightarrow$  *the girl chases Article Noun PrepPhrase*

$\Rightarrow$  *the girl chases the Noun PrepPhrase*

$\Rightarrow$  *the girl chases the dog PrepPhrase*

$\Rightarrow$  *the girl chases the dog Prep ComplexNoun*

$\Rightarrow$  *the girl chases the dog with ComplexNoun*

$\Rightarrow$  *the girl chases the dog with Article Noun*

$\Rightarrow$  *the girl chases the dog with a Noun*

$\Rightarrow$  *the girl chases the dog with a stick*

# Ambiguity: example

Ambiguity: several meanings for the same sentence.

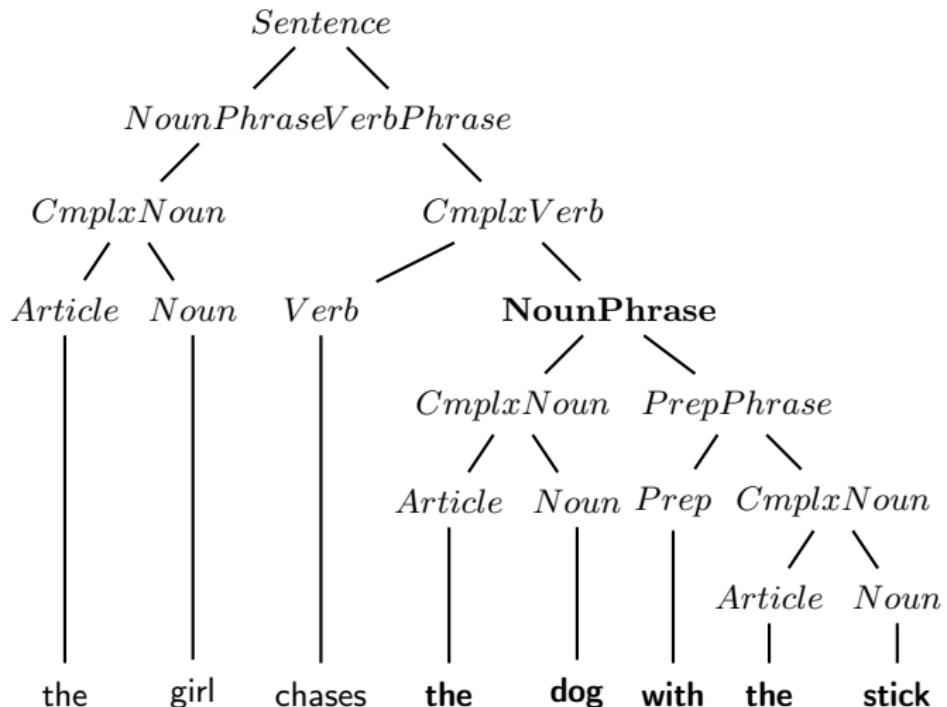
“The girl chases the dog with a stick” has *two leftmost derivations*

*Sentence*  $\Rightarrow^*$  the girl *VerbPhrase*  
 $\Rightarrow$  the girl *ComplexVerb*  
 $\Rightarrow^*$  the girl chases the dog with a stick

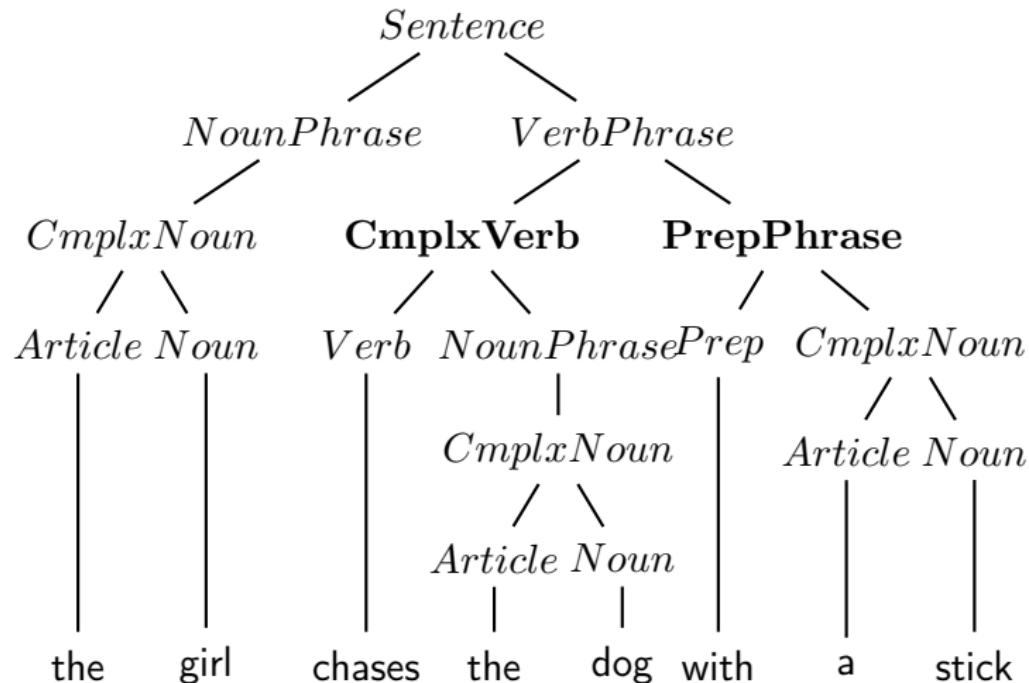
*Sentence*  $\Rightarrow^*$  the girl *VerbPhrase*  
 $\Rightarrow$  the girl *ComplexVerb PrepPhrase*  
 $\Rightarrow^*$  the girl chases the dog with a stick

Who has the stick?

## First Leftmost Derivation Tree



## Second Leftmost Derivation Tree



# Ambiguous Grammars

## Definition

- A string is *ambiguous* on a given grammar if it has two different parse trees.
- A grammar is *ambiguous* if it contains an ambiguous string.

## Good to know

- Each parse tree corresponds to one leftmost derivation.
- So, a string is ambiguous if it has two leftmost derivations.
- The same thing is true of rightmost derivations.

# Is this grammar ambiguous?

$$E \rightarrow E - E$$

$$E \rightarrow a \mid b \mid c$$

Rightmost derivations of  $a - b - c$ :

$$E \Rightarrow E - E$$

$$\Rightarrow E - c$$

$$\Rightarrow E - E - c$$

$$\Rightarrow E - b - c$$

$$\Rightarrow a - b - c$$

$$E \Rightarrow E - E$$

$$\Rightarrow E - E - E$$

$$\Rightarrow E - E - c$$

$$\Rightarrow E - b - c$$

$$\Rightarrow a - b - c$$

i.e.  $(a - b) - c$

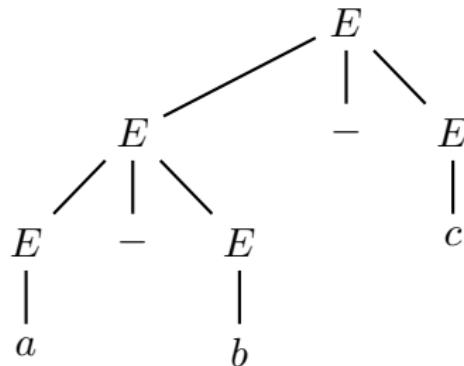
i.e.  $a - (b - c)$

# Is this grammar ambiguous?

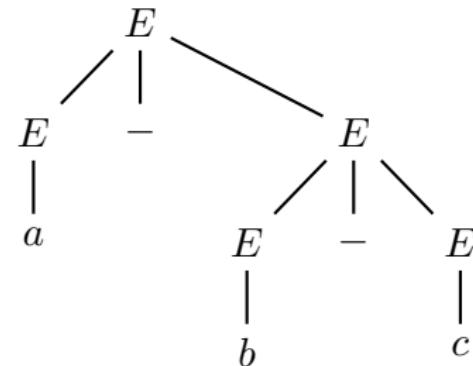
$$E \rightarrow E - E$$

$$E \rightarrow a \mid b \mid c$$

Rightmost derivations of  $a - b - c$ :



i.e.  $(a - b) - c$



i.e.  $a - (b - c)$

# Removing ambiguity (example)

Suppose we want  $a - b - c$  to always mean  $(a - b) - c$ ?

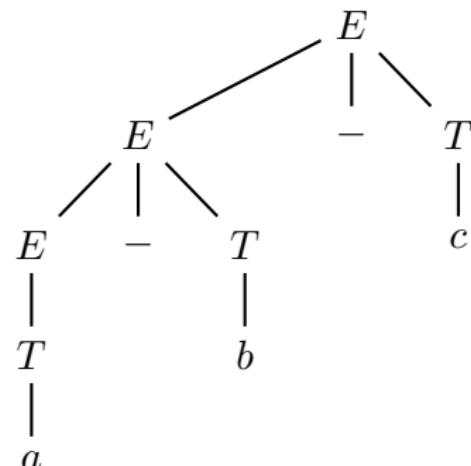
Introduce a new nonterminal symbol  $T$ :

$$E \rightarrow E - T \mid T$$

$$T \rightarrow a \mid b \mid c$$

Now the only rightmost derivation of  $a - b - c$  is:

$$\begin{aligned} E &\Rightarrow E - T \\ &\Rightarrow E - c \\ &\Rightarrow E - T - c \\ &\Rightarrow E - b - c \\ &\Rightarrow T - b - c \\ &\Rightarrow a - b - c \end{aligned}$$



# Types of grammars

The Chomsky Hierarchy consists of 4 classes of grammars, depending on the type of production rules that they allow:

Type 0 (unrestricted)	$\chi \rightarrow \alpha$
Type 1 (context-sensitive)	$uAv \rightarrow u\chi v$ (allowing $S \rightarrow \epsilon$ )
Type 2 (context-free)	$A \rightarrow u$
Type 3 (regular)	$A \rightarrow cB$ and $A \rightarrow d$

- $\chi$  string of one or more variables and terminals, excluding  $\epsilon$ .
- $u, v$  string of variables and terminals, including  $\epsilon$
- $A, B$  variables
- $c, d$  terminals

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 9a: Chomsky Normal Form**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



- A context-free grammar (CFG) generates strings by rewriting.
- Today we will see how to tell if a given context-free grammar (CFG) generates a given string.
- Here is the decision problem:

**Input:** a CFG  $G$  and string  $w$ .

**Output:** 1 if the string  $w$  is generated by  $G$ , and 0 otherwise.

- This basic problem is solved by compilers and parsers.

