Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

Notes:

- Goal: (briefly) introduce the topic [4, Part 1] of

  finite automata ≡ **Finite State Machines (FSMs)**

  in two steps, namely

  1. this lecture ⇒ what FSMs are ≃ the theory
  2. next lecture ⇒ design and implementation of hardware FSMs ≃ the practice

"Automata Theory in 10 minutes" (1)

Notes:

Definition

An **alphabet** is a non-empty set of **symbols**.

Definition

A **string** X wrt. some alphabet Σ is a sequence, of finite length, whose elements are members of Σ, i.e.,

$$X = \langle X_0, X_1, \ldots, X_{n-1} \rangle$$

for some n st. $X_i \in \Sigma$ for $0 \leq i < n$; if n is zero, we term X the **empty string** and denote it ϵ. It can be useful, and is common to write elements in in human-readable form termed a **string literal**: this basically just means writing them from right-to-left without any associated notation (e.g., brackets or commas).

Definition

A **language** is a set of strings.

Definition

A (formal) **grammar** is a tuple

$$G = (N, n, T, P)$$

including

1. N, a finite set of **non-terminal symbols** that includes a **start symbol** $n \in N$,
2. T, a finite set of **terminal symbols** (which is disjoint from N), and
3. P, a finite set of **production rules** each of the form

$$P_i : (N \cup T)^* \to (N \cup T)^*.$$

Notes:

- **Finite State Machines (FSMs)** are a model of **computation**.

Notes:

"Automata Theory in 10 minutes" (3)

- **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.

Notes:

"Automata Theory in 10 minutes" (3)

- **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string wrt. some alphabet A, one symbol at a time; each symbol induces a change in state.

Notes:

"Automata Theory in 10 minutes" (3)

- **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string wrt. some alphabet A, one symbol at a time; each symbol induces a change in state.
  - Once the input is exhausted, C halts; depending on the state it halts in, we say either
    1. C accepts (or recognises) the input string
    2. C rejects the input string

Notes:

"Automata Theory in 10 minutes" (3)

- **Finite State Machines (FSMs)** are a model of **computation**.
  - An FSM is an (idealised) computer C, which, at a given point in time, is in one of some finite set of states.
  - C accepts an input string wrt. some alphabet A, one symbol at a time; each symbol induces a change in state.
  - Once the input is exhausted, C halts; depending on the state it halts in, we say either
    1. C accepts (or recognises) the input string
    2. C rejects the input string
  - For a language L of all possible input strings which C *could* accept, we say

    $$C \text{ accepts (or recognises) } L \quad \equiv \quad L \text{ is the language of } C$$

    and use L to classify C ...

Notes:

"Automata Theory in 10 minutes" (4)

Definition

less powerful ⟷ more powerful

Machine	Combinatorial logic	Finite automaton	Push-down automaton	Linear-bounded automaton	Turing machine
Memory		0 stacks	1 stacks	2 stacks	2 stacks
Language		regular	context free	context sensitive	recursively enumerable
Grammar		regular ($X \to x$ or $X \to xY$)	context free ($X \to \gamma$)	context sensitive ($\alpha X \beta \to \alpha \gamma \beta$)	unrestricted ($\alpha \to \beta$)
Chomsky-Schützenberger hierarchy		type-3	type-2	type-1	type-0

Notes:

- In the description of grammars, the idea is (read from left-to-right) there are progressively less and less restrictions: X and Y both represent non-terminal symbols, x represents a terminal symbol, and α, β and γ are strings of either terminal *or* non-terminal symbols, all wrt. some alphabet and grammar. So, the far right-hand case has rules that are totally unrestricted, for example, whereas the far left-hand case has rules that must be of a particular form.

"Automata Theory in 10 minutes" (5)

Definition

A (deterministic) **Finite State Machine (FSM)** is a tuple

$$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$$

including

1. S, a finite set of **states** that includes a **start state** $s \in S$,
2. $A \subseteq S$, a finite set of **accepting states**,
3. an **input alphabet** Σ and an **output alphabet** Γ,
4. a **transition function**
$$\delta : S \times \Sigma \to S$$
and
5. an **output function**
$$\omega : S \to \Gamma$$
in the case of a **Moore FSM**, or
$$\omega : S \times \Sigma \to \Gamma$$
in the case of a **Mealy FSM**,

noting an **empty** input denoted ϵ allows a transition that can *always* occur.

