COMP2022: Formal Languages and Logic

2018, Semester 2, Week 4

Joseph Godbehere

23rd August, 2018

THE UNIVERSITY OF SYDNEY

Outline

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)
  - Recursion (fixed point, using it)
  - β-equivalence
  - Computational power
  - Functional Programming
- Automata Theory
  - Background concepts
  - DFA
  - Regular languages
  - NFA

Operators

- Application
  - Notation: $(A \cdot B)$
  - Expression B is applied to expression A
  - Left associative: $ABCDEF = (((((AB)C)D)E)F)$

Operators

- Application
  - Notation: $(A \cdot B)$
  - Expression B is applied to expression A
  - Left associative: $ABCDEF = (((((AB)C)D)E)F)$

- Abstraction
  - $(\lambda x.M)$
  - Variable x is abstracted in expression M
  - Right associative: $(\lambda abcdef.M) = (\lambda a.(\lambda b.(\lambda c.(\lambda d.(\lambda e.(\lambda f.M))))))$

Free Variables

Formally the set of free variables of an expression is defined inductively like this:

$$FV(x) = x \qquad \text{(where } x \text{ is an atom)}$$

$$FV(MN) = FV(M) \cup FV(N) \qquad (M, N \text{ are expressions})$$

$$FV(\lambda x.M) = FV(M) - \{x\}$$

Any variable in an expression that is not free, is bound.

Rewriting

- $M[x := N]$

- Expression M, but all free occurrences of x are replaced with N

- e.g.
  - $(xyz\lambda x.(zxz))[x := A] \equiv (Ayz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[y := B] \equiv (xBz\lambda x.(zxz))$
  - $(xyz\lambda x.(zxz))[z := C] \equiv (xyC\lambda x.(CxC))$

α-reduction

- $\lambda x.M = \lambda y.(M[x := y])$

- Renames a λ to remove a name conflict

- y must be a *new* variable
  - Do not choose a symbol that is already in the expression
  - It's usually easiest to start with the innermost λ

β-REDUCTION

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

- Reduces an application into an abstraction

- Note: the free occurrences of x in M are exactly the set of occurrences which are bound to the $\lambda x.$ in $(\lambda x.M)$

- Do α-reductions first if necessary

η-reduction

- If x is *not* free in M (i.e. $x \notin FV(M)$), then we can write:

$$\lambda x.Mx = M$$

- Reduces an abstraction directly

Naming

Naming conflicts:

- If a variable is free, but there is also λ with the same label
- If more than one λ uses the same label
- They can cause problems with β reduction!

To fix them:

- Apply α-reduction repeatedly to change the λ labels
- Always rename to a label not already in use
- Work from the innermost λ to the outermost one

Fixed Point Combinators

A *fixed point combinator* is a combinator which has a fixed point.

We say F has a fixed point if $\exists X\ (FX = X)$

i.e. some input X exists which, when applied to F, outputs X again.

Fixed Point Theorem

1. $\forall F \; \exists X \; FX = X$
   - All functions have a fixed point
   - Proven last week

2. There is a fixed point combinator Y such that

$$\forall F \; F(YF) = YF$$

   - $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
   - i.e. for any function F, YF is a fixed point of F.
   - Proven last week
   - Very useful for recursion

Recursion

Recursive functions often look like this:

$$F = \Big(\lambda xyz.(\text{condition})(\text{base case})(\text{recursive case calls } F)\Big)$$

If we need more cases, we can just add more logic

We can't directly refer to the function F within itself

We can define it like this instead:

$$H = \Big(\lambda fxyz.(\text{condition})(\text{base case})(\text{recursive case calls } f)\Big)$$

$$F = Y\ H$$

$$H = \Big(\lambda fxyz.(\text{condition})(\text{base case})(\text{recursive case calls } f)\Big)$$

$$F = Y\ H$$

Evaluation:

$$\begin{aligned}
F\ a\ b\ c &= (Y\ H)\ a\ b\ c \\
&= H\ (Y\ H)\ a\ b\ c \qquad\qquad (YH \text{ is a fixed point of } H) \\
&= \Big(\lambda fxyz.(\text{condition})(\text{base})(\text{call(s) to } f)\Big)\ (Y\ H)\ a\ b\ c \\
&= \Big(\lambda xyz.(\text{condition})(\text{base})(\text{call(s) to } (Y\ H))\Big)\ a\ b\ c
\end{aligned}$$

Notice that we managed to call $(Y\ H) = F$ from within F

- recursion!