# Possible approaches...

1. Systematically search through all derivations (or all parse-trees) until you find one that derives  $w$ .
  - Try all  $i$ -step derivations for  $i = 1, 2, 3, \text{etc.}$
  - Problem: When to stop and declare “the string  $w$  cannot be derived from  $G$ ”?
  - This problem can be fixed, but the resulting algorithm takes exponential time in the worst case, i.e., is **very slow**.
2. Use a table-filling algorithm (aka tabulation, aka dynamic programming).
  - Systematically compute, for every substring  $v$  of  $w$ , which non-terminals of  $G$  derive  $v$  (if any).
  - Then check if the start state  $S$  is in the set computed for the whole string  $w$ .
  - The resulting algorithm takes polynomial time in the worst case, i.e., is **acceptably fast**.
  - This is the approach we will take today! It is called the CYK algorithm

**Input:** a CFG  $G$  and string  $w$ .

**Output:** 1 if the string  $w$  is generated by  $G$ , and 0 otherwise.

## Plan

1. Convert  $G$  into a grammar  $G'$  such that  $L(G) = L(G')$  and  $G'$  is in a special normal form called Chomsky Normal Form.
2. Apply the table-filling algorithm to  $G'$  and  $w$ , and read off the answer.

## Skeptic

Q. Why do we do the normal form?

A. In order to make the table small and easier to implement.

E.g., the parse-trees of a grammar in CNF are binary trees!

# Chomsky Normal Form

## Definition

A grammar  $G$  is in **Chomsky Normal Form (CNF)** if every rule is in one of these forms:

- $A \rightarrow BC$  ( $A, B, C$  are any variables, except that neither  $B$  nor  $C$  is the start symbol)
- $A \rightarrow a$  ( $A$  is any variable and  $a$  is a terminal)
- $S \rightarrow \varepsilon$  (where  $S$  is the start symbol).

In the next slides, we will give a 5-step algorithm that transforms every CFG into an equivalent one in Chomsky Normal Form:

1. START: Eliminate the start symbol from the RHS of all rules
2. TERM: Eliminate rules with terminals, except for rules  $A \rightarrow a$
3. BIN: Eliminate rules with more than two variables
4. DEL: Eliminate epsilon productions
5. UNIT: Eliminate unit rules

## Chomsky Normal Form: algorithm

1. Eliminate the start symbol from the RHS of all rules
  2. Eliminate rules with terminals, except for rules  $A \rightarrow a$
  3. Eliminate rules with more than two variables
  4. Eliminate epsilon productions
  5. Eliminate unit rules
- 

Add the new start symbol  $S_0$  and the rule  $S_0 \rightarrow S$

# Chomsky Normal Form: algorithm

1. Eliminate the start symbol from the RHS of all rules
  2. **Eliminate rules with terminals, except for rules  $A \rightarrow a$**
  3. Eliminate rules with more than two variables
  4. Eliminate epsilon productions
  5. Eliminate unit rules
- 
- Replace every terminal  $a$  on the RHS of a rule (that is not of the form  $A \rightarrow a$ ) by the new variable  $N_a$ .
  - For each such terminal  $a$  create the new rule  $N_a \rightarrow a$ .

# Chomsky Normal Form: algorithm

1. Eliminate the start symbol from the RHS of all rules
  2. Eliminate rules with terminals, except for rules  $A \rightarrow a$
  3. **Eliminate rules with more than two variables**
  4. Eliminate epsilon productions
  5. Eliminate unit rules
- 

For every rule of the form  $A \rightarrow X_1X_2\dots X_n$  (where  $n > 2$ ), delete it and create new variables  $A_1, A_2, \dots, A_{n-2}$  and rules:

$$A \rightarrow X_1A_1$$

$$A_1 \rightarrow X_2A_2$$

⋮

$$A_{n-3} \rightarrow X_{n-2}A_{n-2}$$

$$A_{n-2} \rightarrow X_{n-1}X_n$$

# Chomsky Normal Form: algorithm

1. Eliminate the start symbol from the RHS of all rules
  2. Eliminate rules with terminals, except for rules  $A \rightarrow a$
  3. Eliminate rules with more than two variables
  4. **Eliminate epsilon productions**
  5. Eliminate unit rules
- 

For every rule of the form  $U \rightarrow \varepsilon$  (except  $S_0 \rightarrow \varepsilon$ )

- Remove the rule.
- For each rule  $A \rightarrow \alpha$  containing  $U$ , add every possible rule  $A \rightarrow \alpha'$  where  $\alpha'$  is  $\alpha$  with one or more  $U$ 's removed; only add the rule  $A \rightarrow \varepsilon$  if this rule has not already been removed.

# Chomsky Normal Form: algorithm

1. Eliminate the start symbol from the RHS of all rules
  2. Eliminate rules with terminals, except for rules  $A \rightarrow a$
  3. Eliminate rules with more than two variables
  4. Eliminate epsilon productions
  5. **Eliminate unit rules**
- 

For each rule of the form  $A \rightarrow B$ :

- Remove the rule  $A \rightarrow B$
- For each rule of the form  $B \rightarrow \alpha$  add the new rule  $A \rightarrow \alpha$   
(unless it was previously removed)

# Chomsky Normal Form: example

$$\begin{aligned}S &\rightarrow ASA \mid aB \\A &\rightarrow B \mid S \\B &\rightarrow b \mid \varepsilon\end{aligned}$$

Step 1 (START): Eliminate start symbol from the RHS of all rules:

$$\begin{aligned}\mathbf{S_0 \rightarrow S} \\S &\rightarrow ASA \mid aB \\A &\rightarrow B \mid S \\B &\rightarrow b \mid \varepsilon\end{aligned}$$

# Chomsky Normal Form: example

$$\begin{aligned}S_0 &\rightarrow S \\S &\rightarrow ASA \mid aB \\A &\rightarrow B \mid S \\B &\rightarrow b \mid \varepsilon\end{aligned}$$

Step 2 (TERM): Eliminate rules with terminals, except for rules  
 $A \rightarrow a$ :

$$\begin{aligned}S_0 &\rightarrow S \\S &\rightarrow ASA \mid \mathbf{N_a}B \\A &\rightarrow B \mid S \\B &\rightarrow b \mid \varepsilon \\\mathbf{N_a} &\rightarrow a\end{aligned}$$

# Chomsky Normal Form: example

$$S_0 \rightarrow S$$

$$S \rightarrow ASA \mid N_a B$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \varepsilon$$

$$N_a \rightarrow a$$

Step 3 (BIN): Eliminate rules with more than two variables:

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_a B$$

$$S_1 \rightarrow SA$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \varepsilon$$

$$N_a \rightarrow a$$

# Chomsky Normal Form: example

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB$$

$$S_1 \rightarrow SA$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \varepsilon$$

$$N_a \rightarrow a$$

Step 4 (DEL): Eliminate epsilon production  $B \rightarrow \varepsilon$

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid \mathbf{N_a}$$

$$S_1 \rightarrow SA$$

$$A \rightarrow B \mid S \mid \varepsilon$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

# Chomsky Normal Form: example

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid N_a$$

$$S_1 \rightarrow SA$$

$$A \rightarrow B \mid S \mid \varepsilon$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

Step 4 (DEL): Eliminate epsilon production  $A \rightarrow \varepsilon$

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid N_a \mid \mathbf{S_1}$$

$$S_1 \rightarrow SA \mid \mathbf{S}$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

# Chomsky Normal Form: example

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid N_a \mid S_1$$

$$S_1 \rightarrow SA \mid S$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

Step 5 (UNIT): Eliminate unit rules  $S \rightarrow N_a, A \rightarrow B, S \rightarrow S_1$

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid \mathbf{a} \mid \mathbf{SA} \quad (\text{and useless } S \rightarrow S)$$

$$S_1 \rightarrow SA \mid S$$

$$A \rightarrow \mathbf{b} \mid S$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

# Chomsky Normal Form: example

$$S_0 \rightarrow S$$

$$S \rightarrow AS_1 \mid N_aB \mid a \mid SA$$

$$S_1 \rightarrow SA \mid S$$

$$A \rightarrow b \mid S$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

Step 5 (UNIT): Eliminate unit rules  $S_0 \rightarrow S$ ,  $S_1 \rightarrow S$ ,  $A \rightarrow S$

$$S_0 \rightarrow \mathbf{AS_1} \mid \mathbf{N_aB} \mid a \mid \mathbf{SA}$$

$$S \rightarrow AS_1 \mid N_aB \mid a \mid SA$$

$$S_1 \rightarrow \mathbf{AS_1} \mid \mathbf{N_aB} \mid a \mid SA$$

$$A \rightarrow b \mid \mathbf{AS_1} \mid \mathbf{N_aB} \mid a \mid \mathbf{SA}$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

# Chomsky Normal Form: example

All done!

$$S_0 \rightarrow AS_1 | N_aB | a | SA$$

$$S \rightarrow AS_1 | N_aB | a | SA$$

$$S_1 \rightarrow AS_1 | N_aB | a | SA$$

$$A \rightarrow b | AS_1 | N_aB | a | SA$$

$$B \rightarrow b$$

$$N_a \rightarrow a$$

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 9b: Membership problem for CFGs in CNF**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
**SYDNEY**



# Membership problem for CFG in CNF

**Input:** a CFG  $G$  that is in CNF, and string  $w$ .

**Output:** 1 if the string  $w$  is generated by  $G$ , and 0 otherwise.

## Table-filling algorithm (aka Dynamic Programming)

- This approach accumulates information about smaller subproblems to solve the larger problem (similar to divide and conquer)
- The table records the solution to the subproblems, so we only need to solve each subproblem once (aka memoisation)
- Main steps: define the subproblems, find the recursion, make sure you solve each subproblem once.
- You will see this again in COMP3027: Algorithm Design
- The algorithm we will see is known as the **CYK algorithm** (Cocke–Younger–Kasami).