Notes:

Question

Design an FSM that decides whether a binary sequence X has an odd number of 1 elements in it.

Notes:

"Automata Theory in 10 minutes" (6)

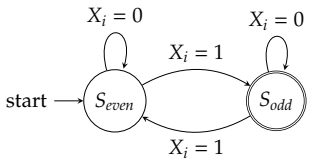
Question

Design an FSM that decides whether a binary sequence X has an odd number of 1 elements in it.

Algorithm (tabular)

Q	δ	
	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{even}	S_{odd}
S_{odd}	S_{odd}	S_{even}

Algorithm (diagram)



Notes:

© Daniel Page (<redacted>)
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

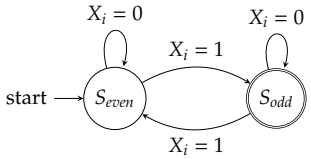
Question

Design an FSM that decides whether a binary sequence X has an odd number of 1 elements in it.

Algorithm (tabular)

	δ	
Q	Q'	
	$X_i = 0$	$X_i = 1$
S_{even}	S_{even}	S_{odd}
S_{odd}	S_{odd}	S_{even}

Algorithm (diagram)



- Example:
  1. For the input string $X = \langle 1, 0, 1, 1 \rangle$ the transitions are

$$\rightsquigarrow S_{even} \overset{X_0=1}{\rightsquigarrow} S_{odd} \overset{X_1=0}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{even} \overset{X_3=1}{\rightsquigarrow} S_{odd}$$

     so the input is accepted (i.e., has an odd number of 1 elements).
  2. For the input string $X = \langle 0, 1, 1, 0 \rangle$ the transitions are

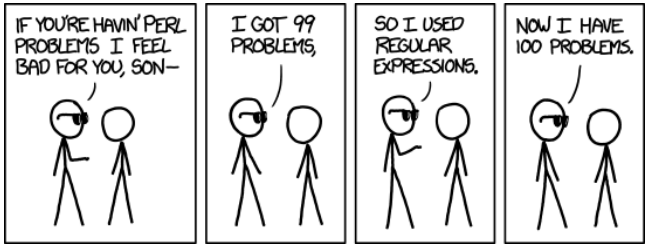
$$\rightsquigarrow S_{even} \overset{X_0=0}{\rightsquigarrow} S_{even} \overset{X_1=1}{\rightsquigarrow} S_{odd} \overset{X_2=1}{\rightsquigarrow} S_{even} \overset{X_3=0}{\rightsquigarrow} S_{even}$$

     so the input is rejected (i.e., has an even number of 1 elements).

Notes:

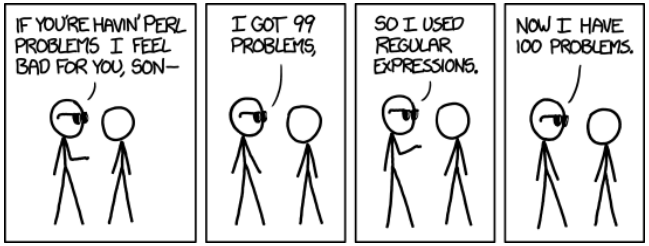
Some "real-world" motivation (1)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers



http://xkcd.com/1171/

Notes:



- Idea:

$$\textbf{arithmetic expression} \overset{\text{evaluate}}{\rightsquigarrow} \text{number}$$
$$\textbf{regular expression} \overset{\text{evaluate}}{\rightsquigarrow} \text{language}$$

st. you can think of a regular expression (or regex) as

1. a pattern used to describe or generate a language, *or*
2. a pattern used to identify (i.e., match) members of a language.

---

http://xkcd.com/1171/

Notes:

Question

We say X is a **regular expression** if it is

1. a symbol in the alphabet, i.e., $\{x\}$ for $x \in \Sigma$,
2. the union of regular expressions X and Y st.

$$X \cup Y = \{x \mid x \in X \vee x \in Y\},$$

3. the concatination of regular expressions X and Y st.

$$X \parallel Y = \{\langle x, y \rangle \mid x \in X \wedge y \in Y\},$$

or

4. the **Kleene star** of regular expression X st.

$$X^* = \{\langle x_0, x_1, \ldots, x_{n-1} \rangle \mid n \geq 0, x_i \in X\}.$$

allowing for various short-hands, e.g.,

$$\begin{array}{lcl} x & \equiv & \{x\} \\ xy & \equiv & \{x\} \parallel \{y\} \\ X^+ & \equiv & X \parallel X^* \end{array}$$

