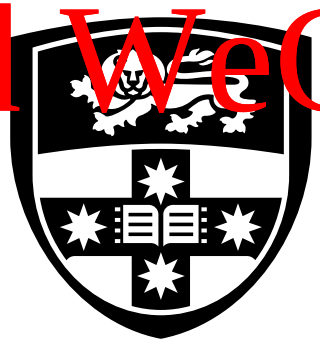
COMP2022: Formal Languages and Logic

2018, Semester 2, Week 4



THE UNIVERSITY OF SYDNEY

OUTLINE

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)

- Automata Theory
  - Background, concepts
  - DFA
  - Regular languages
  - NFA

Operators

- Application
  - Notation: $(A \cdot B)$

Operators

- Application
  - Notation: $(A \cdot B)$

- Abstraction
  - $(\lambda x.M)$
  - Variable x is abstracted in expressio
  - 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:

$$FV(x) = \{x\} \qquad \text{(con)}$$

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

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

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

Rewriting

- $M[x := N]$

d with

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

α-REDUCTION

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

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

β-reduction

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

- Note: the free occurrences of x i
occurrences which are bound to the

- Do α-reductions first if necessary

η-REDUCTION

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

- Reduces an abstraction directly

Naming

Naming conflicts:

- If a variable is free, but there is also λ with the same label
-
-

To fix them:

- Apply α-reduction repeatedly to chang
- 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 sa

i.e. some input X exists which, when appli
again.

Fixed Point Theorem

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

2. $\forall F\ F(YF)$

   - $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
   - 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:

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

If we ne

We ca

We can define it like this instead:

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

$$F = Y\ H$$

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

$$F = Y\ H$$

Eval

$$F\ a\ b\ c$$

$$= H\ (Y\ H)\ a\ b\ c \qquad H)$$

$$= \Big(\lambda fxyz.(\text{condition})(\text{base})($$

$$= \Big(\lambda xyz.(\text{condition})(\text{base})(\text{call(s) to } (Y\ H)) \quad a\ b\ c$$

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

-
-

Example: last week's tutorial questions

We wanted a function that would build a list $\{x_1 ... x_n\}$ from an expression like $(Fnx_1 ... x_n)$

Key ideas:

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

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

Example: last week's tutorial questions

$$
\begin{aligned}
&= F\ 3\ a\ b\ c \\
&= Y\ H\ NIL\ 3\ a\ b\ c \\
&= H \\
&= \big(\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Y\ H)\ NIL\ 3 \\
&= \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ED\ 3))\ a\ b\ c \\
&= \ldots = FALSE\ NIL\ (\lambda e.(Y\ H)\ (CO \qquad\qquad\qquad\qquad\qquad\qquad a\ b\ c \\
&= \ldots = (\lambda e.(Y\ H)\ (CONS\ e\ NIL))(\\
&= \ldots = (Y\ H)\ (CONS\ a\ NIL)\ (PRE \\
&= \ldots = (Y\ H)\ (CONS\ a\ NIL)\ 2\ b\ c \\
&= \ldots = (Y\ H)\ (CONS\ b\ (CONS\ a\ NIL))\ 1\ c \\
&= \ldots = (Y\ H)\ (CONS\ c\ (CONS\ b\ (CONS\ a\ NIL))\ 0 \\
&= \ldots = (CONS\ c\ (CONS\ b\ (CONS\ a\ NIL)))
\end{aligned}
$$

Example: last week's tutorial questions

Reversing a list:

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

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

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

Notation

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

We co

- $M \twoheadrightarrow_\beta N$: M β-reduces to N
- $M \rightarrow_\beta N$: M β-reduces to N (i
- $M =_\beta N$: M is β-convertable to

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

-

Nor

found by following the leftmost reduction

- $((\lambda x.xx)(\lambda x.xx))(\lambda ab.b)$ does not h
because the leftmost reduction loops infin

- $(\lambda ab.b)\big((\lambda x.xx)(\lambda x.xx)\big)$ 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
- them
- $(\lambda xy.x)ab$ and $(\lambda xy.x)ac$ both r
- Therefore, they are β-equivalen
- ... even though we have no way to reduce bet
- We've been using this property often, without stating it

β-equivalence

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

YF $_{\beta}$ YF

Turi

$\Theta \equiv \big(\lambda xy.y(xxy)\big)\big(\lambda xy.y(xxy)\big)$, which h

$\Theta F \twoheadrightarrow_{\beta} F(\Theta F)$

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 s
whic

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

Computational Power

Lambda calculus can compute all the computable (recursive) functions

- Proof is beyond the scope of this course
-

statements)
  - If a normal form exists, we can find it (Norm
  - 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 C
Turi

The proof is relatively simple. If we have time at the en
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 h

- They have equivalent computational po
- 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
- https://eduassistpro.github.i

  'crawl' over our data. Performing differ
  now just becomes arguments to these me
  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 function

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

OUTLINE

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)

- **Automata Theory**
  - Background, concepts
  - DFA
  - Regular languages
  - NFA

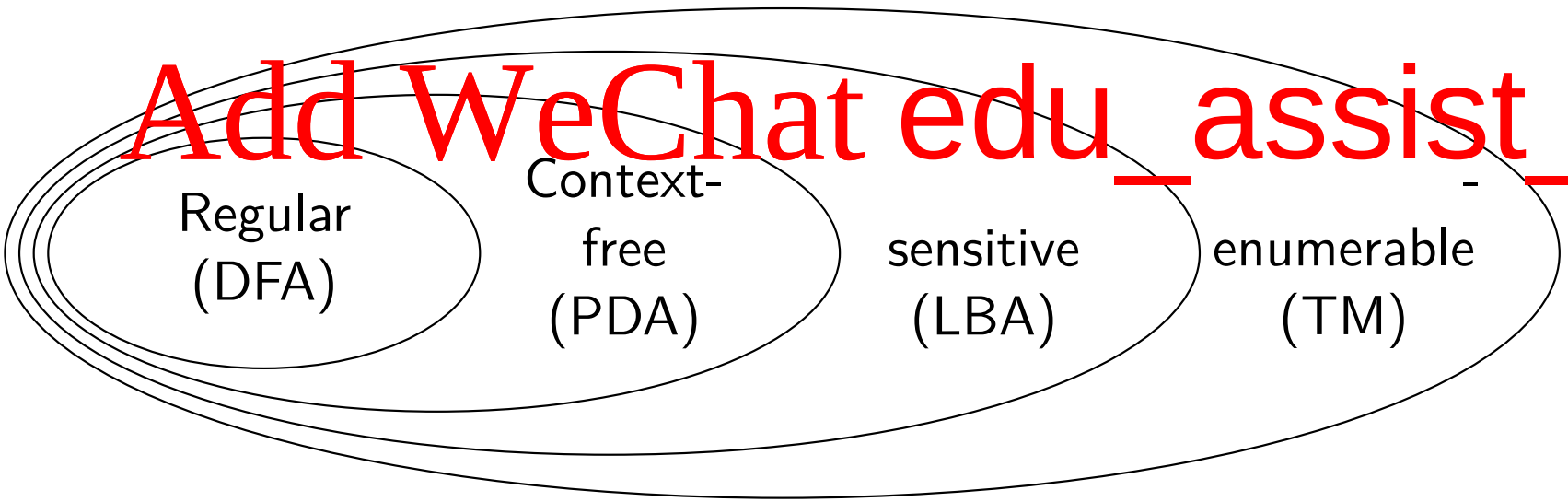
Alan Turing

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

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

Noam Chomsky

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



Alphabet

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

For ex

- All lower case letters: $\Sigma = \{a, b,$
- Alphanumerics: $\Sigma = \{a, \ldots z, A, \ldots$