# What are the subproblems?

## Idea

- A parse tree for a string  $w$  is built from a root, a left subtree, and a right subtree (remember that  $G$  is in CNF).
- The left (right) subtree is parse tree of a prefix (suffix) of  $w$ .
- So, the immediate subproblems for  $w$  are computing which variables generate prefixes/suffixes of  $w$ .
- So, the subproblems for  $w$  are computing which variables generate which substrings of  $w$ .

# What are the entries in the table?

Given  $G$  in CNF, and a non-empty string  $w = w_1w_2 \cdots w_n$ :

- Write  $\text{table}(i, j)$  for the set of variables  $A$  that generate the substring  $w_iw_{i+1} \dots w_j$ .
- The algorithm will compute  $\text{table}(i, j)$  for all  $1 \leq i < j \leq n$ .
- Once the table is computed, just check if  $S \in \text{table}(1, n)$ . If yes, then the grammar generates  $w$ , and if no, then it doesn't.

# Computing the table recursively

Compute  $\text{table}(i, j)$  using the following recursive procedure:

1. If  $i = j$  then  $\text{table}(i, j)$  is the set of variables  $A$  such that  $A \rightarrow w_i$  is a rule of the grammar.
2. If  $i < j$  then  $\text{table}(i, j)$  is the set of variables  $A$  for which there is a rule  $A \rightarrow BC$  and an integer  $k$  with  $i \leq k < j$  such that  $B \in \text{table}(i, k)$  and  $C \in \text{table}(k + 1, j)$ .

**Q: Why is the recursion correct?**

# Computing the table recursively

Compute  $\text{table}(i, j)$  using the following recursive procedure:

1. If  $i = j$  then  $\text{table}(i, j)$  is the set of variables  $A$  such that  $A \rightarrow w_i$  is a rule of the grammar.
2. If  $i < j$  then  $\text{table}(i, j)$  is the set of variables  $A$  for which there is a rule  $A \rightarrow BC$  and an integer  $k$  with  $i \leq k < j$  such that  $B \in \text{table}(i, k)$  and  $C \in \text{table}(k + 1, j)$ .

**Q: Why does this recursion stop?**

- At each step, we call the procedure on “smaller” problems.
- In what sense is  $\text{table}(i, k)$  and  $\text{table}(k + 1, j)$  smaller than  $\text{table}(i, j)$ ? The size of the intervals  $[i, k]$  and  $[k + 1, j]$  is smaller than the size of the interval  $[i, j]$ .

# We want to avoid computing table entries more than once.

- So, before making a recursive call just check if the value has already been computed. If yes, use that value and don't recurse. If not, recurse.

# Can we write this with a bunch of loops?

Yes, but it is harder to read. See Sipser (edition 3) Theorem 7.16.

$D$  = “On input  $w = w_1 \cdots w_n$ :

1. For  $w = \epsilon$ , if  $S \rightarrow \epsilon$  is a rule, *accept*; else, *reject*.  $\llbracket w = \epsilon \text{ case} \rrbracket$
2. For  $i = 1$  to  $n$ :  $\llbracket$  examine each substring of length 1  $\rrbracket$
3. For each variable  $A$ :
4. Test whether  $A \rightarrow b$  is a rule, where  $b = w_i$ .
5. If so, place  $A$  in *table*( $i, i$ ).
6. For  $l = 2$  to  $n$ :  $\llbracket l \text{ is the length of the substring} \rrbracket$
7. For  $i = 1$  to  $n - l + 1$ :  $\llbracket i \text{ is the start position of the substring} \rrbracket$
8. Let  $j = i + l - 1$ .  $\llbracket j \text{ is the end position of the substring} \rrbracket$
9. For  $k = i$  to  $j - 1$ :  $\llbracket k \text{ is the split position} \rrbracket$
10. For each rule  $A \rightarrow BC$ :
11. If *table*( $i, k$ ) contains  $B$  and *table*( $k + 1, j$ ) contains  $C$ , put  $A$  in *table*( $i, j$ ).
12. If  $S$  is in *table*( $1, n$ ), *accept*; else, *reject*.”

(pseudocode from “Introduction to the theory of computation” by Michael Sipser)

# Can we fill the table by hand?

Yes, but this is best done by a computer!

(1,7)							
(1,6)	(2,7)						
(1,5)	(2,6)	(3,7)					
(1,4)	(2,5)	(3,6)	(4,7)				
(1,3)	(2,4)	(3,5)	(4,6)	(5,7)			
(1,2)	(2,3)	(3,4)	(4,5)	(5,6)	(6,7)		
(1,1)	(2,2)	(3,3)	(4,4)	(5,5)	(6,6)	(7,7)	

$w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5 \quad w_6 \quad w_7 \quad w_8$

Q: Where do I put the entry  $\text{table}(i, j)$ ?

- horizontal co-ordinate = **starting** position of substring =  $i$
- vertical co-ordinate = **length** of substring =  $j - i + 1$

Q: What entries are needed to compute  $\text{table}(i, j)$ ?

- You have to look at the pairs  $\text{table}(i, k), \text{table}(k + 1, j)$  for  $k = i, \dots, j - 1$ ; it's the **right-angled triangle below**  $\text{table}(i, j)$ .

Q: In what **order** are the entries computed?

- Row by row, bottom to top, left to right

# Example

S							
	VP						
S							
	VP						
S	NP PP						
NP	VP, V	Det	N	P	Det	N	
she	eats	a	fish	with	a	fork	
1	2	3	4	5	6	7	

Q: In what order are the entries computed?

- To compute an entry, you need the entries in the “right-angled triangle below it”.
- Row by row, bottom to top, left to right

# Example

S	VP						$S \rightarrow NP \ VP$
S							$VP \rightarrow VP \ PP \mid V \ NP$
S	VP						$PP \rightarrow P \ NP$
S							$NP \rightarrow Det \ N$
NP	VP, V	Det	N	P	Det	N	
she	eats	a	fish	with	a	fork	
1	2	3	4	5	6	7	

$VP \in \text{table}(2, 4)$  because

- the string from position 2 to 4 is "eats a fish",
- which can be split into "eats" from position 2 to 2,
- and "a fish" from position 3 to 4, and
- and  $VP \rightarrow V \ NP$  is a rule,  $V \in \text{table}(2, 2)$ , and  $NP \in \text{table}(3, 4)$ .

# Example

S	VP						$S \rightarrow NP \ VP$
S							$VP \rightarrow VP \ PP \mid V \ NP$
S	VP						$PP \rightarrow P \ NP$
S							$NP \rightarrow Det \ N$
NP	VP, V	Det	N	P	Det	N	
she	eats	a	fish	with	a	fork	
1	2	3	4	5	6	7	

$VP \in table(2, 7)$  because

- the string from position 2 to 7 is "eats a fish with a fork",
- which can be split into "eats a fish" from position 2 to 4,
- and "with a fork" from position 5 to 7, and
- and  $VP \rightarrow VP \ PP$  is a rule,  $VP \in table(2, 4)$ , and  $PP \in table(5, 7)$ .

# How efficient is this algorithm?

## Time complexity

- There are  $O(n^2)$  entries in the table,
- and each entry requires  $O(n|G|)$  work to compute, since one must check each rule and check at most  $n$  splits,
- So the total time is  $O(n^3|G|)$ .
- Here  $|G|$ , the size of  $G$ , is the number of bits required to write the grammar down, which is polynomial in the number of rules, variables and terminals.

## Asides

- For fixed  $G$  and varying  $w$ , the time is  $O(n^3)$ .
- If the input is large (e.g., a compiling a very large program), then this complexity is too high. So, in this case, one uses restricted grammars for which there are faster algorithms, see COMP3109: Programming Languages and Paradigms

# What if I want to compute a derivation?

You can adjust the algorithm to store more information in order to produce a derivation (or parse tree).

- Store in  $\text{table}(i, j)$  a rule  $A \rightarrow BC$  and splitting point  $k$  ( $i \leq k < j$ ) that can be used to derive  $A \Rightarrow^* w_i w_{i+1} \dots w_j$ .
- You can then deduce a rightmost derivation using a stack.
- Start by pushing the element  $(S, 1, n)$  onto the stack, and then repeat the following:
  - if  $(A, i, i)$  is the top element of the stack then apply the rule  $A \rightarrow w_i$  and pop the stack.
  - if  $(A, i, j)$  is the top-element of the stack, then apply the rule  $A \rightarrow BC$ , pop the stack, and push the element  $(B, i, k)$  followed by  $(C, k + 1, j)$  onto the stack.

## Good to know

- There is a machine-theoretic characterisation of context-free languages (pushdown automaton = NFA + stack).
  - Come to COMP2922 to learn more! or see Sipser Chapter 2.2
- Not every language is context-free. E.g.,  $\{ww : w \in \{0, 1\}^*\}$  is not context-free.
  - The proof of this uses a pumping argument, see Sipser Chapter 2.3.

# Where are we going?

Next week we start learning about an even more powerful model of computation that can even recognise non-context-free languages — the Turing machine (= automaton + unbounded memory).

This is the most powerful model of computation that we know of, and is a model of a general purpose computer.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 10a: Turing Machines**

### **Sipser Chapter 3**

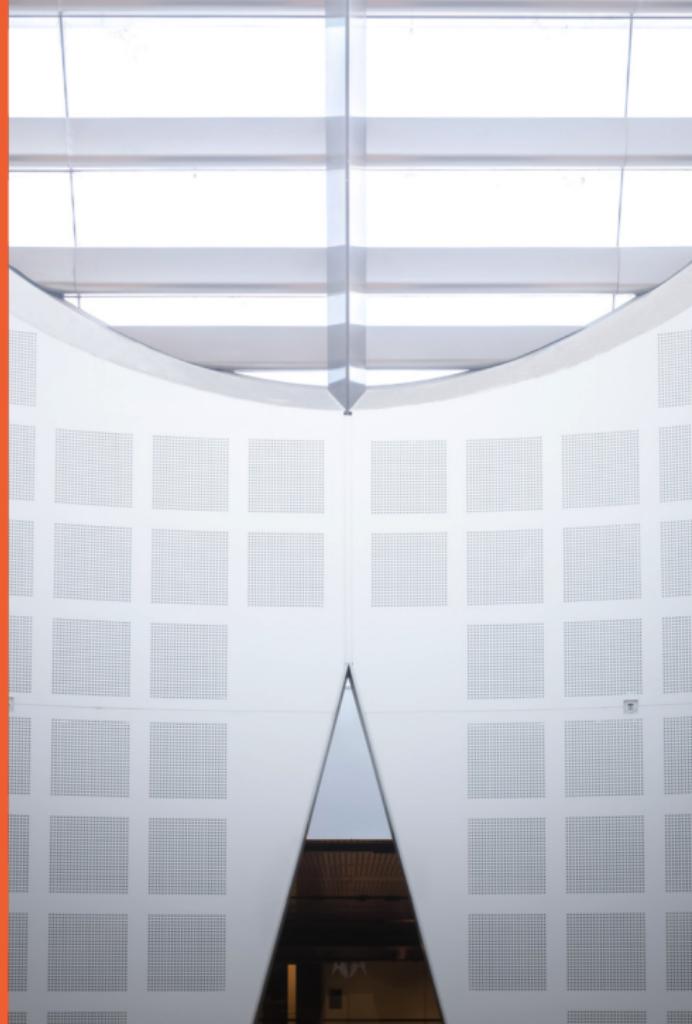
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# Course topics

Here are some important mathematical models of computation:

1. Propositional Logic, Predicate Logic
2. Finite Automata, Regular Expressions, Regular Grammars
3. Pushdown Automata, Context-free grammars
4. **Turing Machines, Lambda Calculus**

# Motivation

What is an algorithm? Is every decision problem solved by some algorithm? some program?

Before a model of computation was invented for this, an algorithm was described as a “process with a finite number of operations”.

## Example 1a

Is there an algorithm for finding real-number solutions to polynomial equations?

- There is a simple algorithm for solving quadratic equations

$$ax^2 + bx + c = 0$$

- You can use the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- These give **real-number** (even complex-number) solutions.
- Is there an algorithm for solving **polynomial** equations?

$$ax^3 + bx^2 + cx + d = 0, \text{etc ?}$$

- The answer is **yes** — Tarski (21st century Mathematician) found an algorithm in the 1940s.

## Example 1b

David Hilbert (20th century mathematician) asked in 1900 if there is an algorithm for finding **integer-number** solutions to polynomial equations.

- Matiyasevich-Robinson-Davis-Putnam (1970) proved the answer is **no!**
- To prove this, even to state this question precisely, they needed a **model** of algorithms.

### Example 3

Programs may run into infinite loops. Is there an algorithm that tells you if a given Python program  $p$  run on input  $x$  results in an infinite loop or not?

- You could run the program  $p$  on input  $x$ ... But, how long should you wait for it to stop?
- General principle: any problem about general computation cannot be solved by programs.
- To prove this, even to state this precisely, we need a **model** of algorithms.

# Models of algorithms

Various formalisms have been proposed:

- Turing machines (Alan Turing 1936)
- Post systems (Emil Post)
- $\mu$ -recursive functions (Kurt Gödel, Jacques Herbrand)
- $\lambda$ -calculus (Alonzo Church, Stephen C. Kleene)
- Combinatory logic (Moses Schönfinkel and Haskell B. Curry)

# Church-Turing thesis

Surprisingly,

## Theorem

*All these formalisms are equivalent: a problem is decidable by one approach if and only if it is decidable by any other.*

This observation strongly supports the idea that there is only one form of general computation:

## Church-Turing Thesis

Turing Machines define computability.

# Turing Machines = “DFA + Read/Write Tape”

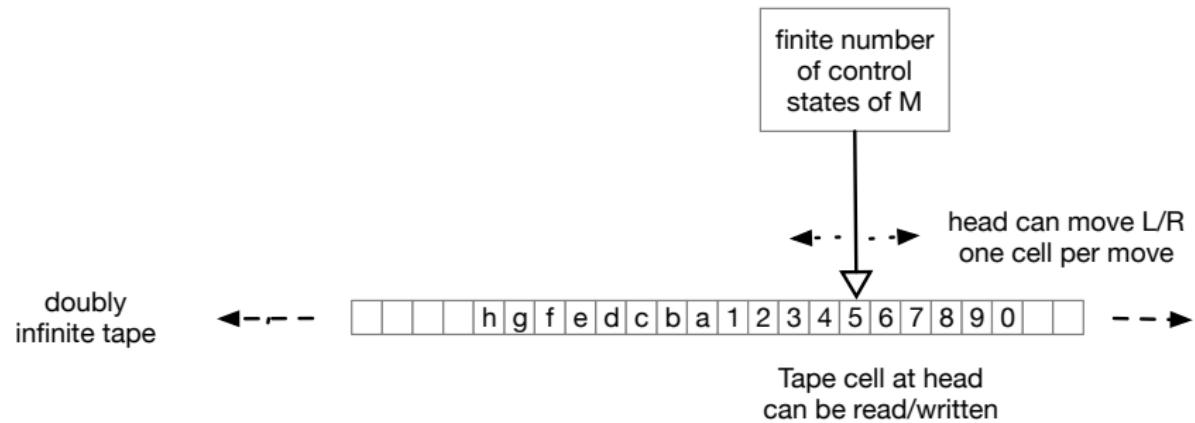
Finite control with a *tape* which is used for input and for unbounded storage. There is a “head” that can read/write the tape.

- Tape is infinite in both directions.
- There is a special “blank” symbol ( $\sqcup$ ).
- All but a finite number of positions are blank at any given time.

## Definition

A *move* of a TM is based on the current state and the symbol currently being scanned, and it changes state, writes a new symbol, and moves left or right.

# Turing Machines: schematic view



# Turing Machines

## Definition

A TM is a tuple  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$  with components:

1.  $Q$  is a finite set of **states**
2.  $\Sigma$  is the **input alphabet**, not containing the *blank symbol*  $\sqcup$
3.  $\Gamma$  is the **tape alphabet**, containing  $\sqcup$  and all of  $\Sigma$
4.  $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{l}, \text{r}\}$  is the **transition function**
5.  $q_0$  is the **start state**
6.  $q_{\text{accept}}$  is the **accept state**
7.  $q_{\text{reject}}$  is the **reject state** ( $\neq q_{\text{accept}}$ )

# TM configurations

A **configuration** for a TM is a triple  $(u, q, v) \in \Gamma^* \times Q \times \Gamma^*$  typically written  $uqv$ .

It represents the situation in which

- $q$  is the current state,
- the tape content is  $uv$  (the infinite string of blanks to the left and right of  $uv$  is suppressed),
- the head is at the first symbol of  $v$ .

# TM configurations

For  $w \in \Sigma^*$ :

- configurations of the form  $q_0w$  are called **start configurations**.
- Configurations of the form  $uq_{\text{accept}}v$  are called **accepting configurations**.
- Configurations of the form  $uq_{\text{reject}}v$  are called **rejecting configurations**.

# TM configurations

## Definition

For  $a, b, c \in \Gamma$   $u, v \in \Gamma^*$ , and  $q, q' \in Q$ :

- Configuration  $uaqbv$  **yields**  $uq'acv$  if  $\delta(q, b) = (q', c, l)$ .
- Configuration  $uaqbv$  **yields**  $uacq'v$  if  $\delta(q, b) = (q', c, r)$ .

## Definition

A TM  $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$  **accepts** input  $w \in \Sigma^*$  if a sequence  $C_1, \dots, C_n$  of configurations exists, where

1.  $C_1 = q_0w$ ,
2. each  $C_i$  yields  $C_{i+1}$ , and
3.  $C_n$  is an accepting configuration.

The **language**  $L(M)$  **recognised** by  $M$  is the set of strings accepted by  $M$ .

## Example

A Turing Machine that recognises the language  
 $\{ww : w \in \{0, 1\}^*\}$ .

Informal description:

- 1 Place endmarkers  $\vdash$ ,  $\dashv$  at either end of the input
- 2 Check the input is of even length – reject if it is not
- 3 Take the leftmost symbol that is a 0 or 1: Replace 0 with  $a$  and 1 with  $b$
- 4 Take the rightmost symbol that is a 0 or 1: Replace 0 with  $A$  and 1 with  $B$
- 5 Repeat steps 3 and 4 until there are no more 0's or 1's
- 6 Take the leftmost symbol that is an  $a$  or  $b$ : Erase that symbol, but use the state to remember it
- 7 Take the leftmost symbol that is an  $A$  or  $B$ : Reject if the symbol does not match the state, otherwise erase the symbol
- 8 Repeat steps 6 and 7 until there are no more symbols
- 9 Accept the input

## Example

A Turing Machine that recognises the language  
 $\{ww : w \in \{0, 1\}^*\}$ .

### Formal description

$(\{q_0, \dots, q_{12}, q_{\text{accept}}, q_{\text{reject}}\}, \{0, 1\}, \{\sqcup, 0, 1, a, b, A, B, \vdash, \dashv\}, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

where  $\delta$  is defined as:

	$\sqcup$	0	1	a	b	A	B	$\vdash$	$\dashv$
$q_0$	$q_{\text{accept}}$	$(q_1, 0, L)$	$(q_1, 1, L)$						
$q_1$		$(q_2, \vdash, R)$							
$q_2$	$(q_4, \dashv, L)$	$(q_3, 0, R)$	$(q_3, 1, R)$						
$q_3$	$q_{\text{reject}}$	$(q_2, 0, R)$	$(q_2, 1, R)$						
$q_4$		$(q_5, A, L)$	$(q_5, B, L)$	$(q_8, a, L)$	$(q_8, b, L)$				
$q_5$		$(q_5, 0, L)$	$(q_5, 1, L)$	$(q_6, a, R)$	$(q_6, b, R)$			$(q_6, \vdash, R)$	
$q_6$		$(q_7, a, R)$	$(q_7, b, R)$						
$q_7$		$(q_7, 0, R)$	$(q_7, 1, R)$			$(q_4, A, L)$	$(q_4, B, L)$		
$q_8$	$(q_8, \sqcup, L)$			$(q_8, a, L)$	$(q_8, b, L)$	$(q_8, A, L)$	$(q_8, B, L)$	$(q_9, \vdash, R)$	
$q_9$	$(q_9, \sqcup, R)$			$(q_{10}, \sqcup, R)$	$(q_{11}, \sqcup, R)$				$q_{\text{accept}}$
$q_{10}$	$(q_{10}, \sqcup, R)$			$(q_{10}, a, R)$	$(q_{10}, b, R)$	$(q_8, \sqcup, L)$			$q_{\text{reject}}$
$q_{11}$	$(q_{11}, \sqcup, R)$			$(q_{11}, a, R)$	$(q_{11}, b, R)$	$q_{\text{reject}}$	$(q_8, \sqcup, L)$		$q_{\text{reject}}$

## Example

A Turing Machine that recognises the language  
 $\{ww : w \in \{0, 1\}^*\}$ .

	$q_0 0101$	$\rightarrow$	$q_1 \sqcup 0101$	$\rightarrow$	$\vdash q_2 0101$
$\rightarrow$	$\vdash q_3 101$	$\rightarrow$	$\vdash 01q_2 01$	$\rightarrow$	$\vdash 010q_3 1$
$\rightarrow$	$\vdash 0101q_2 \sqcup$	$\rightarrow$	$\vdash 010q_4 1 \dashv$	$\rightarrow$	$\vdash 01q_5 0B \dashv$
$\rightarrow$	$\vdash 0q_5 10B \dashv$	$\rightarrow$	$\vdash q_5 010B \dashv$	$\rightarrow$	$q_5 \vdash 010B \dashv$
$\rightarrow$	$\vdash q_6 010B \dashv$	$\rightarrow$	$\vdash aq_7 10B \dashv$	$\rightarrow$	$\vdash a1q_7 0B \dashv$
$\rightarrow$	$\vdash a10q_7 B \dashv$	$\rightarrow$	$\vdash a1q_4 0B \dashv$	$\rightarrow$	$\vdash aq_5 1AB \dashv$
$\rightarrow$	$\vdash q_5 a1AB \dashv$	$\rightarrow$	$\vdash aq_6 1AB \dashv$	$\rightarrow$	$\vdash abq_7 AB \dashv$
$\rightarrow$	$\vdash aq_4 bAB \dashv$	$\rightarrow$	$\vdash q_8 abAB \dashv$	$\rightarrow$	$q_8 \vdash abAB \dashv$
$\rightarrow$	$\vdash q_9 abAB \dashv$	$\rightarrow$	$\vdash \sqcup q_{10} bAB$	$\rightarrow$	$\vdash \sqcup bq_{10} AB \dashv$
$\rightarrow$	$\vdash \sqcup q_8 b \sqcup B \dashv$	$\rightarrow$	$\vdash q_8 \sqcup b \sqcup B \dashv$	$\rightarrow$	$q_8 \vdash \sqcup b \sqcup B \dashv$
$\rightarrow$	$\vdash q_9 \sqcup b \sqcup B \dashv$	$\rightarrow$	$\vdash \sqcup q_9 b \sqcup B \dashv$	$\rightarrow$	$\vdash \sqcup \sqcup q_{11} \sqcup B \dashv$
$\rightarrow$	$\vdash \sqcup \sqcup q_{11} B \dashv$	$\rightarrow$	$\vdash \sqcup \sqcup q_8 \sqcup \sqcup \dashv$	$\rightarrow$	$\vdash \sqcup q_8 \sqcup \sqcup \sqcup \dashv$
$\rightarrow$	$\vdash q_8 \sqcup \sqcup \sqcup \dashv$	$\rightarrow$	$q_8 \vdash \sqcup \sqcup \sqcup \dashv$	$\rightarrow$	$\vdash \sqcup q_9 \sqcup \sqcup \sqcup \dashv$
$\rightarrow$	$\vdash \sqcup \sqcup q_9 \sqcup \sqcup \dashv$	$\rightarrow$	$\vdash \sqcup \sqcup \sqcup q_9 \sqcup \dashv$	$\rightarrow$	$\vdash \sqcup \sqcup \sqcup \sqcup q_9 \dashv \rightarrow q_{\text{accept}}$

## Definition

$L \in \Sigma^*$  is **Turing-recognisable**<sup>1</sup> if some TM  $M$  recognises it, i.e.  
 $L = L(M)$ .

A TM can fail to accept some input either by:

- *rejecting*: eventually entering  $q_{\text{reject}}$ , or
- *diverging*: entering an infinite loop.

TMs that do not diverge on any input are called **deciders**.

## Definition

$L \in \Sigma^*$  is **Turing-decidable**<sup>2</sup> if some TM decides it, i.e., for every  $w \in \Sigma^*$ :

- if  $w \in L$  then the TM accepts  $w$ , and
- if  $w \notin L$  then the TM rejects  $w$  (i.e., it enters a rejecting state).

Note.  $L$  decidable implies  $L$  recognisable.

---

<sup>1</sup>aka *recursively enumerable*, aka *computably enumerable*, aka *recognisable*

<sup>2</sup>aka *recursive*, aka *computable*, aka *decidable*

Turing machines are fairly robust, i.e., variations and extensions of the model do not change the languages that can be recognised.

Two machines are **equivalent** if they recognise the same language.

# Stay put TMs

So far, our TMs had to move left or right in each step. Suppose we add a 3rd option—to stay put, so the type of a TM's transition function becomes:

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{l, r, s\}$$

## Theorem

*Every stay put TM has an equivalent ordinary TM.*

## Proof Idea.

Replace each s-transition with two transitions, an r-transition followed by an l-transition. □

# Left-bounded TM

- The head starts on the left-most square of the tape with the head on the first letter of the input.
- Should the transition function suggest to move left when the head is already at the left-most position, the head stays put.

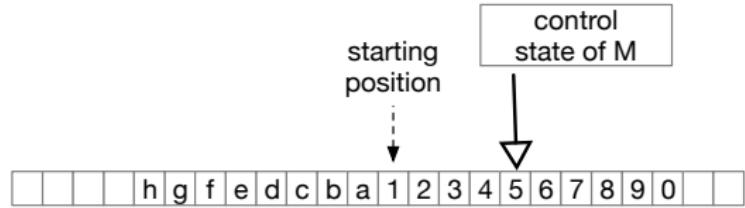
## Theorem

*Every doubly infinite tape TM has an equivalent left-bounded TM.*

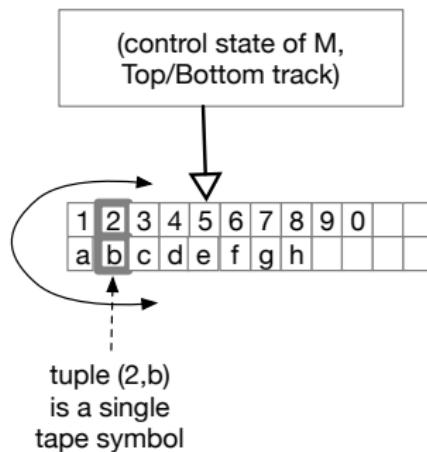
## Proof idea.

To simulate a doubly infinite tape TM by a left-bounded one, we split the left-bounded tape into an upper and a lower half. The upper half represents the right half of the tape (from the initial head position onwards) and the lower half represents the left half. An extra state component keeps track of the half in which the head currently is. □

doubly  
infinite tape



simulation using  
left-bounded  
infinite tape



# Multitape TMs

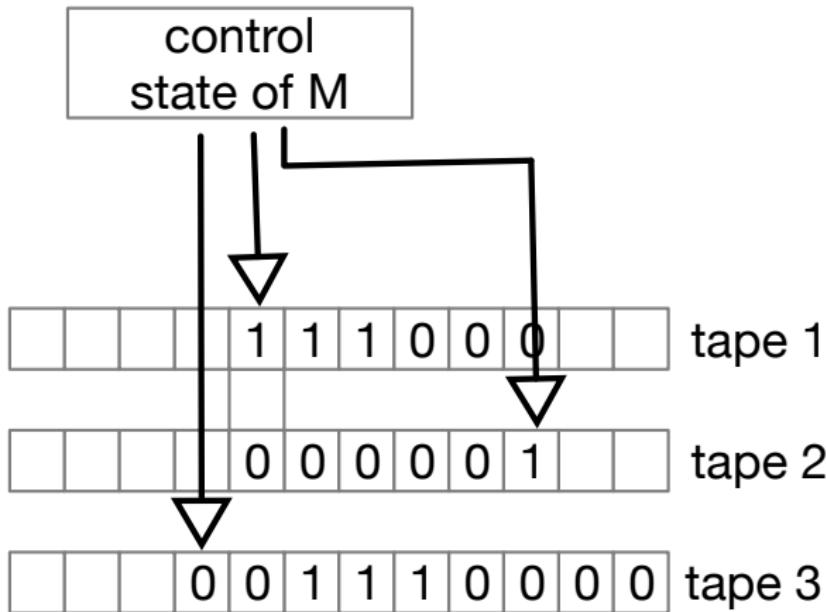
- A *multitape TM* has multiple tapes, each with its own head for reading and writing.
- Initially the input appears on tape 1, and the others start out blank.
- The type of the transition function of a  $k$ -tape TM becomes:

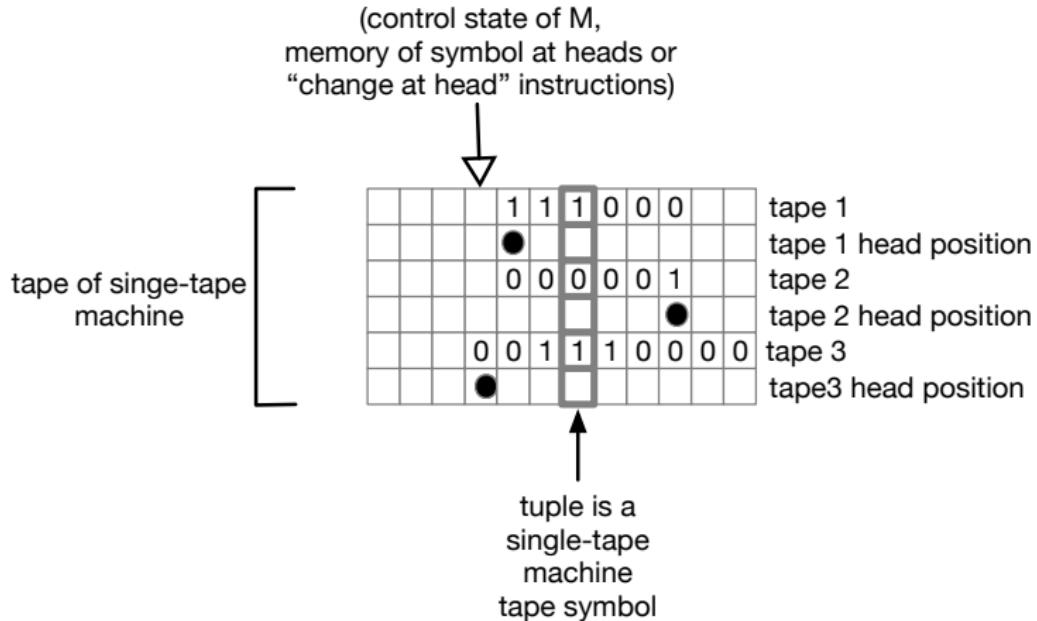
$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{l, r, s\}^k$$

## Theorem

*Every multitape TM has an equivalent single-tape TM.*

Idea: split the tape into  $2k$ -many "tracks", with the  $2i$ th track corresponding to the  $i$ th tape and the  $2i + 1$ th track storing the position of the  $i$ th head. So, the input alphabet contains symbols that correspond to a "slice" of all the tapes (including their heads).





# Non-Deterministic TMs

The type of the transition function of a non-deterministic TM becomes:

$$\delta : Q \times \Gamma \longrightarrow 2^{Q \times \Gamma \times \{l,r,s\}}$$

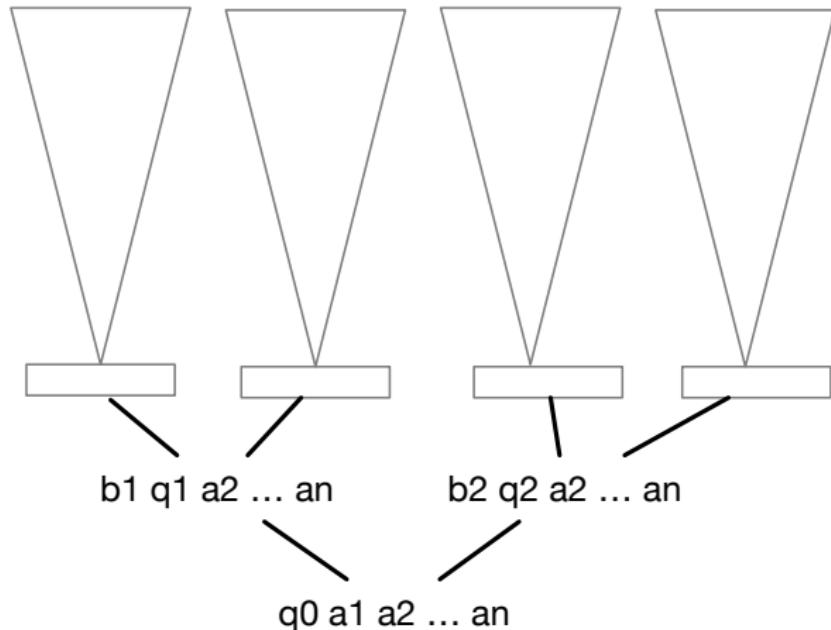
## Theorem

*Every non-deterministic TM has an equivalent deterministic TM.*  
[A variant of this theorem holds for deciders.]

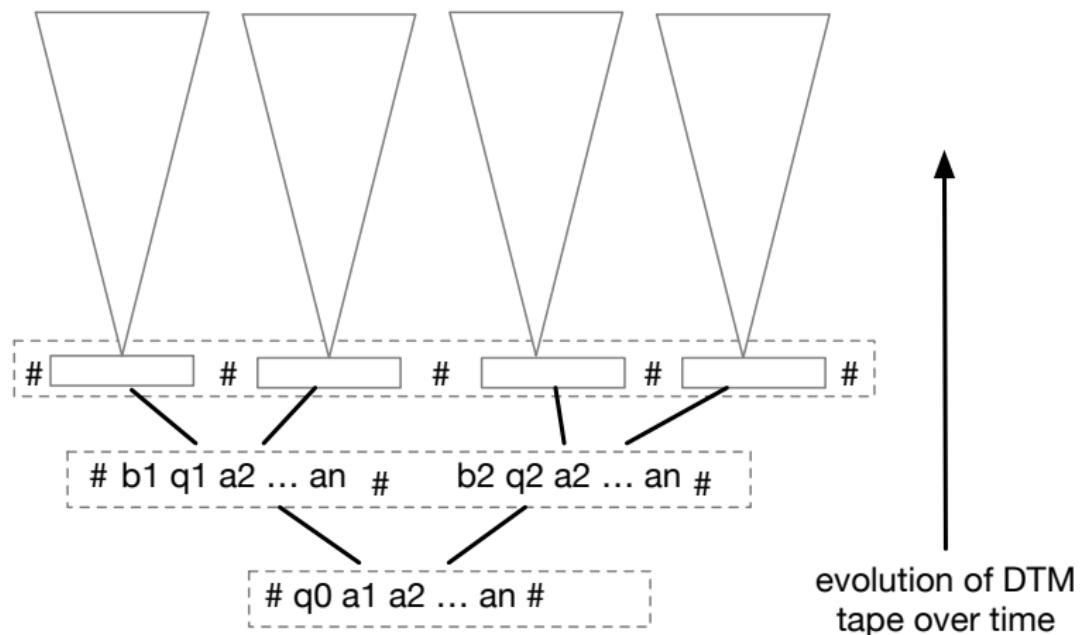
## Proof idea.

We simulate any non-deterministic TM  $N$  with a deterministic TM  $D$  that breadth-first searches for  $q_{\text{accept}}$  on all possible branches of  $N$ 's non-deterministic computation. If  $D$  ever finds  $q_{\text{accept}}$  on one of these branches, it accepts. Otherwise it will diverge.  $\square$

# A computation tree of a non-deterministic machine



# Deterministic simulation of the computation tree of a non-deterministic machine



# **COMP2022|2922**

## **Models of Computation**

### **Lesson 10b: Decidability**

### **Sipser Chapter 4**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# Encodings

- The input to a Turing Machine is always a string, but it can be useful to work with objects other than strings over  $\Sigma$ .
  - e.g., the language of all binary strings that encode integers  $x, y, z$  such that  $z = x + y$ .
- We can encode almost anything as a string (integers, sets of integers, graphs, programs, Turing machines!)
  - The exact encoding is unimportant (for us).
- If  $O$  is a (finitely-presented) object, an **encoding** of  $O$  over  $\Sigma$ , written  $\langle O \rangle$  is a representation of  $O$  as a string.
- For instance, an encoding of a Boolean formula  $F$  and an assignment  $\alpha$  is written  $\langle F, \alpha \rangle$ .
  - So, strictly speaking, we don't feed  $F, \alpha$  into a TM, but rather the string  $\langle F, \alpha \rangle$ .

# Encodings: Example

We can encode sets of strings

- One standard approach for encoding a **set** of binary strings is by listing them in some order, separated by a new alphabet symbol.
- So the set  $\{00, 1, 000, 10\} \subseteq \{0, 1\}^*$  is encoded by the string

$\#00\#\#000\#10\#$

as well as by the string

$\#1\#\#00\#\#10\#\#000\#\#$

## Encodings: Example

We can encode Turing Machines as strings over  $\{0, 1\}$ .

E.g. if  $\langle M \rangle$  begins with the prefix

$$0^n 1 0^m 1 0^k 1 0^s 1 0^t 1 0^r 1 0^u$$

this might indicate that  $M$

- has  $n$  states  $(0, 1, \dots, n - 1)$ ,
- has  $m$  tape symbols  $(0, 1, \dots, m - 1)$ , of which the first  $k$  are input symbols;
- the start, accept and reject states are  $s, t$  and  $r$  respectively, and
- the blank symbol is  $u$ .

The remainder can specify the transitions:

- $0^p 1 0^a 1 0^q 1 0^b 1 0$  might indicate that  $\delta$  contains the transition

$$((p, a), (q, b, L)).$$

# Decidable problems

- Recall that in previous lectures we mentioned various decision problems (about logics, automata, grammars).
- These can now be formalised.

## Example

The *membership problem* for DFAs<sup>3</sup> is the language

$$A_{\text{DFA}} = \{ \langle D, w \rangle \mid D \text{ is a DFA that accepts } w \}$$

---

<sup>3</sup>Aka *acceptance problem* for DFAs

# Decidable problems

$$A_{\text{DFA}} = \{ \langle D, w \rangle \mid D \text{ is a DFA that accepts } w \}$$

## Theorem

$A_{\text{DFA}}$  is decidable.

## Proof idea.

A TM that decides  $A_{\text{DFA}}$  could simulate  $D$  on  $w$  by keeping track of  $D$ 's current state and of how much of the input word  $w$  has been read by writing these data items onto the tape. □

# Decidable problems

$A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts } w \}$

$A_{\text{RE}_{\Sigma}} = \{ \langle R, w \rangle \mid R \text{ is a RE and } w \in L(R) \}$

$E_{\text{DFA}} = \{ \langle B \rangle \mid B \text{ is a DFA and } L(B) = \emptyset \}$

$EQ_{\text{DFA}} = \{ \langle A, B \rangle \mid A, B \text{ are DFAs and } L(A) = L(B) \}$

# Decidable Problems for CFLs

## Theorem

$$A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$$

*is decidable.*

## Proof idea.

Convert  $G$  to CNF and apply the CYK algorithm. □

# Decidable? Recognisable?

The *acceptance problem* for TMs is the language

$$A_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w \}$$

This language is recognisable.

- Construct a TM  $U$ , which, on input  $\langle M, w \rangle$  simulates  $M$  on  $w$  and accepts if  $M$  enters  $q_{\text{accept}}$  and rejects if  $M$  enters  $q_{\text{reject}}$ .
- $U$  recognises  $A_{\text{TM}}$  but it doesn't decide it. Why?

# Not all languages are decidable

## Theorem

$A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM that accepts } w \}$   
*is not decidable.*

## Proof idea to show $A_{\text{TM}}$ is not decidable.

Assume to the contrary that  $H$  is a decider for  $A_{\text{TM}}$ . Let  $D$  be a TM that uses  $H$  as a building block. Its inputs are TM descriptions.

$D$  does the following on input  $\langle M \rangle$ :

1. write  $\langle M, \langle M \rangle \rangle$  on the tape,
2. run  $H$  which will accept or reject,
3. output the opposite of  $H$ 's result.

So  $D$  is a TM, and so it has an encoding  $\langle D \rangle$ .

Crazy question: what happens when we run  $D$  on  $\langle D \rangle$ ?

$$D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle \end{cases}$$

which is a contradiction. Conclude that there is no decider  $H$  for  $A_{\text{TM}}$ .



# What have we learned

- Turing-machines are a robust model of algorithms/computation.
- Not every language can be decided by a TM.
  - The proof is subtle, and uses **self-reference**.
  - The same proof can be applied to any programming language to conclude that, there is a decision problem that can't be solved by any program in that language.
- There is lots more to learn...
  - There is a most general TM, i.e., a Universal TM.
  - Not every language can be recognised by a TM.
  - For instance
    - $\overline{A_{\text{TM}}} = \{ \langle M, w \rangle \mid M \text{ is a TM that does not accept } w \} \text{ is not recognisable!}$
  - In fact, there are even degrees of **uncomputability**.

# Other useful material

1. Closure properties, i.e., how to build new decidable/recognisable languages from old ones.
2. Universal TM

# Closure properties of Decidable/Recognisable Languages

## Theorem

*The decidable/recognisable languages are closed under union.*

## Proof.

Let  $L_1$  and  $L_2$  be languages that are decided (accepted) by TMs  $M_1$  and  $M_2$  respectively. We construct a TM that decides (accepts)  $L_2 \cup L_2$  as follows.

On input  $x$ :

1. Run  $M_1$  and  $M_2$  in parallel on  $x$ .
2. Accept if either  $M_1$  or  $M_2$  accepts.



# Closure properties of Decidable/Recognisable Languages

## Theorem

*The decidable languages are closed under complement.*

## Proof.

- Suppose  $L$  is decided by  $M$ .
- Define  $M'$  to be the same as  $M$  with  $q_{\text{accept}}$  and  $q_{\text{reject}}$  swapped.
- $M'$  decides  $\Sigma^* \setminus L$ .



## Corollary

*The decidable languages are closed under intersection.*

**Question.** What about recognisable languages?

## Theorem

*A language is decidable iff it and its complement are recognisable.*

### Proof idea.

$\Rightarrow$ : follows from the closure properties of decidable languages.

$\Leftarrow$ : We construct a decider  $D$  for  $L$  from a TM  $M$  that accepts  $L$  and a TM  $M'$  that accepts  $\overline{L}$  as follows.

On input  $x$ :

1. Run  $M$  and  $M'$  on  $x$  in parallel.
2. If  $M$  accepts, then accept.
3. If  $M'$  accepts, then reject.

# The Universal Turing Machine

- A **universal** TM is a TM  $U$  that can simulate the actions of any Turing machine.
- I.e. for any TM  $M$  and any input  $x$  of  $M$ ,  $U$  accepts (respectively rejects or loops on)  $\langle M, x \rangle$ , if and only if,  $M$  accepts (respectively rejects or loops on)  $x$ .
- This is similar to an interpreter of C written in C.

Note this is different (quantifier order) from results such as every NDTM can be simulated by a deterministic TM.

# Universal Turing Machine

## How to build a Universal TM $U$ ?

$U$  will have three tapes:

1. First tape holds the transition function  $\delta_M$  of the input TM  $M$ .
2. The second tape is used to hold the simulated contents of  $M$ 's tape.
3. The third tape holds the current state of  $M$ , and the current position of  $M$ 's tape head.
  - $U$  simulates  $M$  on input  $x$  one step at a time.
  - In each step, it updates  $M$ 's state and simulated tape contents and head position as dictated by  $\delta_M$ . If ever  $M$  halts and accepts or rejects, then  $U$  does the same.

# **COMP2022|2922**

## **Models of Computation**

### **Lesson 11: Introduction to the Lambda Calculus**

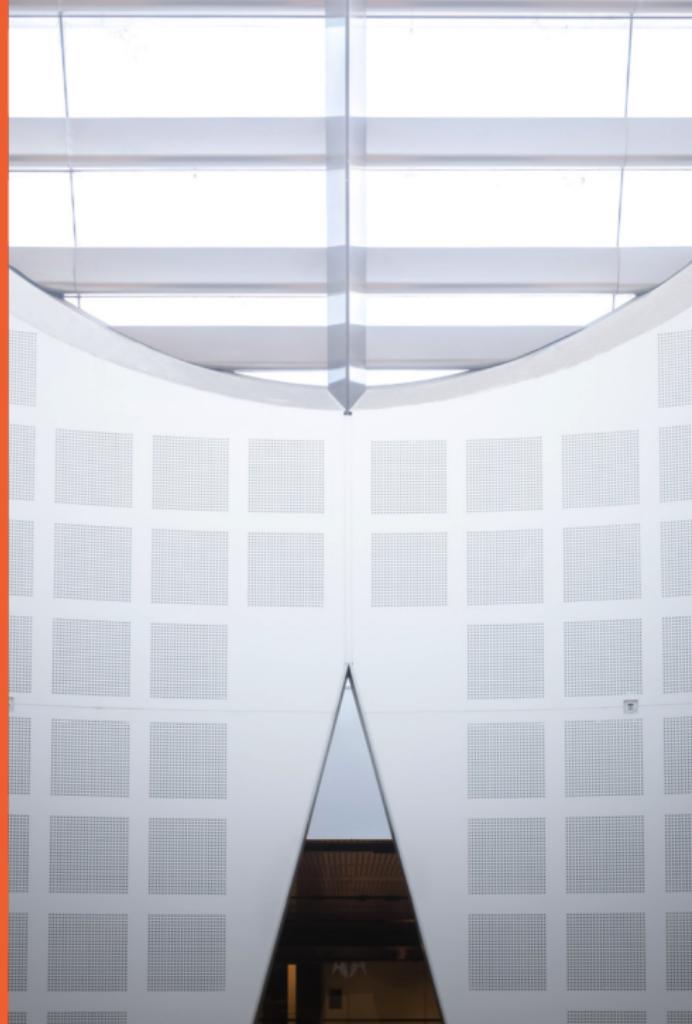
**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



# History

## Leibniz's dream

1. Create a universal language in which all possible problems can be stated.
  - For mathematical problems, use predicate logic.
2. Find a decision method to solve all the problems stated in the universal language.
  - In 1936, Church and Turing proved there was no such decision method. To do this:
  - Turing (1936/7) invented Turing machines, which lead to random access register machines, and imperative programming languages (Fortran, Pascal, C, Java).
  - Church (1936) invented the Lambda-calculus, which lead to functional programming languages (OCaml, Haskell)

# Computing with functions

In mathematics, the lambda-notation is used to represent functions:

- Expression  $\lambda x.f(x)$  **abstracts** the expression  $f(x)$  to get the function that on input  $x$  outputs  $f(x)$ .
- We can **rename** bound variables without changing the meaning:
  - $\lambda y.f(y)$  means the same thing as  $\lambda x.f(x)$ .
- Expression  $(\lambda x.f(x))7$  **applies** this function to the input 7.
- **Substituting** 7 for  $x$  gives the new expression  $f(7)$ .

# Lambda-calculus in a nutshell

- The  $\lambda$ -calculus consists of objects called  $\lambda$ -terms and rules for manipulating them.
- It was originally designed to formalise the notion of functional abstraction and functional application.
- Renaming is called  $\alpha$ -conversion, and substitution is called  $\beta$ -reduction.
  - Computation is done by applying reductions, i.e., rewriting the expression!
- Turing (1936/37) showed the  $\lambda$ -calculus and Turing-machines have the same expressive power.
- Terms represent both functions and data
  - Numbers (like 7), and functions like  $+1$  can be encoded.

# Lambda-calculus

## Definition

The  $\lambda$ -terms are defined by the following recursive process:

1. every variable  $x, y, z, \dots$  is a  $\lambda$ -term
2. if  $A, B$  are  $\lambda$ -terms then so is  $(AB)$ 
  - **application**: think of this as the function  $A$  that is about to be applied to the input  $B$
3. if  $x$  is a variable and  $M$  is a  $\lambda$ -term, then so is  $(\lambda x.M)$ 
  - **abstraction**: think of this as the function that on input  $x$  computes  $M$ .

## Examples

- $(xy)$
- $(\lambda x.x)$
- $(\lambda x.(yx))$
- $((\lambda x.(yx))z)$

# Notation

- We often drop outermost parentheses; so  $(xy)$  is written  $xy$ .
- We write  $A, B, C, \dots$  denote  $\lambda$ -terms
- Application associates to the left:
  - $((FA_1)A_2)$  is written  $FA_1A_2$
- Abstraction associates to the right:
  - $(\lambda x_1.(\lambda x_2.A))$  is written  $\lambda x_1x_2.A$
  - This notation allows us to think that  $A$  is a function of two variables (cf. Currying)

## Examples

- $\lambda x.yx$  is shorthand for  $(\lambda x.(yx))$ , not  $(\lambda x.y)x$
- $\lambda x.yxz$  is shorthand for  $\lambda x.((yx)z)$ , not  $(\lambda x.(yx))z$
- $((\lambda y.(\lambda x.(yx)))z)w$  is written  $(\lambda yx.yx)zw$

# Notation

$(\lambda xy.xyx)ab$  is shorthand for  
 $(\lambda x.(\lambda y.(xyx)))ab$  which is shorthand for  
 $(\lambda x.(\lambda y.((xy)x)))ab$  which is shorthand for  
 $((\lambda x.(\lambda y.((xy)x)))a)b.$

# Rewrite rules: conversions

$\alpha$ -conversions correspond to variable renaming

Rename bound variables, e.g.,

$$\lambda x.(\lambda y.x) \xrightarrow{\alpha} \lambda z.(\lambda y.z)$$

This is useful to avoid name clashes (like in predicate logic).

# Rewrite rules: reductions

$\beta$ -reductions correspond to function evaluation  
“plugging arguments into functions”

- Only applies to  $\lambda$ -terms of the form  $(\lambda x.M)N$
- Simultaneously substitute all free occurrences of  $x$  in  $M$  by  $N$ :

$$(\lambda x.M)N \xrightarrow{\beta} M[N/x]$$

$$(\lambda x.xx)(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)(\lambda y.y) \xrightarrow{\beta} (\lambda y.y)$$

$$(\lambda xy.xyx)ab \xrightarrow{\beta} (\lambda y.aya)b \xrightarrow{\beta} aba$$

# Take care not to capture free variables

- $M[N/x]$  means substitute free occurrences of  $x$  in  $M$  by  $N$ .
- However, we must take care that we do not capture free variables.
- I.e., Do  $M[N/x]$  only if no free variable in  $N$  is bound in  $M$
- Otherwise, rename bound variables in  $M$  first using  $\alpha$ -conversions
- E.g.,
  - $M = \lambda x.yx, N = \lambda z.xz$
  - $(\lambda y.M)N \xrightarrow{\beta} M[N/y] = \lambda x.Nx = \lambda x.(\lambda z.xz)x$ 
    - This is not what we intend. Why?
  - $(\lambda y.M)N \xrightarrow{\alpha} (\lambda y(\lambda x'.yx'))N \xrightarrow{\beta} \lambda x'.Nx' = \lambda x'.(\lambda z.xz)x'$

# Rewrite Rules

$$\begin{aligned} ((\lambda x.(\lambda y.((xy)x)))a)b &= \\ (((\lambda x.M)a)b) &\xrightarrow{\beta} \\ (M[a/x])b &= \\ (\lambda y.(ay)a)b &\xrightarrow{\beta} (ab)a \end{aligned}$$

- Aaaah, too many brackets!
- Use shorthand:

$$\begin{aligned} (\lambda xy. xyx)ab &\xrightarrow{\beta} (\lambda y.aya)b \\ &\xrightarrow{\beta} aba \end{aligned}$$

# Computation?

- Computation is done by  $\beta$ -reducing subterms for as long as possible.
- A term is in **normal form** if no  $\beta$ -reductions apply.
- Normal forms may not exist,
  - e.g.,  $(\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} ??$
- If they do exist, they are unique up to  $\alpha$ -renaming.

# Example: Encoding Booleans

$$F = \lambda xy.y \quad T = \lambda xy.x \quad A = \lambda xy.xyx$$

$$\begin{aligned} ATF &= (\lambda xy.xyx)TF \\ &\xrightarrow{\alpha} (\lambda pq.pqp)TF \\ &\xrightarrow{\beta} (\lambda q.TqT)F \\ &\xrightarrow{\beta} TFT = (TF)T \\ &\xrightarrow{\beta} ((\lambda xy.x)F)T \\ &\xrightarrow{\beta} (\lambda y.F)T \\ &\xrightarrow{\beta} F \end{aligned}$$

Similarly:

$$AFT \rightarrow^* F \quad ATT \rightarrow^* T \quad AFF \rightarrow^* F$$

So  $A$  behaves like  $\wedge$  if we view  $T$  as true and  $F$  as false.

# Example: Encoding arithmetic

$$F = \lambda xy.y \quad S = \lambda nfx.f(nfx)$$

$$SF \xrightarrow{*} \lambda xy.xy$$

$$S(SF) \xrightarrow{*} \lambda xy.x(xy)$$

$$S(S(SF)) \xrightarrow{*} \lambda xy.x(x(xy))$$

⋮

- These are called *Church Numerals*
- So  $S$  behaves like  $+1$  if we view  $F$  as  $0$
- Intuition: numbers are used to do things
  - $(\lambda xy.x(x(xy)))fa \xrightarrow{\beta} f(f(fa))$ , i.e., apply  $f$  to  $a$  three times
  - $f$  = “bite” and  $a$  = “apple” then this means apply the bite function to the apple three times
- Addition, multiplication, can also be encoded.

# What about recursion?

**Every  $\lambda$ -term  $G$  has a fixed point**

i.e., a  $\lambda$ -term  $X$  such that  $X \xrightarrow{*} GX$ .

- $W = \lambda x.G(xx)$ ,  $X = WW$ .
- Then  $X = WW = (\lambda x.G(xx))W \xrightarrow{\beta} G(WW) = GX$ .

**This process can be abstracted into a function  $Y$  that given  $G$  returns a fixpoint of  $G$**

- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$  is called a fixed point combinator.
  - Then  $YG \xrightarrow{\beta} X$ , so  $YG \xrightarrow{*} G(YG)$

# Why is finding fixpoints useful?

- Solving recursive equations can be seen as finding fixpoints
- Fixed point combinators can be used to implement recursive definitions.
- CFGs are recursive, and the language of a CFG is a fixpoint.
  - $\{a^n b^n : n \geq 0\}$  is the unique  $\subseteq$ -minimal solution of the equation  $X = aXb \cup \epsilon$ .
- Addition can be recursively defined using  $+1$ .
- Multiplication can be recursively defined using addition.

# Outlook

- The  $\lambda$ -calculus consists of objects called  $\lambda$ -terms and rules for manipulating them:  $\alpha$ -conversion and  $\beta$ -reduction.
- Turing (1936) proved that Lambda-calculus can simulate Turing machines and vice versa.
- Functional programming languages are based on the  $\lambda$ -calculus.
- You will see more lambda-calculus in COMP3109: Programming Languages and Paradigms

# **COMP2022|2922**

## **Models of Computation**

### **Lecture 12: USS, Exam Prep**

**Presented by**

Sasha Rubin

School of Computer Science



THE UNIVERSITY OF  
SYDNEY



## Student feedback matters. It is used to improve the course!

- e.g., based on USS and tutor feedback from 2019, this year I:
  1. started using textbooks in this course
  2. made the connection between logic and models of computation clearer (A1, A2)
  3. made the content more rigorous and precise
- e.g., based on the Canvas survey in week 6 this year I:
  1. made handwritten content added to slides is more legible.
  2. streamlined Q+A by stopping every main idea (roughly every few slides) to ask for questions.
  3. took steps to improve the structure and use of Ed forum.
  4. cut out some extra material in lectures so that they are less full.

**Student feedback matters. It is used to improve the course!**

- USS is anonymous and confidential.
- Usually it is students that were unhappy with some part of the course that respond. This is good! But we also need to know what went **well** so that we don't make unnecessary changes to the course and so that credit goes where credit is due.
- Links to the surveys are here:  
<https://edstem.org/courses/4385/discussion/350057>
- Please fill them in now (10 minutes set aside in the final lecture).

# Overview of exam

COMP2022 FINAL EXAM (S2, 2020)



THE UNIVERSITY OF  
SYDNEY

CONFIDENTIAL EXAM PAPER

This paper is not to be removed from the exam venue.

Computer Science

EXAMINATION

Semester 2 - Practice, 2020

COMP2022 Models of Computation

**EXAM WRITING TIME:** 3 hours

**READING TIME:** 10 minutes

#### EXAM CONDITIONS:

This is an OPEN book examination. You are allowed to use passive information sources (i.e., existing written materials such as books and websites); however, you must not ask other people for answers or post questions on forums; always answer in your own words. You must not reveal the questions to anyone else. All work must be **done individually** in accordance with the University's "Academic Dishonesty and Plagiarism" policies.

#### INSTRUCTIONS TO STUDENTS:

1. Type your answers in your text editor and submit a **single PDF** via Canvas. Figures/diagrams can be rendered any way you like (hand drawn, latex, etc), as long as they are perfectly readable and part of the submitted PDF.
2. Start by typing your student ID at the top of the first page of your submission. Do **not** type your name.
3. Submit only your answers to the questions. Do **not** copy the questions. Start each of the five problems on a new page.
4. **You must always justify your answer/give reasons, unless explicitly stated otherwise.**

# Overview of exam

**Five sections, each with multiple questions, roughly weighted in proportion to the number of lectures.**

1. Propositional logic (20%)
2. Predicate logic (20%)
3. Regular languages (30%)
4. Context-free languages (20%)
5. Universal models of computation (10%)

# General skills that are assessable

1. How to **write and apply** recursive definitions to do with computational problems.
2. How to **model** computational problems using models of computation (such as REs, DFAs, NFAs, CFGs, TMs).
3. How to **reason** about computational problems and models of computation.

What follows is a list of specific notions/skills that we have taught.  
It is fairly representative of what you should master for the exam,  
but is **not exhaustive**.

# Propositional logic

1. Know all definitions (propositional formula, subformula, logically equivalence, satisfiability, validity, NNF, CNF, etc.) and the connections between these notions.
2. Express propositional logic formulas in English and vice-versa.
3. Recognise and apply shorthands for propositional logic.
4. Prove that two formulas are logically equivalent.
5. Check if a formula is valid.
6. Put a formula in NNF, CNF.
7. Construct proofs in natural deduction.
8. Analyse algorithms for propositional logic, e.g., for testing satisfiability, validity, logical equivalence, for putting formulas into NNF, CNF.

# Predicate logic

1. Know all definitions (signature, term, formula, subformula, sentence, logically equivalence, satisfiability, validity, PNF, etc.) and the connections between these notions.
2. Express predicate logic formulas in English and vice-versa.
3. Apply the recursive definition of truth-value.
4. Put formulas into PNF.
5. Construct proofs in natural deduction.

# Regular languages

1. Know all definitions (REs, DFAs, NFAs, etc.) and the connections between these notions.
2. Show that a given language is regular (by building RE, DFA, NFA for it).
3. Describe the language of a given RE/DFA/NFA.
4. Prove that a given language is not regular (by using the PHP).
5. Transform RE to NFA with epsilon transitions to NFA without epsilon transitions to DFA to RE.
6. Reason about regular languages, including closure properties.

# Context-free languages

1. Know all definitions (CFGs, generate, yield, CNF, etc.) and the connections between these notions.
2. Create a CFG for a given CFL, and argue why it is correct.
3. Describe the language of a given CFG.
4. Show that a given CFG is ambiguous.
5. Argue that a given CFG is not ambiguous.
6. Convert a CFG to CNF.
7. Apply the CYK algorithm to a grammar in CNF and an input string.
8. Reason about context-free languages, including closure properties.

# Universal models of computation

1. Know all definitions (Turing machine, Turing-decidable, Turing-recognisable, undecidable, unrecognisable, lambda-calculus, etc.) and the connections between these notions as well as to weaker models of computations (DFAs, NFAs, CFGs, etc).
2. Describe the language recognised by a given TM.
3. Create a TM for recognising or deciding a given language.
4. Reason about Turing-decidable/recognisable languages, including closure properties.
5. Reduce a lambda-calculus expression to normal form.
6. Recognise and apply shorthands for lambda-terms.