Notes:

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$0^*10^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ \text{a single 1} \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

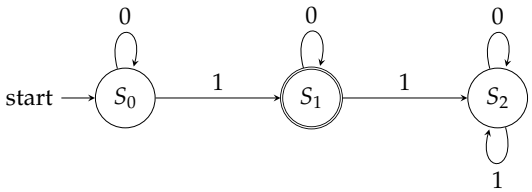
Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$0^*10^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ \text{a single 1} \end{array} \right\}$$

which can be realised using



© Daniel Page ⟨...⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

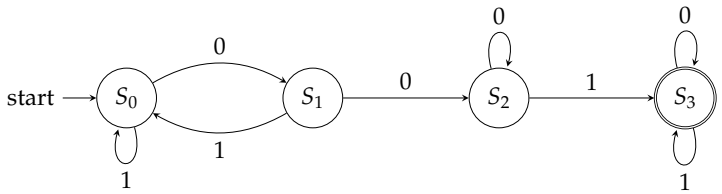
Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$\Sigma^* 001 \Sigma^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using



Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$\Sigma^* 001 \Sigma^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string containing} \\ 001 \text{ as a sub-string} \end{array} \right\}$$

which can be realised using

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

© Daniel Page ⟨cs@bristol.ac.uk⟩
Intro. to Computer Architecture
git # 3b6f641 @ 2018-01-09
University of BRISTOL

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$(\Sigma\Sigma)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

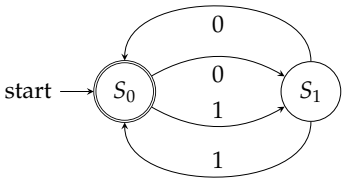
Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$(\Sigma\Sigma)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string} \\ \text{of even length} \end{array} \right\}$$

which can be realised using



© Daniel Page ⟨...⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$1^*(01^+)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string in which} \\ \text{every 0 is followed by at least one 1} \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

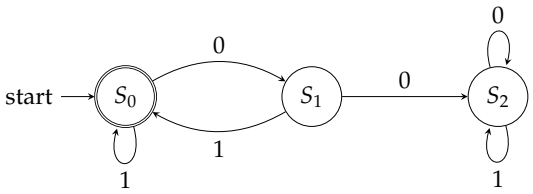
Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{0, 1\}$, then

$$1^*(01^+)^* \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a string in which} \\ \text{every 0 is followed by at least one 1} \end{array} \right\}$$

which can be realised using



© Daniel Page ⟨...⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point.
  In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

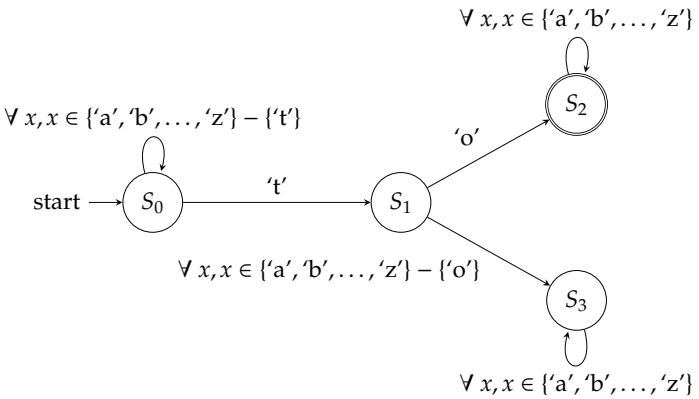
Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

which can be realised using



© Daniel Page ⟨...⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

which can be realised using

```
1  forall lines X read from stdin do
3  |  Q ← s
4  |  for i = 0 upto n − 1 do
5  |  |  Q ← δ(Q, X_i)
6  |  end
7  |  if Q ∈ A then
8  |  |  print line X to stdout
10 |  end
11 end
```


Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some "real-world" motivation (3)