- 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"
-
-
- ε (epsilon) denotes the *empty strin*
  - $|\varepsilon| = 0$
  - If $x = abacad\varepsilon$ then $|x| =$?
- $xy =$ the concatenation of two strings
- 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"
-
-
- ε (epsilon) denotes the *empty strin*
  - $|\varepsilon| = 0$
  - If $x = abacad\varepsilon$ then $|x| = 4$
- $xy =$ the concatenation of two strings
- x^n the string x repeated n times

Powers of an alphabet

Let

-
-
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$ (i.e. all stri

Languages

- A *language* L over alphabet Σ is a subset of Σ^* (i.e. $L \subseteq \Sigma^*$)
- Examples:

L	n	ollowed

L

0s and 1s:

- $L = \{\varepsilon, 01, 10, 001, 0101, 0\}$

- $\emptyset$ denotes the empty language
  - Beware: $\{\varepsilon\}$ is NOT $\emptyset$

The Membership problem

Problem:

Given a string w Σ^* and a language L over Σ, decide whether or not

Example:

Let $w = 100011$

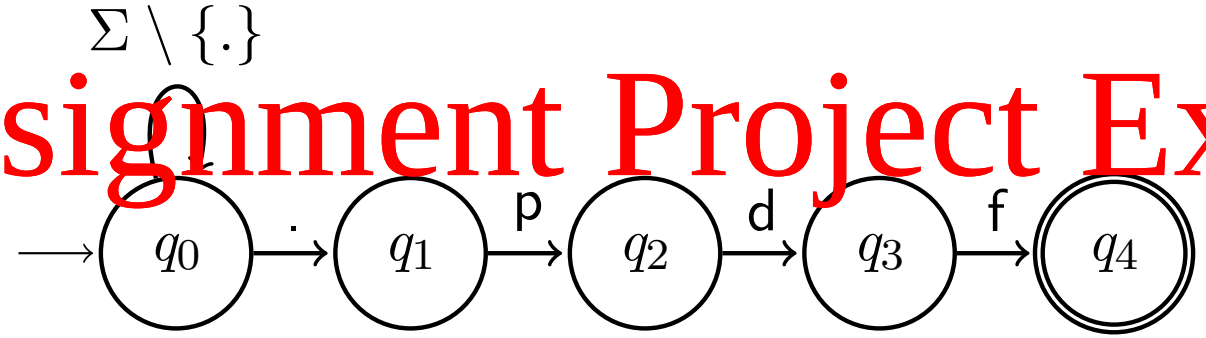
Does w belong to the language of strings with an even number of 0s and 1s?

OUTLINE

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)

- Automata Theory
  - Background, concepts
  - DFA
  - Regular languages
  - NFA

Introductory Example

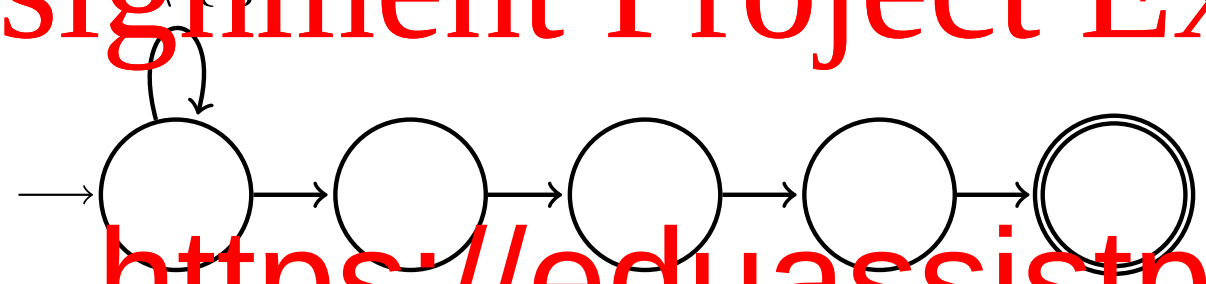


For infor

- Information is represented by the
- *Transition* rules (arrows) define state ch input

- *Start* state is denoted →
- *Accept* state(s) denoted by double circle
- The set of all possible input symbols is the *alphabet*, Σ

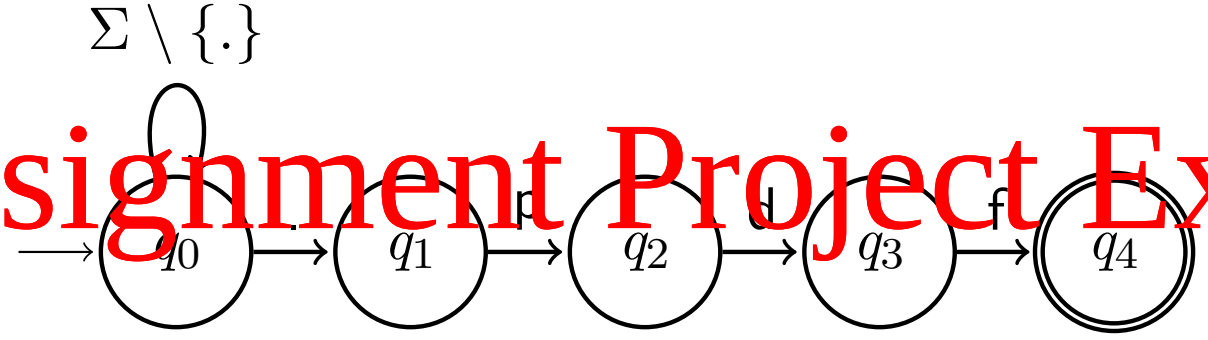
Introductory Example



Give

- Begin in the *start* state
- Follow one transition for each symbol in the i
- 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

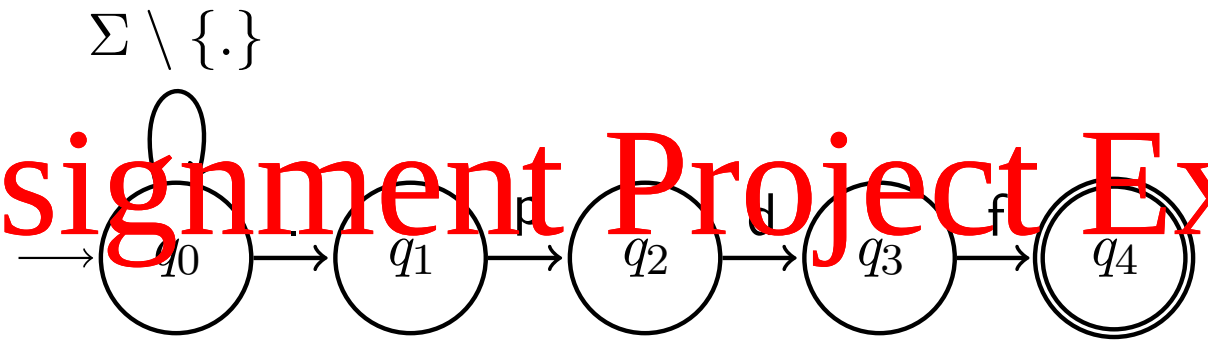
Introductory Example



Exa

-
-
- then q_2 for ‘p’, q_3 for ‘d’, q_4 for ‘f’
- after all input is scanned, we ended in an accepting state, therefore “example.pdf” is accepted

Introductory Example



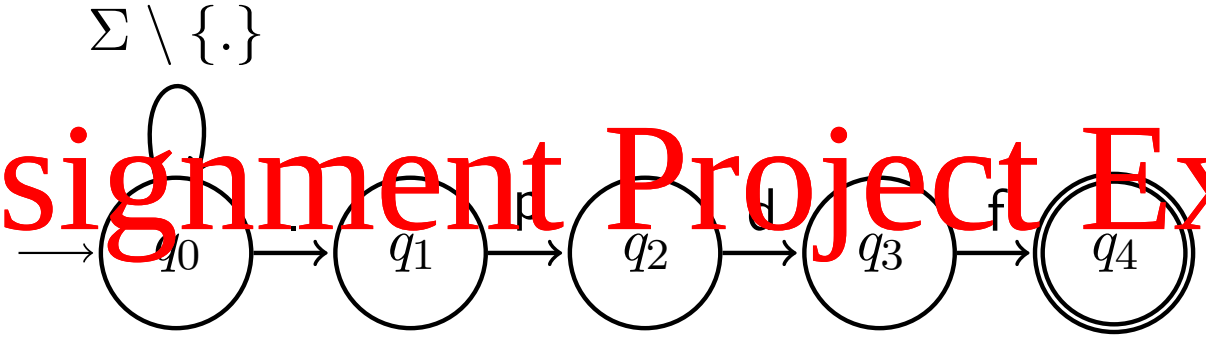
Exa

- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an
- therefore "example.pdf" is accepted

How about:

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

Introductory Example



Exa

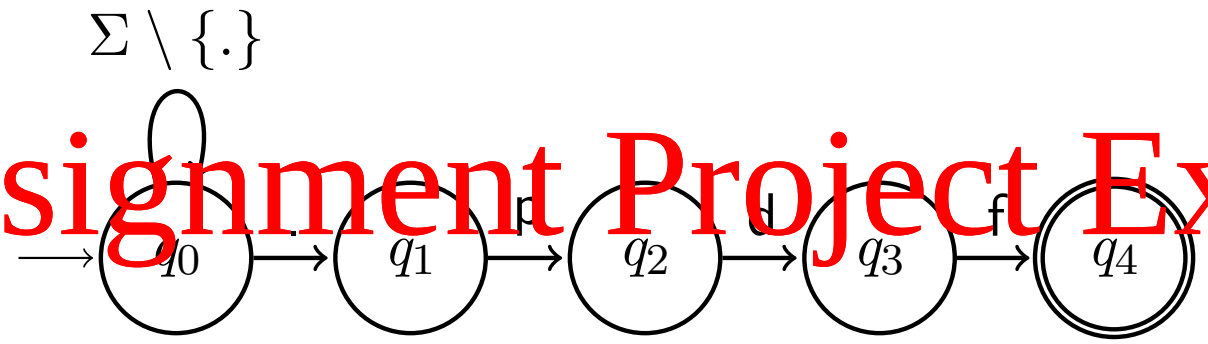
-
-
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an

therefore "example.pdf" is accepted

How about:

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

Introductory Example



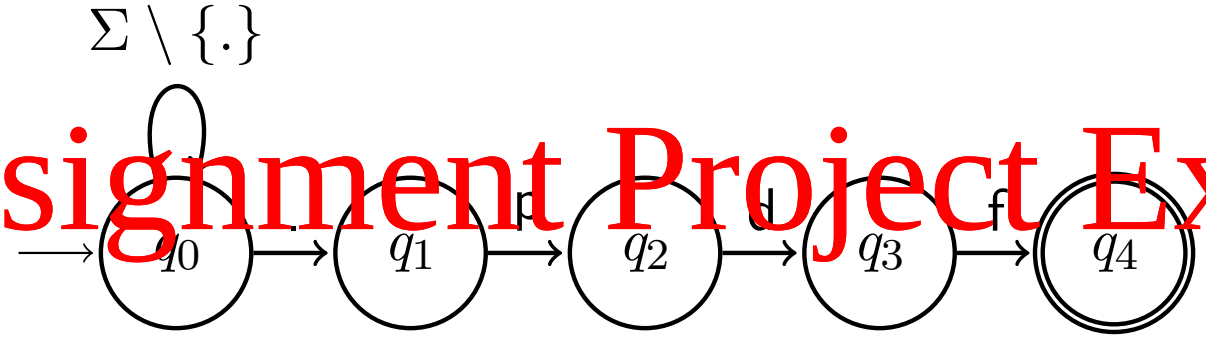
Exa

-
-
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an
  therefore "example.pdf" is accepted

How about:

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

Introductory Example



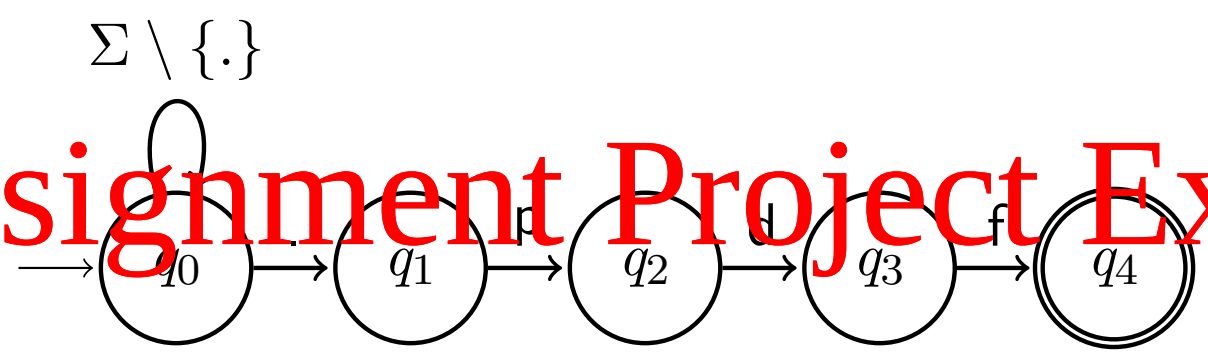
Exa

-
-
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an
  therefore "example.pdf" is accepted

How about:

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

Introductory Example



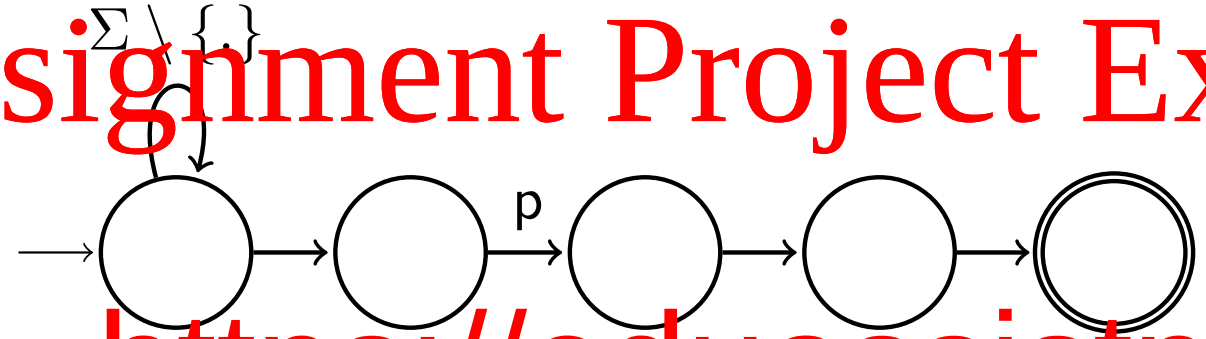
Exa

-
-
- then q_2 for 'p', q_3 for 'd', q_4 for 'f'
- after all input is scanned, we ended in an
  therefore "example.pdf" is accepted

How about:

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

Introductory Example

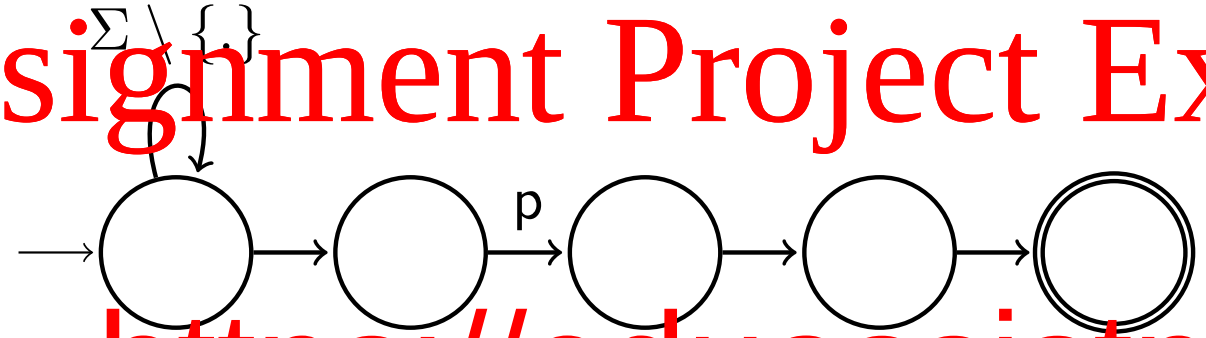


Reca

- ▶

What information is represented by state

Introductory Example



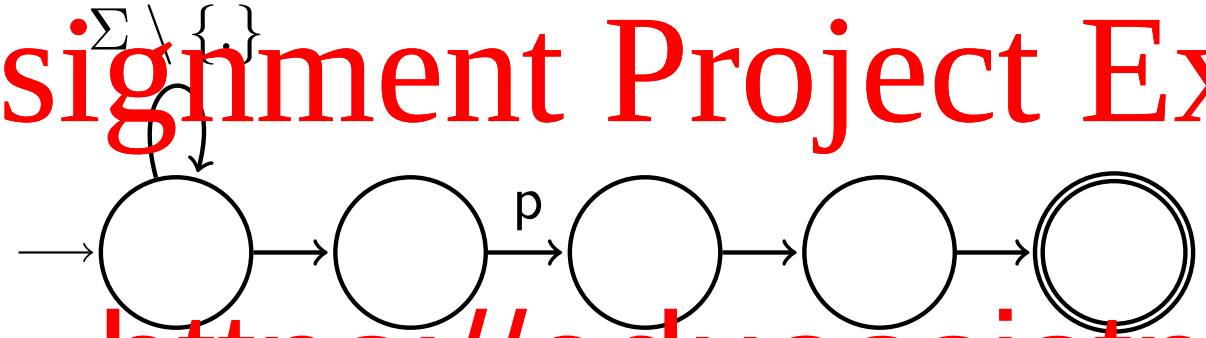
Reca

-

What information is represented by state

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

Introductory Example



Reca

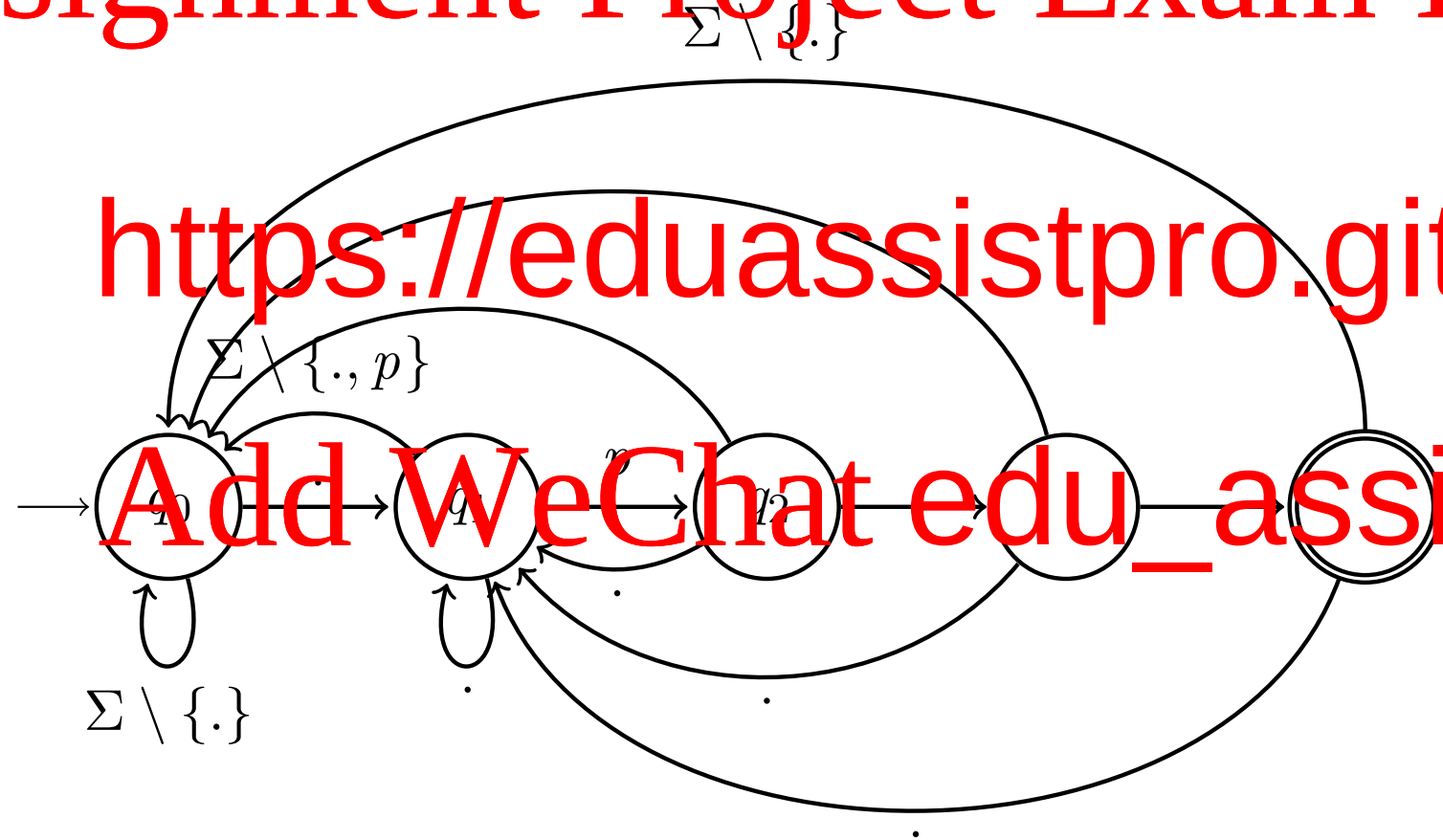
- ▶

What information is represented by state

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

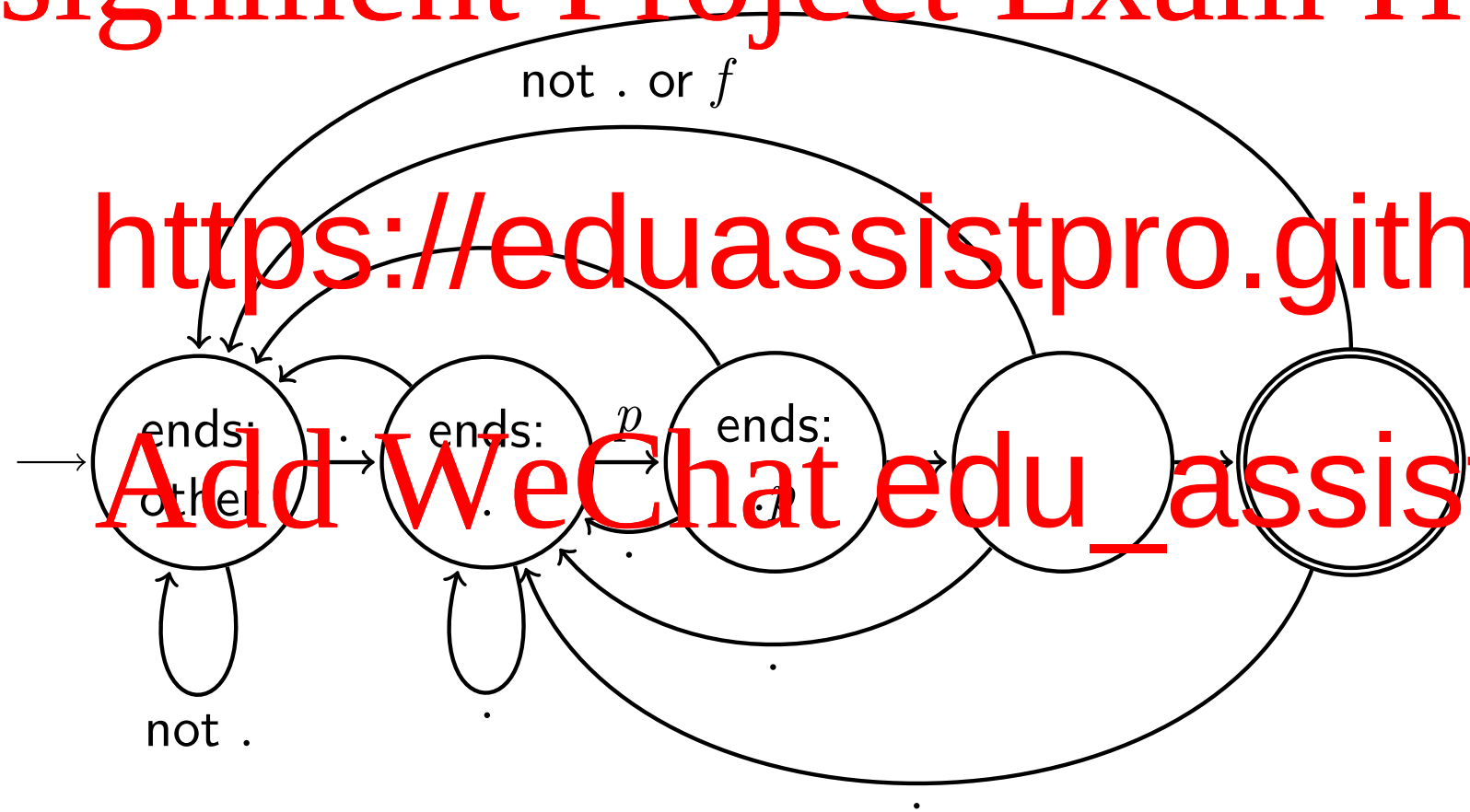
Our first valid DFA

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.
2.
3.
4. They have a behaviour given by t
   - Exactly one for each alphabet symbol fro
5. They have *accept state(s)*

Deterministic Finite Automata (DFA)

Formal definition:

A finit

1.
2.
3. $\delta : Q \times \Sigma \to Q$ is the *transition fu*
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.

	0	1

4. $q_0 \in Q$
5. $F = \{q_1\}$

Now we can draw M_1:

Draw a DFA from the definition

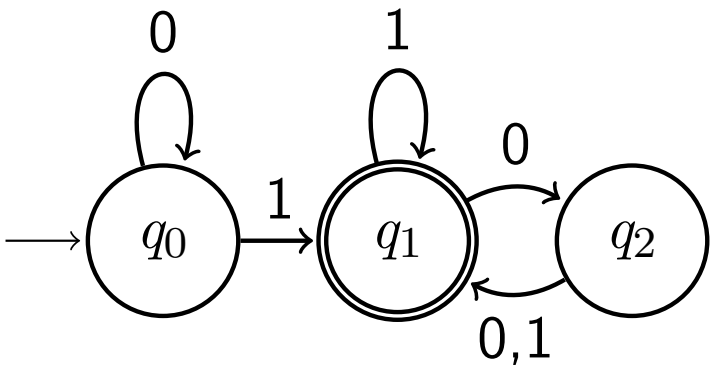
Let $M_1 = (Q, \Sigma, \delta, q_0, F)$ where:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3.

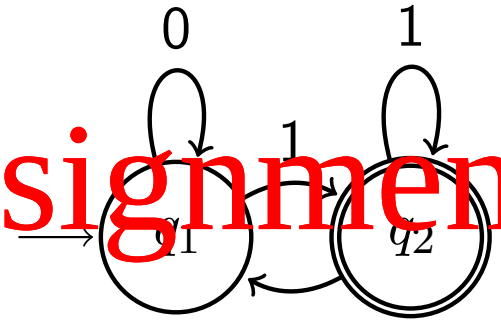
	0	1

4. $q_0 \in Q$
5. $F = \{q_1\}$

Now we can draw M_1:



Define a DFA from a diagram



We ca

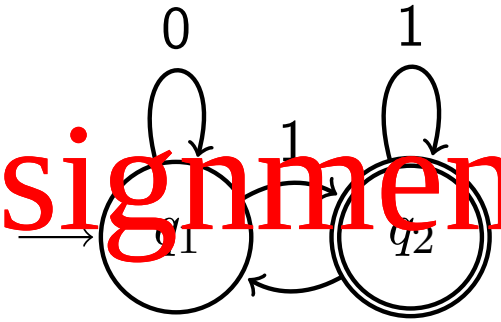
1.
2. $\Sigma =$

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



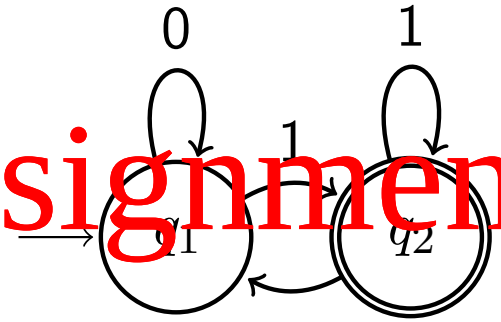
We ca

1.
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 ca

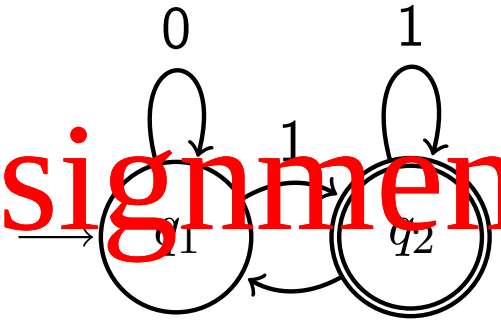
1.
2. $\Sigma = \{0, 1\}$

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 ca

1.
2. $\Sigma = \{0, 1\}$

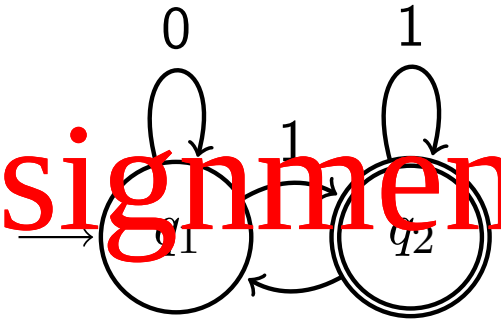
3. $\delta : Q \times \Sigma \to Q$ is given by

q		
q_2	1	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 ca

1.
2. $\Sigma = \{0, 1\}$

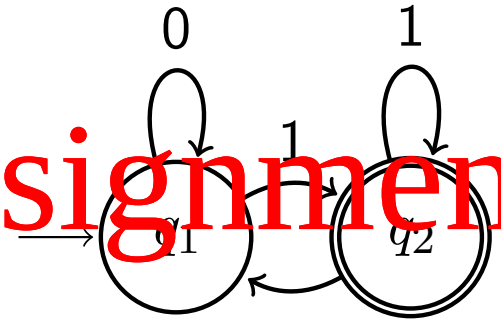
3. $\delta : Q \times \Sigma \to Q$ is given by

q		
q_2	1	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 ca

1.
2. $\Sigma = \{0, 1\}$

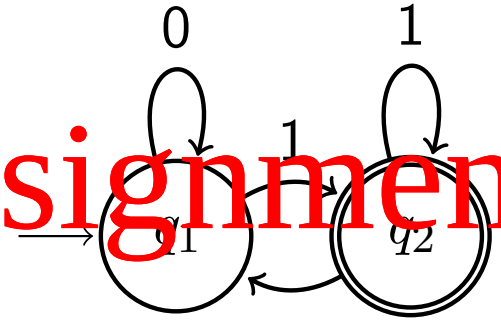
3. $\delta : Q \times \Sigma \to Q$ is given by

q		
q_2	1	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 ca

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

q		
q_2	1	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)

- 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

The $\ldots r_n$

from

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

Language of an automaton

- Automata of all kinds define languages
-
-

- We often denote the language recognised b
- $L(M)$ is the set of all strings labelling paths fr 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:

Prov } is a

regul

Regular Language

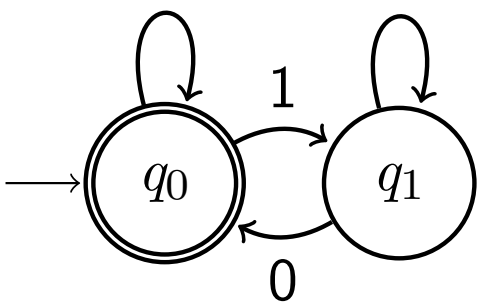
A language is *regular* if and only if there exists a finite automaton which recognises it.

Exercise:

Prov } is a regul

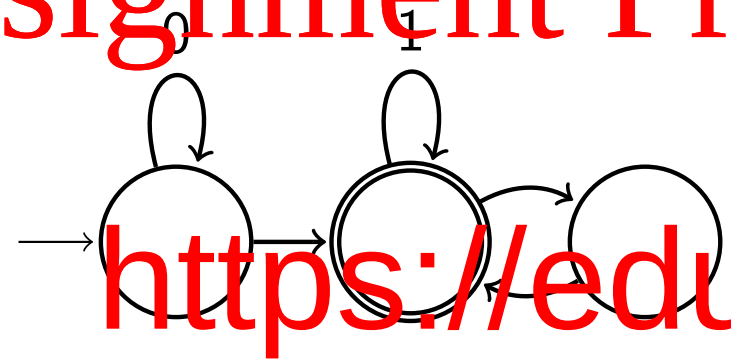
Solu

The following DFA recognises A, theref



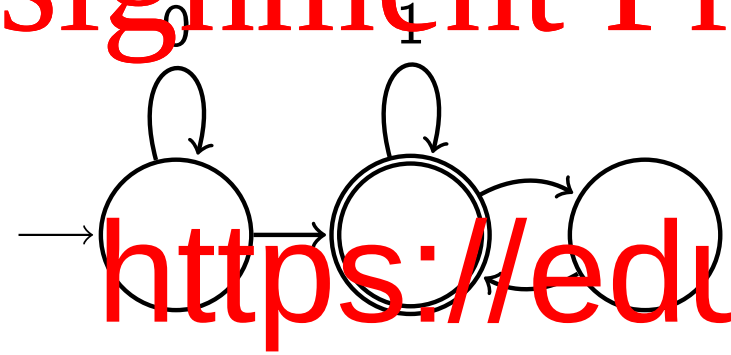
Examples of Regular Languages

What language is accepted by M_1?



Examples of Regular Languages

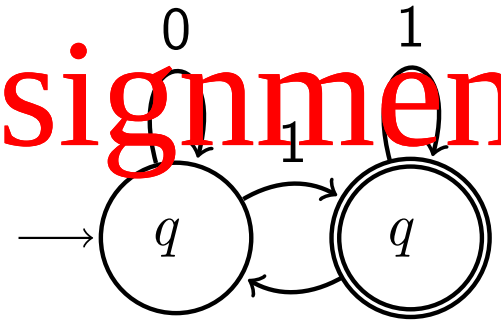
What language is accepted by M_1?



$L(M_1) = \{w | w$ contains at leas

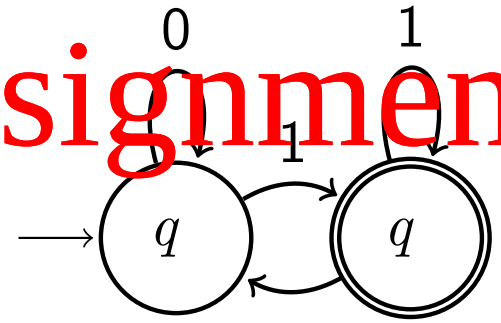
even number of 0s follo $\}$

Examples of Regular Languages



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

Examples of Regular Languages



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

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 ne

-
-

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

Designing finite automata

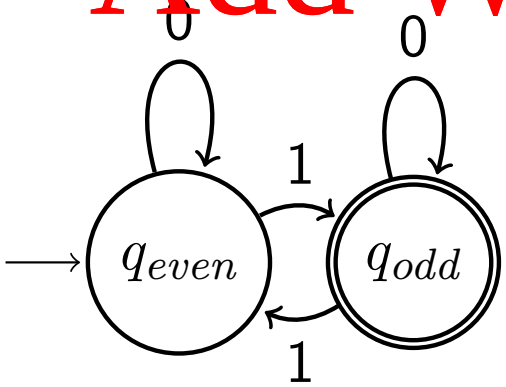
"Reader as Automaton" method: The *states* encode what you need to *remember* about the string as you are reading it.

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

We ne

-
-

So we will need exactly two states. Then it just remai 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:

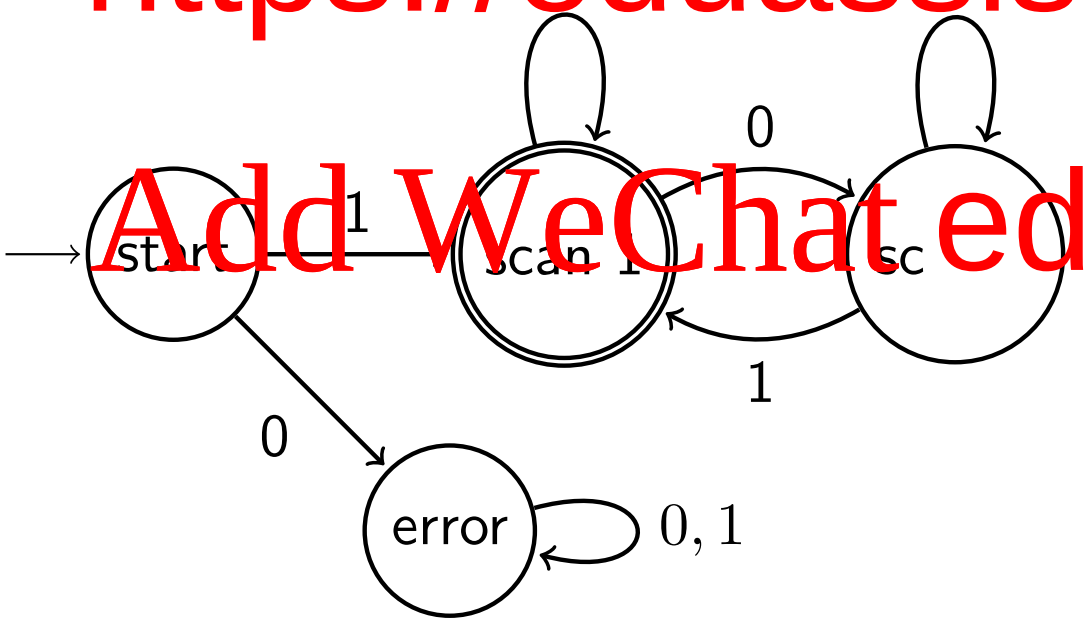
- Whether or not the string began with 1
-

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
-



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?

-

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?

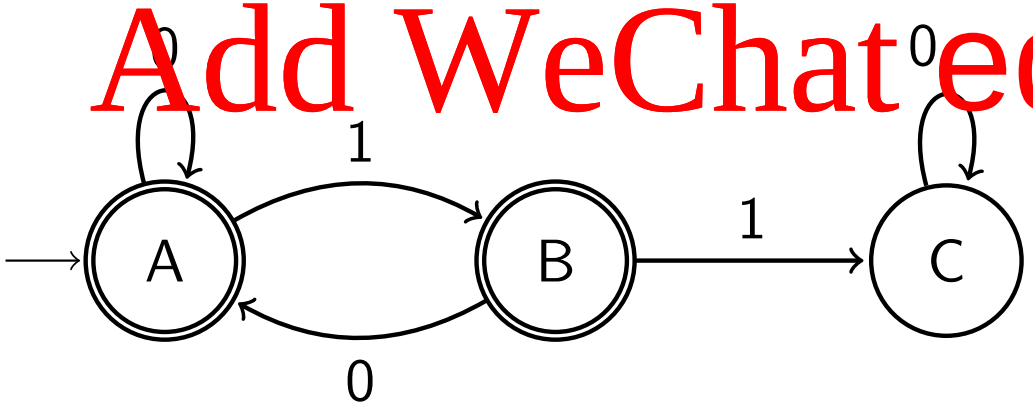
-
-
-

Example 3

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

What do we need to remember?

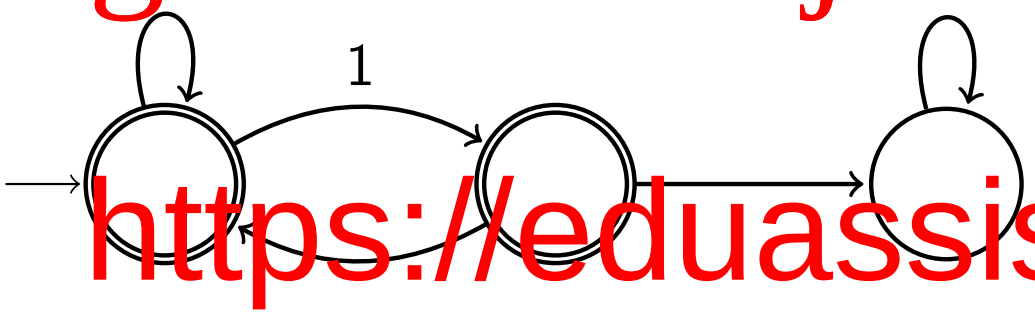
-
-
-



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

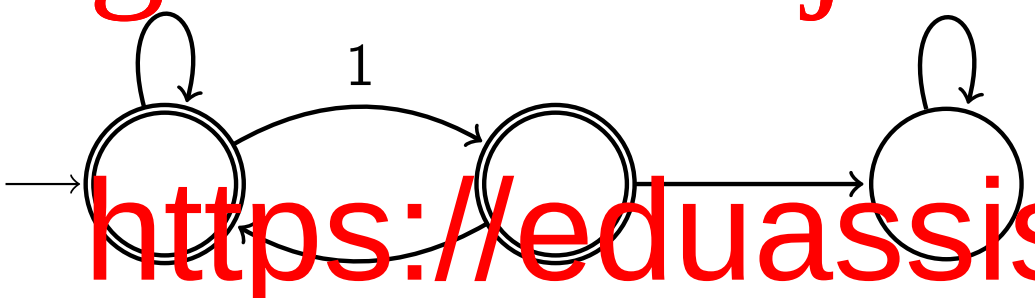


Is the string 101 in L?

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



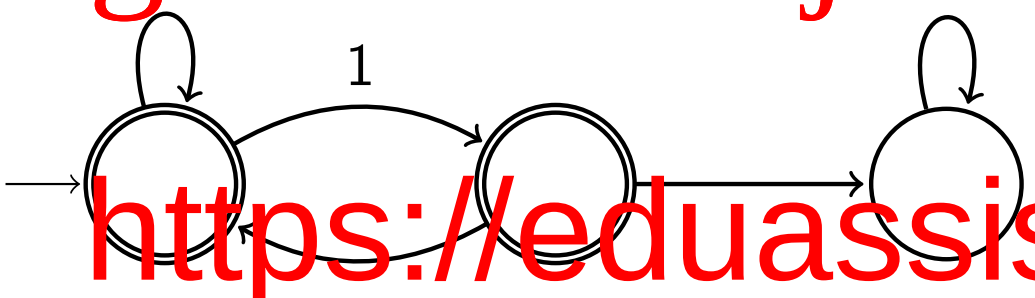
Is the string 101 in L?

- Start at A

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



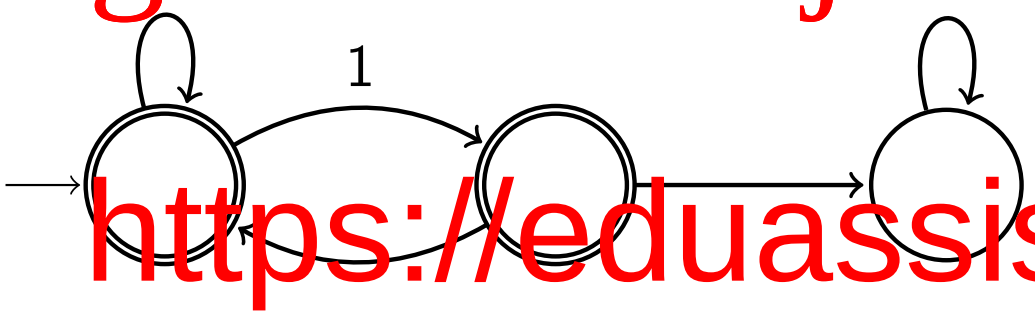
Is the string 101 in L?

- 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



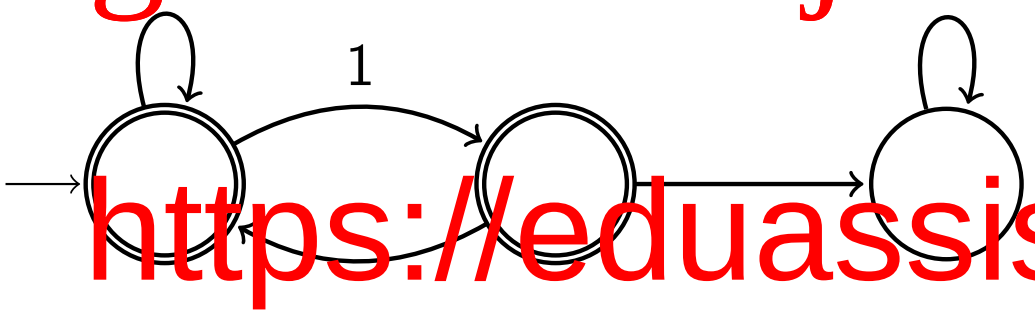
Is the string 101 in L?

- 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



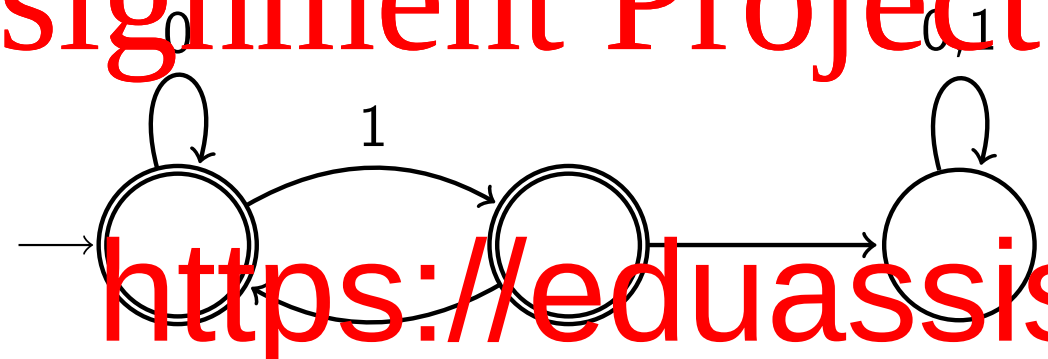
Is the string 101 in L?

- 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?

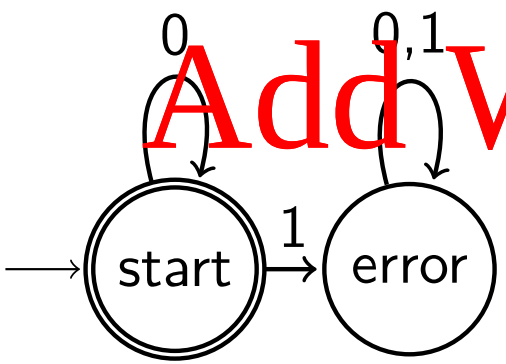
- Start at A
- Follow transition 1 to reach B
- Follow transition 0 to reach A
- Follow transition 1 to reach B
- The result is an accepting state, so 101 is in the language

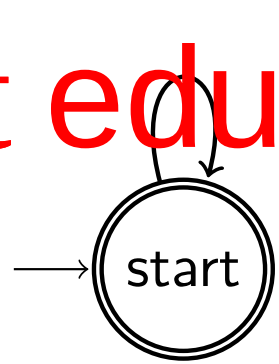
Error State(s)

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

Thes





(error states not shown)

???

OUTLINE

- Lambda Calculus
  - Concepts (Operators, FV, Reductions)

- Automata Theory
  - Background, concepts
  - DFA
  - Regular languages
  - **NFA**

Non deterministic Finite Automata (NFA)

DFA

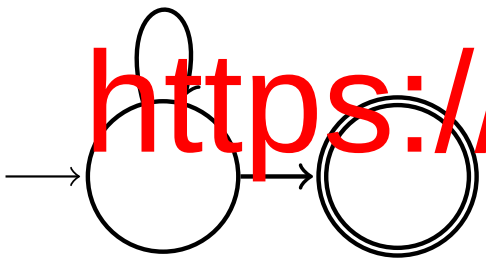
- has exactly one transition per input from each state
-

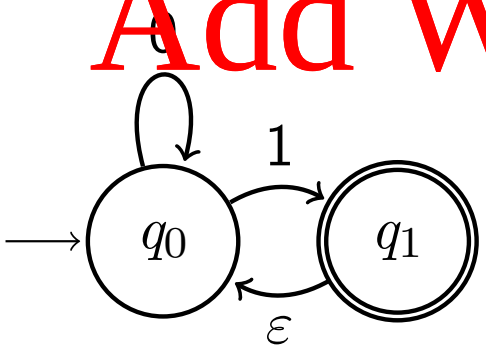
NFA

- can have any number of transitions per input f
- so some steps of the computation might be *stic*
- can also have ε-transitions, i.e. transi
  automaton can follow without scanning any input

Non deterministic Finite Automata (NFA)

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





Review

- Lambda Calculus
  - Revision
  - Computational power
-
- 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