Reducing the Y Combinator

We don't need to reduce $(Y\ H)$ directly

- In the recursive calls, it expands to $H\ (Y\ H)$, then we reduce the leftmost H
- The $(Y\ H)$ will disappear when we reach the base cases

Example: last week's tutorial questions

We wanted a function that would build a list $\{x_1 ... x_n\}$ from an expression like $(F n x_1 ... x_n)$

Key ideas:

- Recurse n times
- Each recursion consumes up one argument
- Base case $n = 0$

$$H = \lambda f t n.(ISZERO\ n)\ t\ (\lambda e.f\ (CONS\ e\ t)\ (PRED\ n))$$
$$F = YHNIL$$

- Prepend the next argument to the list with $(CONSet)$, and recursively call $f\ \mathit{list}\ (n-1)$ with the remaining arguments

Example: last week's tutorial questions

$$
\begin{aligned}
& F\ 3\ a\ b\ c \\
&= Y\ H\ NIL\ 3\ a\ b\ c \\
&= H\ (Y\ \ H)\ NIL\ 3\ a\ b\ c \\
&= \Big(\lambda ftn.(ISZERO\ n)\ t\ (\lambda e.f\ (CONS\ e\ t)\ (PRED\ n))\Big)\ (Y\ \ H)\ NIL\ 3 \\
&= \ldots = (ISZERO\ 3)\ NIL\ \big(\lambda e.(Y\ \ H)\ (CONS\ \ e\ \ NIL)\ (PRED\ 3)\big)\ a\ b\ c \\
&= \ldots = FALSE\ \ NIL\ \big(\lambda e.(Y\ \ H)\ (CONS\ \ e\ \ NIL)\ (PRED\ 3)\big)\ a\ b\ c \\
&= \ldots = \big(\lambda e.(Y\ \ H)\ (CONS\ \ e\ \ NIL)\ (PRED\ 3)\big)\ a\ b\ c \\
&= \ldots = (Y\ \ H)\ (CONS\ \ a\ \ NIL)\ (PRED\ 3)\ b\ c \\
&= \ldots = (Y\ \ H)\ (CONS\ \ a\ \ NIL)\ 2\ b\ c \\
&= \ldots = (Y\ \ H)\ (CONS\ \ b\ (CONS\ \ a\ \ NIL))\ 1\ c \\
&= \ldots = (Y\ \ H)\ (CONS\ \ c\ (CONS\ \ b\ (CONS\ \ a\ \ NIL))\ 0 \\
&= \ldots = (CONS\ \ c\ (CONS\ \ b\ (CONS\ \ a\ \ NIL)))
\end{aligned}
$$

Example: last week's tutorial questions

Reversing a list:

$$H = \lambda fab.(ISNIL\ a)\ b\ (f\ (TAIL\ a)(CONS\ (HEAD\ a)\ b))$$

$$F = \lambda a.Y\ H\ a\ NIL$$

$$
\begin{aligned}
& F\ \{c, b, a\} \\
=& (\lambda a.Y\ H\ a\ NIL)\{c, b, a\} \\
=& Y\ \ H\ \{c, b, a\}\ NIL \\
=& H\ \ (Y\ \ H)\ \{c, b, a\}\ NIL \\
=& \Big(\lambda fab.(ISNIL\ a)\ b\ (f\ (TAIL\ a)(CONS\ (HEAD\ a)\ b))\Big)\ (Y\ \ H)\ \{c, b, a\}\ NIL \\
=& ... = (Y\ \ H)\ (TAIL\ \{c, b, a\})(CONS\ (HEAD\ \{c, b, a\})\ NIL) \\
=& ... = (Y\ \ H)\ \{b, a\}(CONS\ \ c\ \ NIL) \\
=& ... = (Y\ \ H)\ \{b, a\}\{c\} \\
=& ... = (Y\ \ H)\ \{a\}\{b, c\} \\
=& ... = (Y\ \ H)\ NIL\{a, b, c\} \\
=& ... = \{a, b, c\}
\end{aligned}
$$

Notation

We've mostly been using the $=$ sign everywhere. This isn't strictly correct!

We could have used more precise notation, such as:

- $M \equiv N$: M is the same as N
- $M \equiv N$: M is equivalent to N (has the same effect)
- $M \twoheadrightarrow_\beta N$: M β-reduces to N
- $M \rightarrow_\beta N$: M β-reduces to N (in one step)
- $M =_\beta N$: M is β-convertable to N

We've avoided using these so far for the sake of simplicity

β-normal form

An expression is in β-normal form if no β-reductions are available

Not all expressions have normal forms

- e.g. $(\lambda x.xx)(\lambda x.xx)$ does not have a normal form

Normalisation Theorem: If a normal form exists, it can always be found by following the leftmost reduction

- $((\lambda x.xx)(\lambda x.xx))(\lambda ab.b)$ does not have a normal form because the leftmost reduction loops infinitely

- $(\lambda ab.b)\big((\lambda x.xx)(\lambda x.xx)\big)$ has a normal form, $(\lambda b.b)$, even though following the reduction on the right would not have found it

β-equivalence

Two expressions are β-equivalent if they have a common reduct

- This doesn't necessarily mean we can β-reduce between them directly!

- e.g.
  - $(\lambda xy.x)ab$ and $(\lambda xy.x)ac$ both reduce to a
  - Therefore, they are β-equivalent
  - ... even though we have no way to reduce between them!

- We've been using this property often, without stating it

β-equivalence

An interesting example of β-equivalence is the Y Combinator:

$YF =_\beta F(YF)$, but neither $YF \twoheadrightarrow_\beta F(YF)$ nor $F(YF) \twoheadrightarrow_\beta YF$

Turing discovered another fixed point combinator $\Theta \equiv \big(\lambda xy.y(xxy)\big)\big(\lambda xy.y(xxy)\big)$, which has the property that $\Theta F \twoheadrightarrow_\beta F(\Theta F)$

There are an infinite number of fixed point combinators in untyped lambda calculus!

Church-Rosser theorem

If $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$, then there exists some N_3 such that $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$

This is the property we've been using to claim that the order in which we reduce the subexpressions doesn't change the result.

We do need to be a little bit careful sometimes, as some paths do not lead to the normal form, although they remain β-equivalent to it!)

Computational Power

Lambda calculus can compute all the computable (recursive) functions

- Proof is beyond the scope of this course
- Key points:
  - We have numbers and arithmetic operations
  - All recursive functions are λ-definable
  - Lambda Calculus is consistent (you can't prove false statements)
  - If a normal form exists, we can find it (Normalisation Theorem)
  - If a normal form exists, it is unique

Essentially, given any problem you can express with set theory (i.e. maths), then *if* it's possible to solve (compute) it, lambda calculus can do so.

Computational Power

Later in the course we will look at Turing machines.

The Church-Turing hypothesis shows that lambda calculus and Turing machines have the same computational power.

The proof is relatively simple. If we have time at the end of the course, we will show you.

Why?

If we can do just as much using the imperative paradigm, then *why* use the functional paradigm?

$\equiv$ is not $=$

- They have equivalent computational power
- They are not the *same*
- Some things are *easier* with each approach

Positive traits of functional paradigm

- Often easier to write concurrent code (Church-Rosser)
- Often easier to prove correctness
- Lazy evaluation
  - Easy to work with vast / infinite data sets
- Function composition makes some tasks much simpler
  - e.g. Map / fold / reduce templates allow us to define ways to ‘crawl’ over our data. Performing different transformations now just becomes arguments to these methods, instead of entirely new functions.
- Inherent immutability
  - Our functions define relationships between the existing structure and the one we want
  - Notions like undo/redo become almost trivial

Hybrid languages

NOT an either/or choice.

Most languages blend imperative and functional paradigms:

- e.g. Java, C++, Python, ...
- ... because it's so useful to be able to do both

Outline

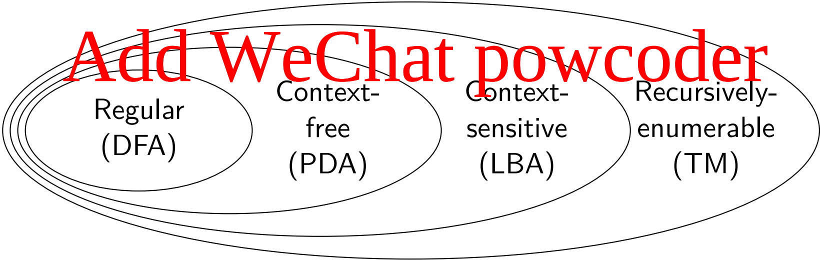
- Lambda Calculus
  - Concepts (Operators, FV, Reductions)
  - Recursion (fixed point, using it)
  - β-equivalence
  - Computational power
  - Functional Programming
- **Automata Theory**
  - Background concepts
  - DFA
  - Regular languages
  - NFA

Alan Turing

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

Noam Chomsky

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



Alphabet

An *alphabet* is a finite, non-empty set of symbols, denoted Σ

For example:

- Binary: $\Sigma = \{0, 1\}$
- All lower case letters: $\Sigma = \{a, b, c, ..., z\}$
- Alphanumerics: $\Sigma = \{a, ..., z, A, ..., Z, 0, ..., 9\}$
- ASCII, unicode
- Set of signals used by a protocol

String

- A *string* is a finite sequence of symbols from Σ. For example "01110", "alice", "Bnode.java"
- The *length* of a string w, denoted $|w|$, is equal to the number of symbols in the string, e.g. if $x = abcd$ then $|x| = 4$

- Σ^* denotes the set of all strings over the alphabet Σ
- ε (epsilon) denotes the *empty string*
  - $|\varepsilon| = 0$
  - If $x = a\varepsilon b\varepsilon c\varepsilon d\varepsilon$ then $|x| = ?$

- xy = the concatenation of two strings x and y
- x^n the string x repeated n times

String

- A *string* is a finite sequence of symbols from Σ. For example "01110", "alice", "Bnode.java"
- The *length* of a string w, denoted $|w|$, is equal to the number of symbols in the string. e.g. if $x = abcd$ then $|x| = 4$

- Σ^* denotes the set of all strings over the alphabet Σ
- ε (epsilon) denotes the *empty string*
  - $|\varepsilon| = 0$
  - If $x = ab\varepsilon c\varepsilon d\varepsilon$ then $|x| = ?$ 4

- xy = the concatenation of two strings x and y
- x^n the string x repeated n times

Powers of an alphabet

Let Σ be an alphabet, then:

- Σ^k is the set of all strings of length k
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$ (i.e. all strings, including ε)
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$ (i.e. all strings except ε)

Languages

- A *language* L over alphabet Σ is a subset of Σ^* (i.e. $L \subseteq \Sigma^*$)
- Examples:
  - Let L be the language of all strings consisting of n 0s followed by n 1s:
    - $L = \{\varepsilon, 01, 0011, 000111, \ldots\}$
    - $L = \{0^n 1^n | n \geq 0\}$
  - Let L be the language of all strings with an equal number of 0s and 1s:
    - $L = \{\varepsilon, 01, 10, 0011, 0101, 0110, 1001, 1010, 1100, \ldots\}$
- $\emptyset$ denotes the empty language
  - Beware: $\{\varepsilon\}$ is NOT $\emptyset$

The Membership problem

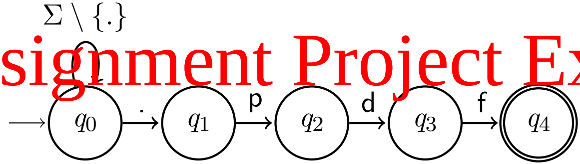
Problem:
Given a string $w \in \Sigma^*$ and a language L over Σ, decide whether or not $w \in L$

Example:
Let $w = 100011$
Does w belong to the language of strings with an equal number of 0s and 1s?

Outline

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)
  - Recursion (fixed point, using it)
  - β-equivalence
  - Computational power
  - Functional Programming
- Automata Theory
  - Background concepts
  - **DFA**
  - Regular languages
  - NFA

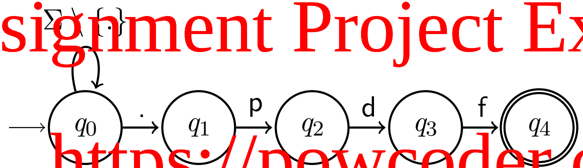
Introductory Example



Formal model which remembers only a finite amount of information

- Information is represented by the *state* (circles)
- *Transition* rules (arrows) define state changes according to input
- *Start* state is denoted $\rightarrow$
- *Accept* state(s) denoted by double circle
- The set of all possible input symbols is the *alphabet*, Σ

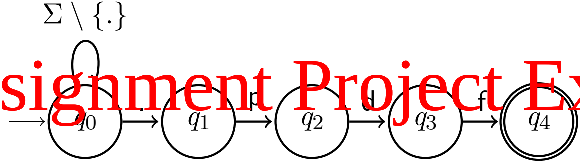
Introductory Example



Given a sequence of input symbols:

- Begin in the *start* state
- Follow one transition for each symbol in the input string
- After reading the entire input string:
  - The input is *accepted* if the automaton is in an *accept* state
  - The input is *rejected* if it is not

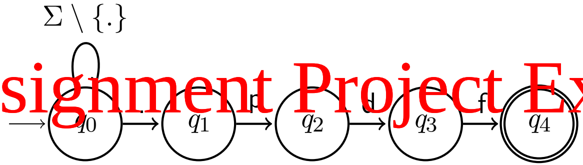
Introductory Example



Example input "example.pdf":

- loops on q_0 through "example"
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

Introductory Example



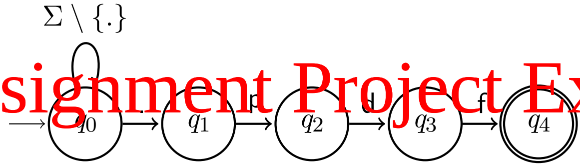
Example input "example.pdf":

- loops on q_0 through 'example'
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

How about:

- .pdf
- pdf
- example.pd
- example.pdf.pdf

Introductory Example



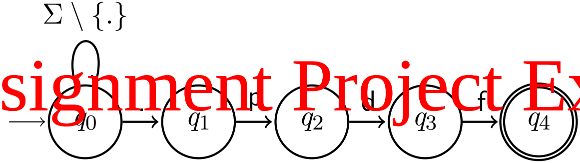
Example input "example.pdf":

- loops on q_0 through "example"
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

How about:

- .pdf **Yes**
- pdf
- example.pd
- example.pdf.pdf

Introductory Example



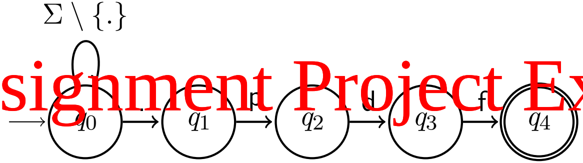
Example input "example.pdf":

- loops on q_0 through "example"
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

How about:

- .pdf **Yes**
- pdf **No**
- example.pd
- example.pdf.pdf

Introductory Example



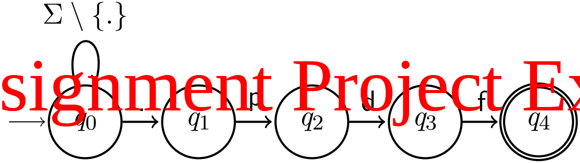
Example input "example.pdf":

- loops on q_0 through 'example'
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

How about:

- .pdf **Yes**
- pdf **No**
- example.pd **No**
- example.pdf.pdf

Introductory Example



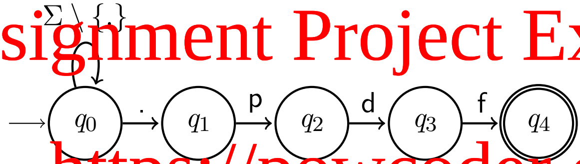
Example input "example.pdf":

- loops on q_0 through "example"
- moves to q_1 when it scans '.'
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an *accept* state, therefore "example.pdf" is accepted

How about:

- .pdf **Yes**
- pdf **No**
- example.pd **No**
- example.pdf.pdf **No(!)**

Introductory Example

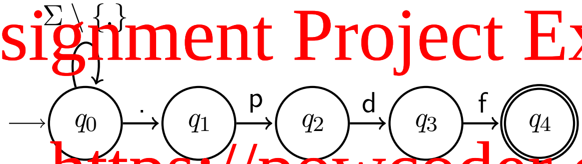


Recall:

- Information is represented by the *state* (circles)

What information is represented by state q_1? By q_2?

Introductory Example



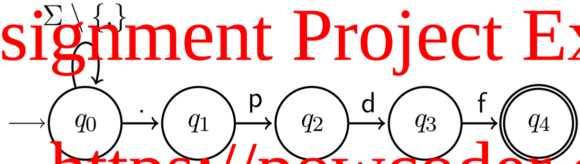
Recall:

- Information is represented by the *state* (circles)

What information is represented by state q_1? By q_2?

- q_1: We have scanned any number of letters, followed by ‘.’

Introductory Example



Recall:

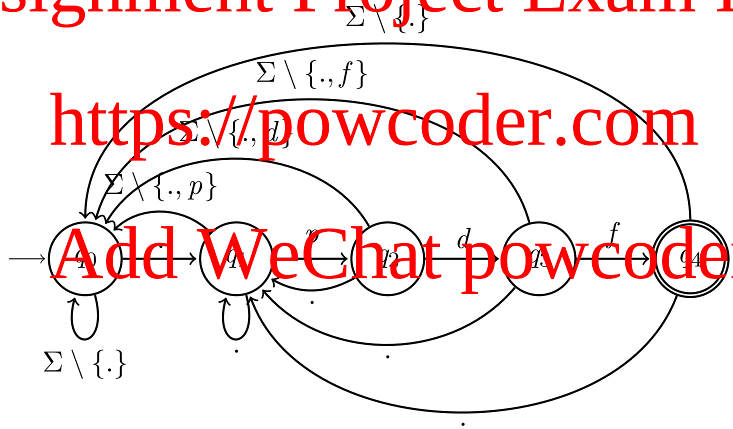
- Information is represented by the *state* (circles)

What information is represented by state q_1? By q_2?

- q_1: We have scanned any number of letters, followed by "."
- q_2: We have scanned any number of letters, followed by ".p"

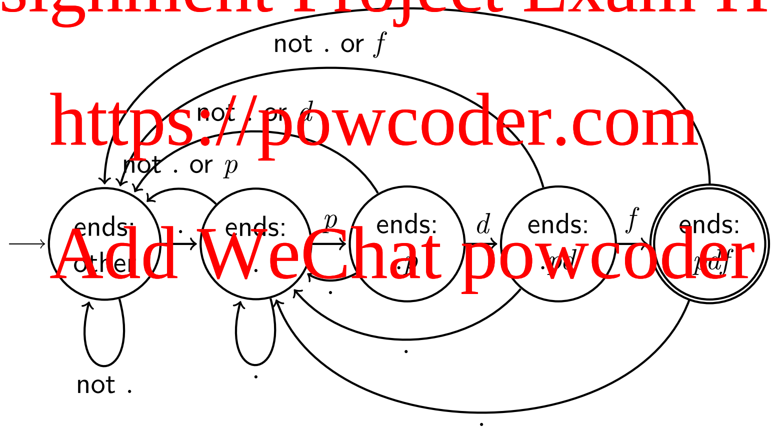
Our first valid DFA

A properly defined DFA will have a transition for every symbol, from every state:



Alternative notation

You can be more descriptive if you like:



Deterministic Finite Automata (DFA)

Informal definition:

1. They have a finite set of *states*
2. They have a finite *alphabet* of symbols
3. They have one *start state*
4. They have a behaviour given by *transitions*
   - Exactly one for each alphabet symbol, from each state
5. They have *accept state(s)*

Deterministic Finite Automata (DFA)

Formal definition:

A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

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

Draw a DFA from the definition

Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ where:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

4. $q_0 \in Q$ is the start state
5. $F = \{q_1\}$

Now we can draw M_1:

Draw a DFA from the definition

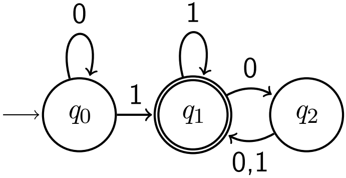
Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ where:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

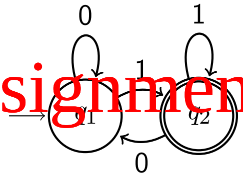
	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

4. $q_0 \in Q$ is the start state
5. $F = \{q_1\}$

Now we can draw M_1:



Define a DFA from a diagram

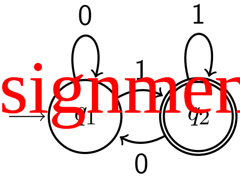


We can formally describe M_2 as:

1. $Q =$
2. $\Sigma =$
3. $\delta : Q \times \Sigma \to Q$ is given by
4. The start state is:
5. The set of accept states is: $F =$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram



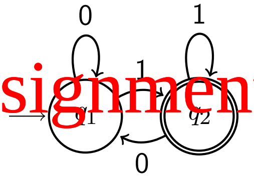
We can formally describe M_2 as:

1. $Q = \{q_1, q_2\}$
2. $\Sigma =$
3. $\delta : Q \times \Sigma \to Q$ is given by

4. The start state is:
5. The set of accept states is: $F =$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram

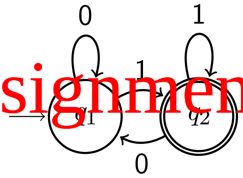


We can formally describe M_2 as:

1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by
4. The start state is:
5. The set of accept states is: $F =$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram



We can formally describe M_2 as:

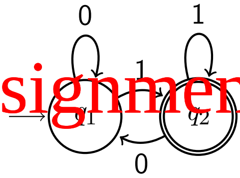
1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

4. The start state is:
5. The set of accept states is: $F =$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram



We can formally describe M_2 as:

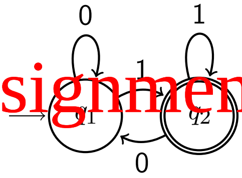
1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

4. The start state is: q_1
5. The set of accept states is: $F =$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram



We can formally describe M_2 as:

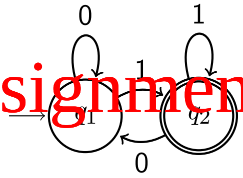
1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

4. The start state is: q_1
5. The set of accept states is: $F = \{q_2\}$

Some strings where M_2 ends on an accept state are:

Define a DFA from a diagram



We can formally describe M_2 as:

1. $Q = \{q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. $\delta : Q \times \Sigma \to Q$ is given by

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

4. The start state is: q_1
5. The set of accept states is: $F = \{q_2\}$

Some strings where M_2 ends on an accept state are:
01, 0111, 0101

Outline

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)
  - Recursion (fixed point, using it)
  - β-equivalence
  - Computational power
  - Functional Programming
- Automata Theory
  - Background concepts
  - DFA
  - **Regular languages**
  - NFA

Formal definition of computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton, and
Let $w = w_1 w_2 \cdots w_n$ be a string over the alphabet Σ
Then M accepts w if there exists a sequence of states $r_0 r_1 \cdots r_n$ from Q which satisfy the following three conditions:

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, \ldots, n-1$
3. $r_n \in F$

Language of an automaton

- Automata of all kinds define languages
- Let A be a language, and M be an automaton.
- We say that M *recognises* A if and only if

$$A = \{w | M \text{ accepts } w\}$$

- We often denote the language recognised by M as $L(M)$
- $L(M)$ is the set of all strings labelling paths from the start state of M to any accept state in M

Regular Language

A language is *regular* if and only if there exists a finite automaton which recognises it.

Exercise:
Prove that $A = \{w|w$ is the empty string or ends with a $0\}$ is a regular language.

Regular Language

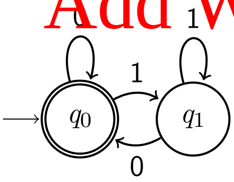
A language is *regular* if and only if there exists a finite automaton which recognises it.

Exercise:

Prove that $A = \{w|w$ is the empty string or ends with a $0\}$ is a regular language.

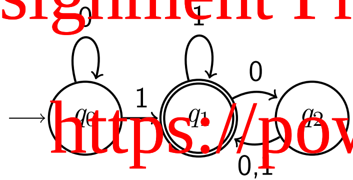
Solution:

The following DFA recognises A, therefore A must be regular:



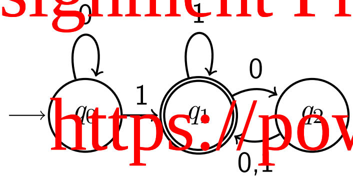
Examples of Regular Languages

What language is accepted by M_1?



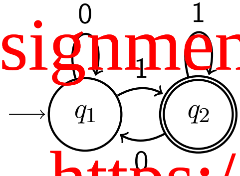
Examples of Regular Languages

What language is accepted by M_1?



$$L(M_1) = \{w | w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}$$

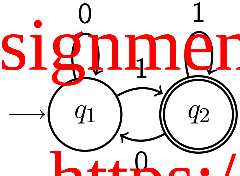
Examples of Regular Languages



What language is accepted by M_2?

What is the language recognised by the automaton obtained by inversing the accept and start states in M_2?

Examples of Regular Languages



What language is accepted by M_2?

$$L(M_2) = \{w | w \text{ ends with a } 1\}$$

What is the language recognised by the automaton obtained by inversing the accept and start states in M_2?

Designing finite automata

"Reader as Automaton" method: The *states* encode what you need to *remember* about the string as you are reading it.

Example: Let $\Sigma = \{0, 1\}$. Devise an automaton which accepts the language consisting of strings with an odd number of 1s.

We need to remember:

- There has been an even number of 1s so far
- There has been an odd number of 1s so far

So we will need exactly two states. Then it just remains to add the transitions and define the start and accept states.

Designing finite automata

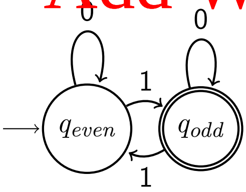
"Reader as Automaton" method: The *states* encode what you need to *remember* about the string as you are reading it.

Example: Let $\Sigma = \{0, 1\}$. Devise an automaton which accepts the language consisting of strings with an odd number of 1s.

We need to remember:

- There has been an even number of 1s so far
- There has been an odd number of 1s so far

So we will need exactly two states. Then it just remains to add the transitions and define the start and accept states.



Example 2

Devise an automaton which accepts the language of strings which begin and end with 1, over $\{0, 1\}$

We need to remember:

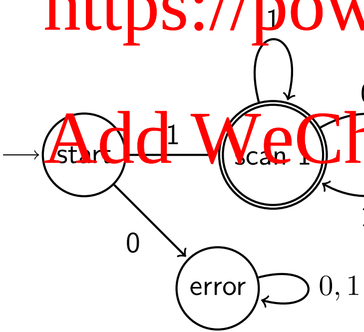
- Whether or not the string began with 1
- Whether or not the last character scanned was a 1

Example 2

Devise an automaton which accepts the language of strings which begin and end with 1, over $\{0, 1\}$

We need to remember:

- Whether or not the string began with 1
- Whether or not the last character scanned was a 1



Example 3

Devise an automaton which accepts the language of binary strings which do not contain 11

What do we need to remember?

Example 3

Devise an automaton which accepts the language of binary strings which do not contain 11

What do we need to remember?

- A: String so far has no 11, does not end in 1

Example 3

Devise an automaton which accepts the language of binary strings which do not contain 11

What do we need to remember?

- A: String so far has no 11, does not end in 1
- B: String so far has no 11, but ends with a single 1

Example 3

Devise an automaton which accepts the language of binary strings which do not contain 11

What do we need to remember?

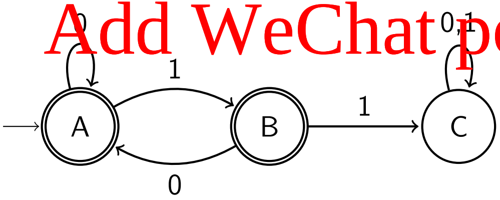
- A: String so far has no 11, does not end in 1
- B: String so far has no 11, but ends with a single 1
- C: Consecutive 1s have been seen

Example 3

Devise an automaton which accepts the language of binary strings which do not contain 11

What do we need to remember?

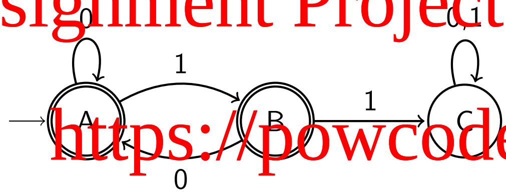
- A: String so far has no 11, does not end in 1
- B: String so far has no 11, but ends with a single 1
- C: Consecutive 1s have been seen



String in a Language

Let $L = \{w | w \in \{0,1\}^* \text{ and } w \text{ does not contain consecutive 1s}\}$

The following DFA accepts the language L

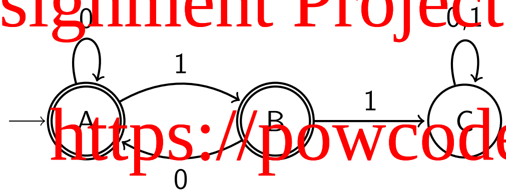


Is the string 101 in L?

String in a Language

Let $L = \{w | w \in \{0,1\}^* \text{ and } w \text{ does not contain consecutive 1s}\}$

The following DFA accepts the language L



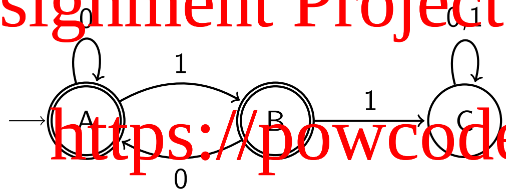
Is the string 101 in L?

- Start at A

String in a Language

Let $L = \{w | w \in \{0,1\}^*$ and w does not contain consecutive 1s$\}$

The following DFA accepts the language L



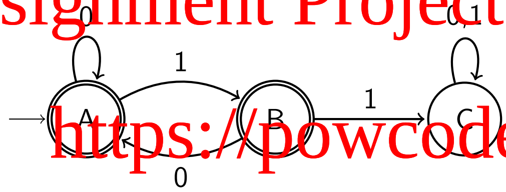
Is the string 101 in L?

- Start at A
- Follow transition 1 to reach B

String in a Language

Let $L = \{w|w \in \{0,1\}^*$ and w does not contain consecutive 1s$\}$

The following DFA accepts the language L



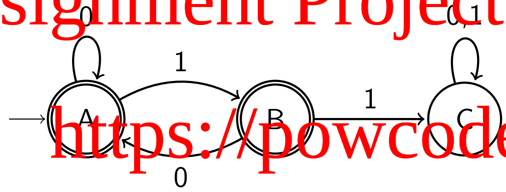
Is the string 101 in L?

- Start at A
- Follow transition 1 to reach B
- Follow transition 0 to reach A

String in a Language

Let $L = \{w | w \in \{0,1\}^*$ and w does not contain consecutive 1s$\}$

The following DFA accepts the language L



Is the string 101 in L?

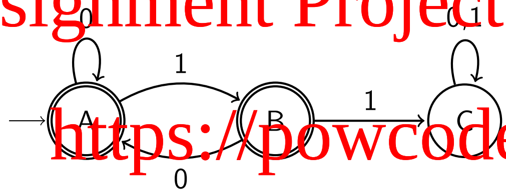
- Start at A

- Follow transition 1 to reach B
- Follow transition 0 to reach A
- Follow transition 1 to reach B

String in a Language

Let $L = \{w | w \in \{0,1\}^*$ and w does not contain consecutive 1s$\}$

The following DFA accepts the language L



Is the string 101 in L?

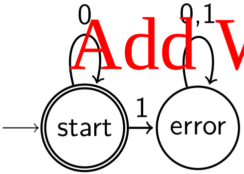
- Start at A
- Follow transition 1 to reach B
- Follow transition 0 to reach A
- Follow transition 1 to reach B
- The result is an accepting state, so 101 is in the language

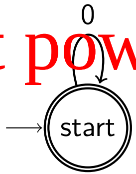
Error State(s)

An *error state* is any state from which it is impossible to reach an accept state. Sometimes we omit these from the diagram. All missing transitions point to an unseen error state.

You *must* write "error states not shown" if you omit them.

These DFA are equivalent and both have *two* states





(error states not shown)

???

With only one state, q_0, how many different finite automata can you devise?

Outline

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)
  - Recursion (fixed point, using it)
  - β-equivalence
  - Computational power
  - Functional Programming
- Automata Theory
  - Background concepts
  - DFA
  - Regular languages
  - **NFA**

Non deterministic Finite Automata (NFA)

DFA

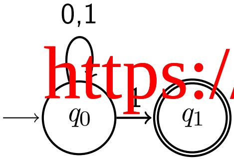
- has exactly one transition per input from each state
- no choice in the computation $\implies$ *deterministic*

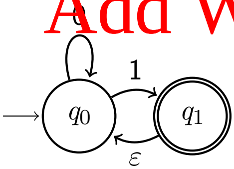
NFA

- can have any number of transitions per input from each state
- so some steps of the computation might be *non-deterministic*
- can also have ε-transitions, i.e. transitions which the automaton can follow without scanning any input

Non deterministic Finite Automata (NFA)

Two examples of NFA which accept the language:
$L(M) = \{w \mid w \in \{0,1\}^*$ and w ends in $1\}$





Review

- Lambda Calculus
  - Revision
  - Computational power
- Automata Theory
  - DFA
    - Formal definition
    - DFA diagrams
    - Using a DFA to do pattern matching
    - Language of a DFA
    - Building a DFA
  - Regular languages
  - Definition
  - How to prove that a language is regular
  - Introduction to NFA