Example #1: regular expressions + `grep` ⇝ FSMs as recognisers

- Example [4, Example 1.53]: if $\Sigma = \{\text{'a'}, \text{'b'}, \ldots, \text{'z'}\}$, then

$$\texttt{grep -E '.*to+.*'} \equiv \left\{ s \;\middle|\; \begin{array}{c} s \text{ is a line read from } \texttt{stdin} \text{ containing} \\ \text{a 't' followed by at least one 'o' character} \end{array} \right\}$$

which can be realised using

```
1  void grep() {
2    char X[ 1024 ];
3
4    while( NULL != fgets( X, 1024, stdin ) ) {
5      int n = strlen( X ), Q = start;
6
7      if( X[ n - 1 ] == '\n' ) {
8        X[ n - 1 ] = '\0'; n--;
9      }
10
11     for( int i = 0; i < n; i++ ) {
12       Q = delta[ Q ][ X[ i ] ];
13     }
14
15     if( accept[ Q ] ) {
16       fprintf( stdout, "%s\n", X );
17     }
18   }
19 }
```

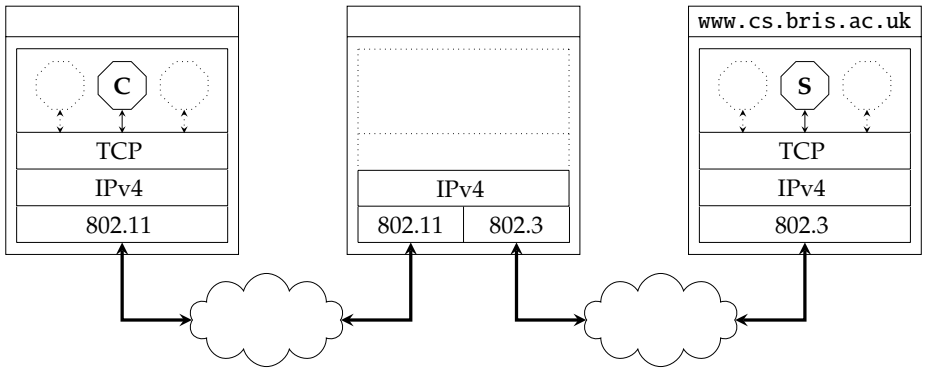
© Daniel Page (<redacted>)
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

- The example implementation in the `grep` example is a little simplistic, so the behaviour (and hence output) will in fact differ vs. *real* `grep`. The technical reason for this is that we have ignored (or disallowed) backtracking: if the FSM accepts a 't' followed by something other than 'o' then it fails, but obviously this ignores the possibility that there might be *another* 't' later which *is* followed by 'o'. So the description might be better written as "s is a line read from `stdin`, where a) there is at least one 't' character, and b) the first 't' character is followed by at least one 'o' character". We could try to capture that using a more complex transition function, but then the example becomes overly complex: although the details is important in a general sense, the underlying idea is more important at this point. In *even* more detail, the difference here relates to a difference between so-called Deterministic Finite Automatons (DFAs) and Non-deterministic Finite Automatons (NFAs): we are dealing with and assume the former, whereas `grep` makes use of the latter (see, e.g., [4, Section 1.2]).

Some “real-world” motivation (4)

Example #2: networked communication via TCP ⇝ FSMs as controllers

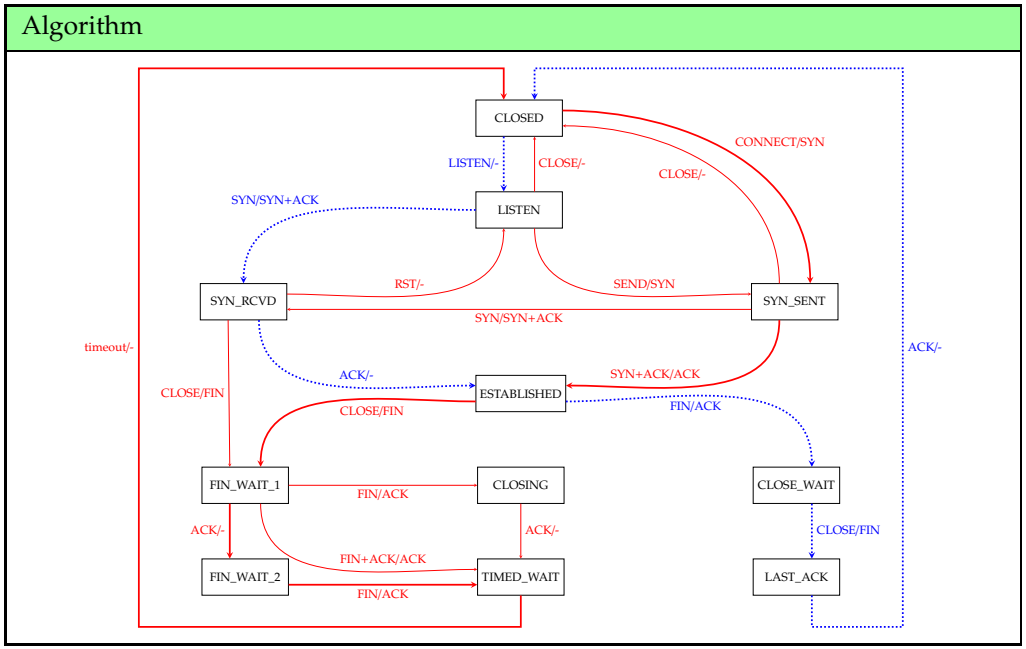


Notes:

Some “real-world” motivation (5)

Example #2: networked communication via TCP ⇝ FSMs as controllers

Algorithm



Notes:

Some "real-world" motivation (6)

Example #3: typical video game "loop" ⇝ FSMs as systems

Algorithm

```
1 reset the game state
2 while ¬ game over do
3 |   read control pad (e.g., check if button pressed)
4 |   update game state (e.g., move player)
5 |   produce graphics and/or sound
6 end
```

Notes:

Some "real-world" motivation (6)

Example #3: typical video game "loop" ⇝ FSMs as systems

Algorithm

```
1 Q ← s
2 while Q ∉ A do
3 |   X_i ← control pad
4 |   Q ← δ(Q, X_i)
5 |   {graphics, sound} ← ω(Q)
6 end
```

Notes:

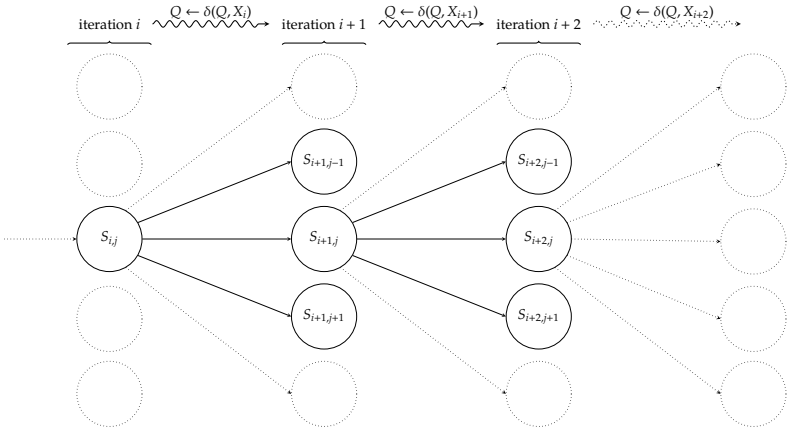
Some "real-world" motivation (7)

Example #3: typical video game "loop" $\rightsquigarrow$ FSMs as systems

- Idea:

$$\begin{array}{rcl} \text{iterations of game loop} & \rightsquigarrow & \text{game tree} \\ & \simeq & \text{state space} \end{array}$$

i.e.,



which is most obvious wrt. turn-based games (e.g., chess).

Notes:

Some "real-world" motivation (8)

- Take away points:
  - FSMs have a variety of practical applications, e.g.,
    1. recognisers,
    2. controllers,
    3. ...
    4. *specifications*: like an algorithm, but more easily able to cater for asynchronous events

    the theory underlying which is expanded upon elsewhere.
  - Even if the state space is *huge*, it may still be finite; δ can a **partial function** st. it needn't be defined for all inputs.

Notes:

Continued in next lecture ...

Notes:

Additional Reading

- *Wikipedia: Finite State Machine (FSM)*. . URL: http://en.wikipedia.org/wiki/Finite-state_machine.
- D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.
- M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006.

© Daniel Page ⟨csdsp@bristol.ac.uk⟩
Intro. to Computer Architecture git # 3b6f641 @ 2018-01-09 University of BRISTOL

Notes:

References

[1] *Wikipedia: Finite State Machine (FSM)*. URL: http://en.wikipedia.org/wiki/Finite-state_machine (see p. 75).

[2] D. Page. "Chapter 2: Basics of digital logic". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 75).

[3] M. Sipster. "Chapter 1: Regular languages". In: *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see p. 75).

[4] M. Sipster. *Introduction to the Theory of Computation*. 2nd ed. Thomson Course Technology, 2006 (see pp. 5, 37–60).

Notes: