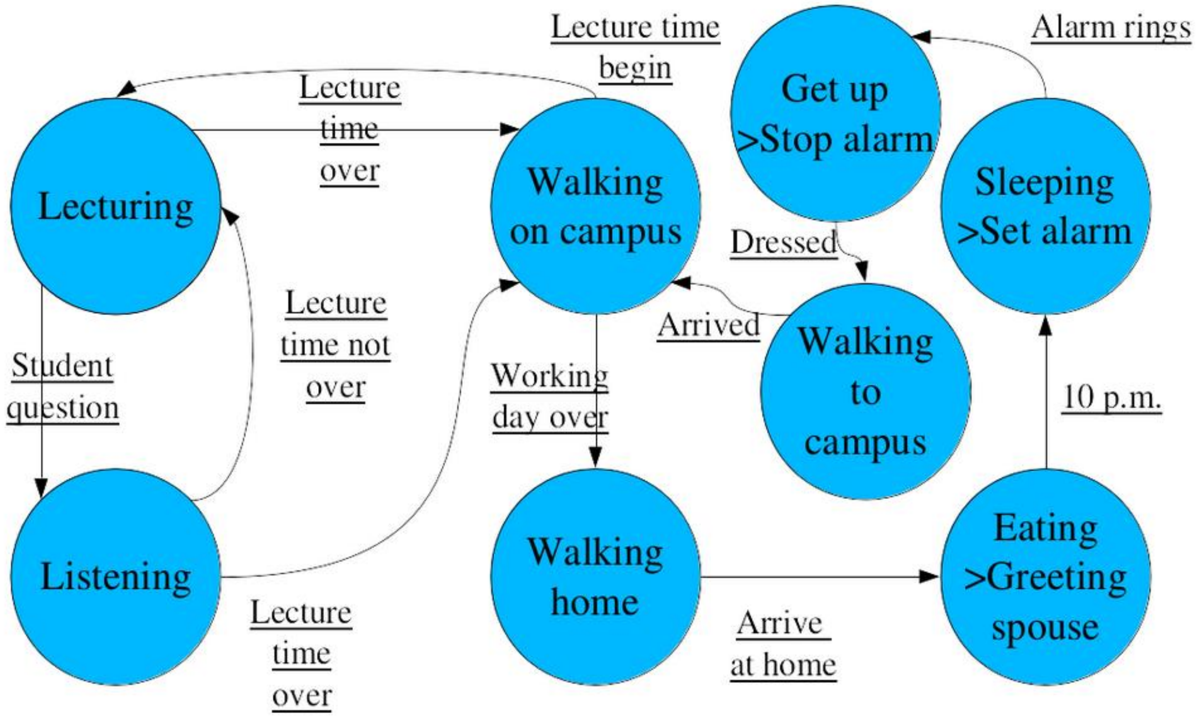
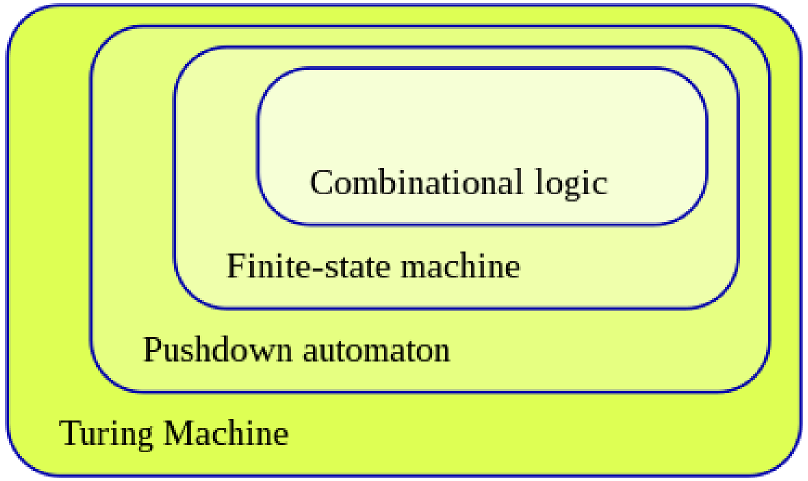
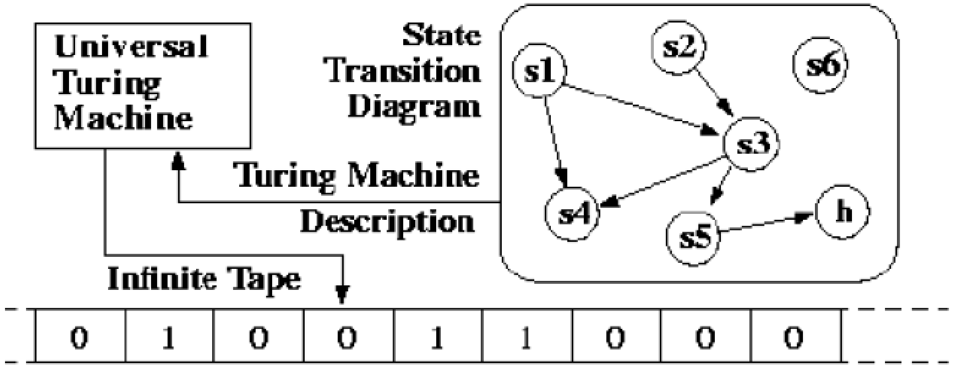
8. Automata and Turing machines







Finite state machine

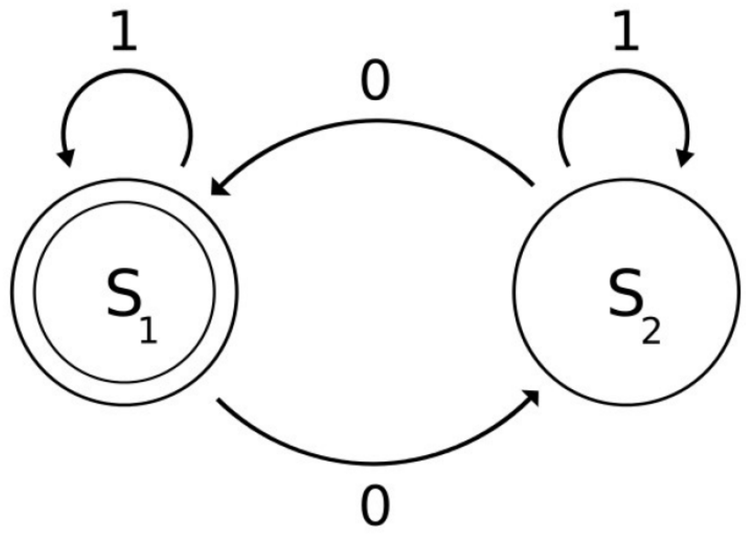
- We already encountered state machines when we discussed the grammars and lexing phase in compiling; let’s dive a bit deeper there now
- A *finite state machine* is a way to model a task, language or data as a group of states and transitions between them
  - An *automaton* of one kind
  - Commonly presented in the form of a state diagram
  - Automaton processes the input one symbol at a time
  - Initial state(s) are represented by input arrows
  - States are circled, transitions are shown as arrows from one state to another
  - Accept (end) states are presented by double circles (or output arrows)

Example of a simple finite state machine
p = start state
a = transition $(p \times a \rightarrow q)$
q = accept state

State transition table

- Transitions between states can be represented by a *state transition table*
- Current states as rows, input symbols as columns
  - Table cell value tells the next state
- This notation has some weaknesses, though:
  - Initial state has not been marked in any way
  - No info on which states are accept states
- Needs improvement!

State Diagram



State Transition Table

State \ Input	1	0
S_1	S_1	S_2
S_2	S_2	S_1

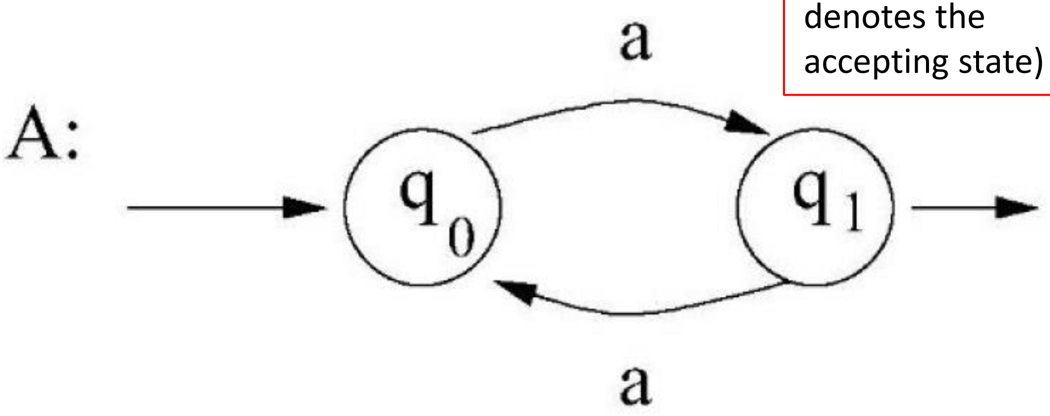
State transition table (improved version)

- Automaton that accepts an input which consists of an odd number of a's
- Mathematically speaking:

$$\{aa^{2n} \mid n \geq 0\}$$

- Now the state transition table holds all information that is needed:
  - Start state is marked with a rightwards arrow
  - Accept state(s) are marked with a leftwards arrow (in some notations, also an asterisk (*) is used)

(Notice: output arrow from q_1 denotes the accepting state)



	a
$\rightarrow q_0$	q_1
$\leftarrow q_1$	q_0

Types of automata

- An automaton can be *deterministic* or *non-deterministic*:
  - Deterministic = the state transitions are unambiguous – there is only one possible transition for each symbol
  - Non-deterministic = more than one possible transition in some state for at least one symbol
- Non-deterministic automaton must make guesses, so it needs to have an “escape route” in case it makes a bad guess
- An automaton can also be *finite* or *infinite*:
  - Finite = there is a finite amount of possible states
  - Infinite = amount of possible states is not limited
- Usually finite automata work with finite input strings; finite automata that can handle infinite inputs are called *ω-automata*

Deterministic finite-state automaton (DFA)

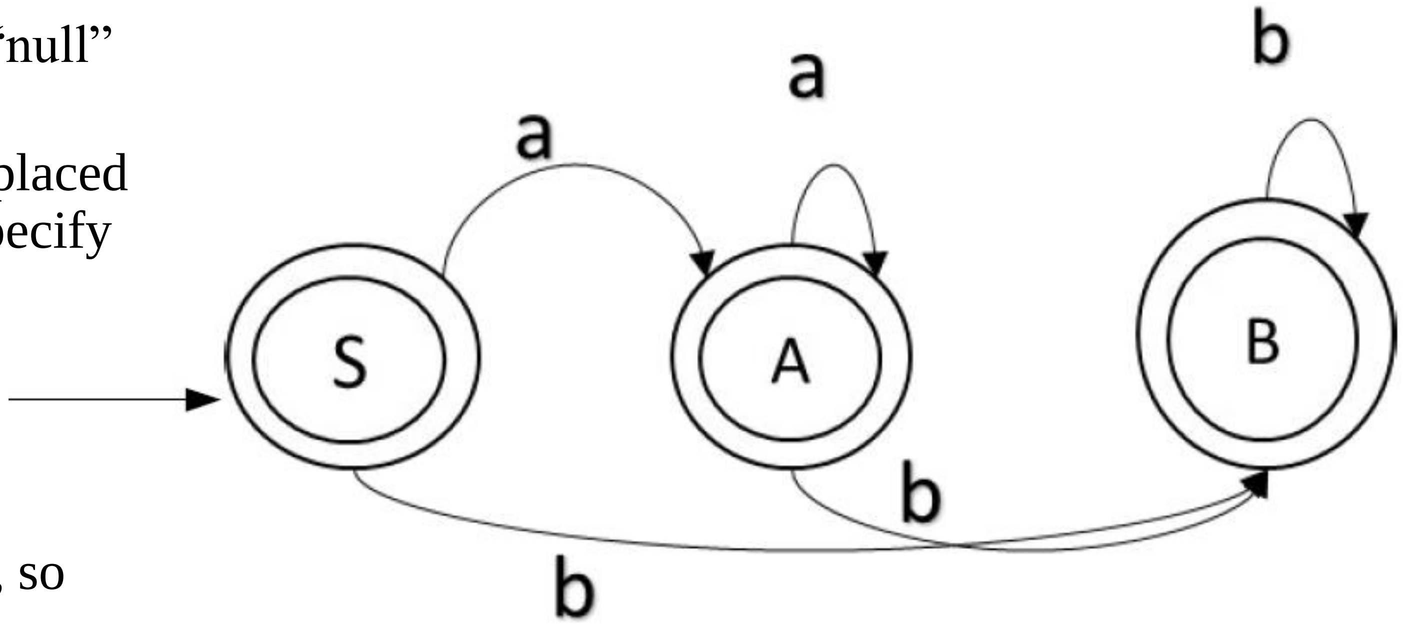
- A *deterministic finite-state automaton* (DFA) is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:
  - Q = a finite group of states
  - Σ = the alphabet of the language
  - δ = a transition function that specifies the transitions $Q \times \Sigma \rightarrow Q$ (or alternatively: $\delta[Q], \Sigma = [Q]$)
  - q_0 = initial state (Note: $q_0 \in Q$, naturally)
  - F = group of accept states (Note: F $\subseteq Q$, naturally)
- DFA accepts an input string if reading it leads from initial state to accept state
  - If reading the string doesn't end in an accepting state, the string is not accepted
- If there is no transition for some character of the string, the input is disqualified
  - Note! A different situation than ”not accepted”!
  - Results in an error and termination of the process
  - How can the automaton recover from the error?

Grammar definition using an automaton

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \lambda$$
$$B \rightarrow Bb \mid \lambda$$

- An automaton can be used to define a grammar of a language
- Simple example:
  - States: $Q = \{S, A, B\}$
  - Alphabet: $\Sigma = \{a, b, \lambda\}$ (λ is a “null” symbol)
  - Productions tell what can be replaced by which, so the productions specify the transitions

$$\delta = S \times a \rightarrow A, S \times b \rightarrow B,$$
$$A \times a \rightarrow A,\ \ A \times b \rightarrow B,\ \ B \times b \rightarrow B$$



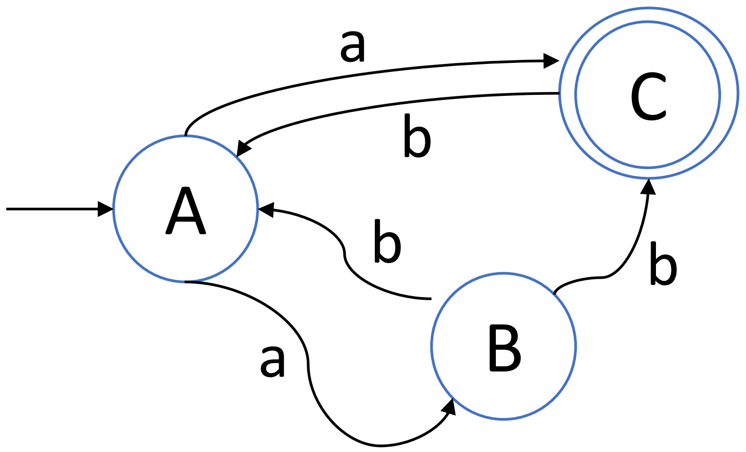
  - Initial state $q_0 = S$
  - Here all states are accept states, so $F = Q = \{S, A, B\}$

From NFA to DFA

- It is common that a problem is, in many cases, easier to approach by constructing a non-deterministic finite automaton (NFA)
- An NFA is problematic to write into a program, though, because the automaton should be able to recover from bad guesses
- We can convert all NFAs to DFAs using subset construction
- A k-state NFA can always be converted to a (max.) 2^k-state DFA
  - In many cases, the DFA will simplify and have less states
- Conversion in a nutshell:
  - Create a transition table for the NFA
  - If some transition has multiple state options, consider this state combination a new state
  - Create a new transition table for the DFA (derive the transitions of new states)
- The resulting DFA can be simplified by deleting unreachable states
- Examples here: https://www.javatpoint.com/automata-conversion-from-nfa-to-dfa

NFA to DFA: Example 1

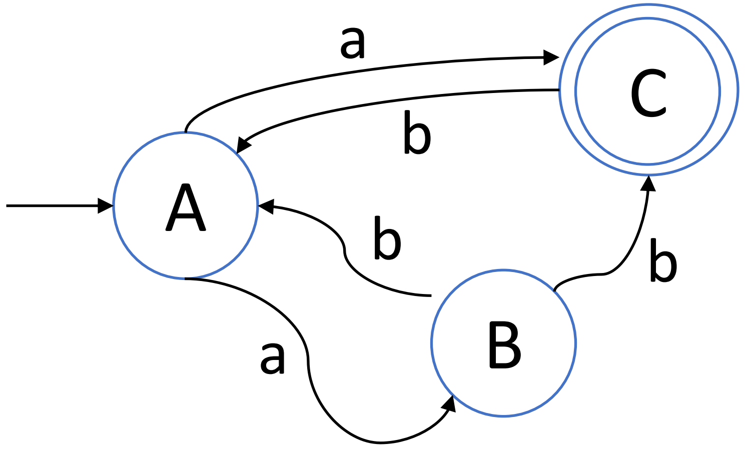
- Convert the following NFA to a DFA.



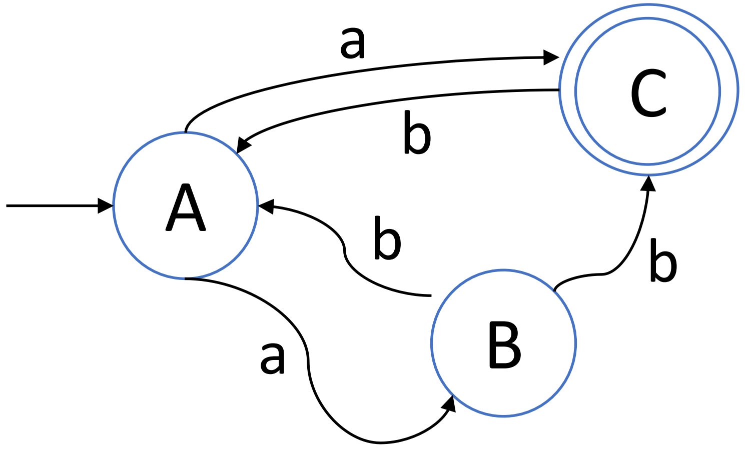
NFA to DFA: Example 1

- Convert the following NFA to a DFA.
- Transition table for the NFA:

	a	b
→A	B,C	-
B	-	A,C
*C	-	A



NFA to DFA: Example 1



- Convert the following NFA to a DFA.
- Transition table for the NFA:

	a	b
→A	B,C	-
B	-	A,C
*C	-	A

- Transitions for new states:

$\delta'[B,C],a = \delta[B],a \cup \delta[C],a = \emptyset \cup \emptyset = \emptyset$

$\delta'[B,C],b = \delta[B],b \cup \delta[C],b = [A,C] \cup [A] = [A,C]$

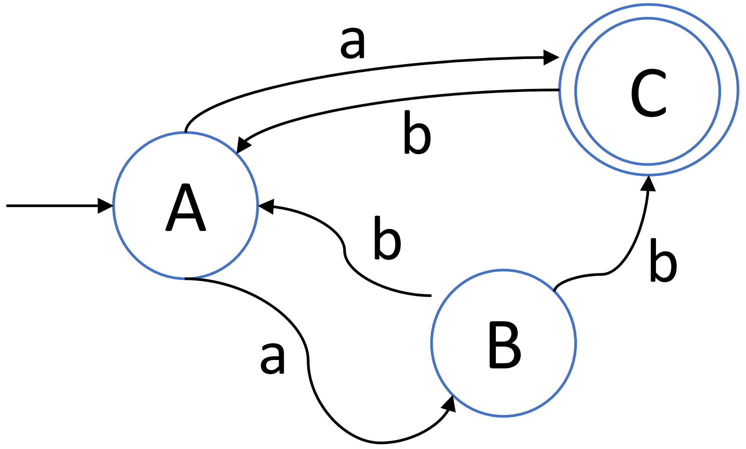
$\delta'[A,C],a = \delta[A],a \cup \delta[C],a = [B,C] \cup \emptyset = [B,C]$

$\delta'[A,C],b = \delta[A],b \cup \delta[C],b = \emptyset \cup [A] = [A]$

NFA to DFA: Example 1

- Convert the following NFA to a DFA.
- Transition table for the NFA:

	a	b
→A	B,C	-
B	-	A,C
*C	-	A



- Transitions for new states:

$\delta'[B,C],a = \delta[B],a \cup \delta[C],a = \emptyset \cup \emptyset = \emptyset$

$\delta'[B,C],b = \delta[B],b \cup \delta[C],b = [A,C] \cup [A] = [A,C]$

$\delta'[A,C],a = \delta[A],a \cup \delta[C],a = [B,C] \cup \emptyset = [B,C]$

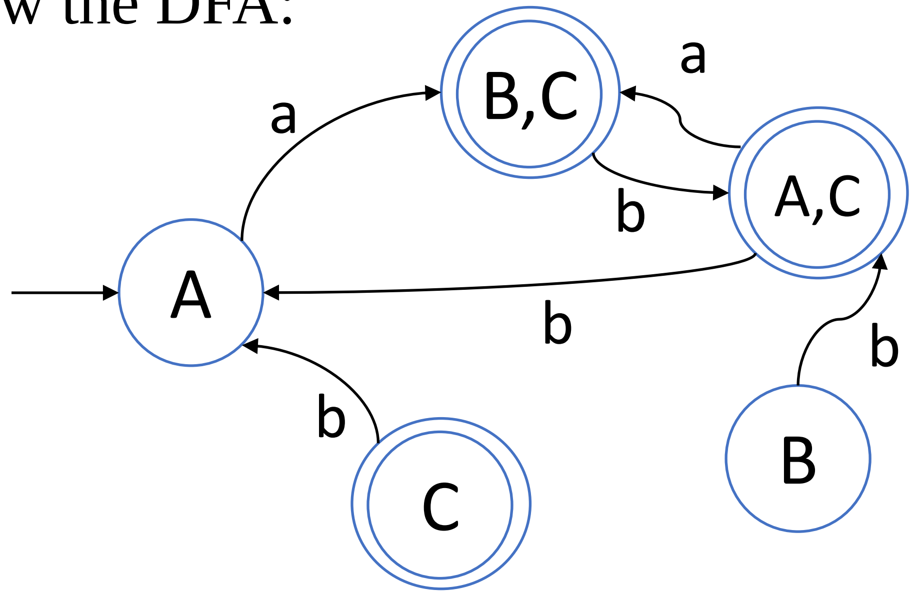
$\delta'[A,C],b = \delta[A],b \cup \delta[C],b = \emptyset \cup [A] = [A]$

- Transition table for the DFA:
  - [B,C] and [A,C] are also accept states, because they contain accept state C

	a	b
→A	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A

NFA to DFA: Example 1

- Draw the DFA:

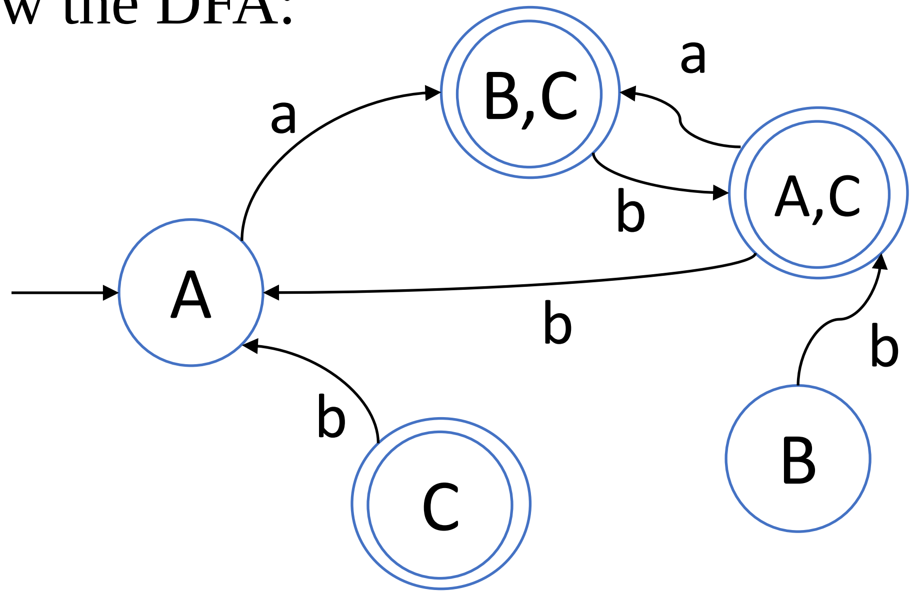


	a	b
→A	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A

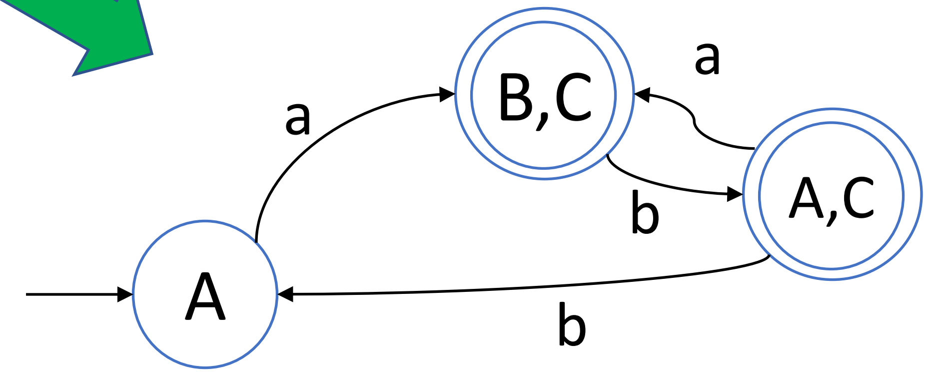
NFA to DFA: Example 1

	a	b
→A	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A

- Draw the DFA:

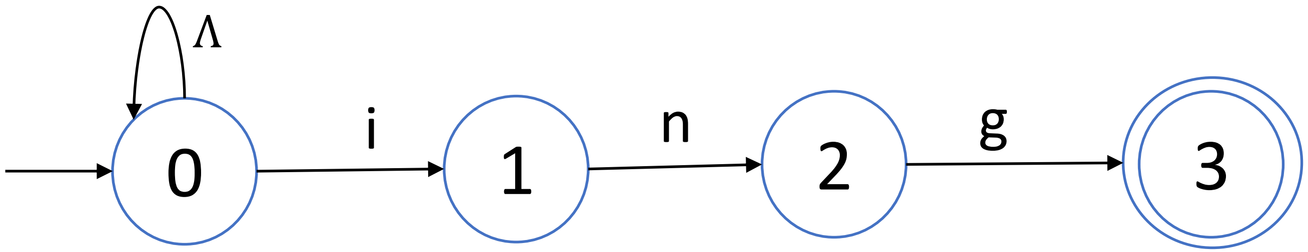


- No transitions can take us from the initial state to B or C, so these states are unreachable → can be discarded:



NFA to DFA: Example 2

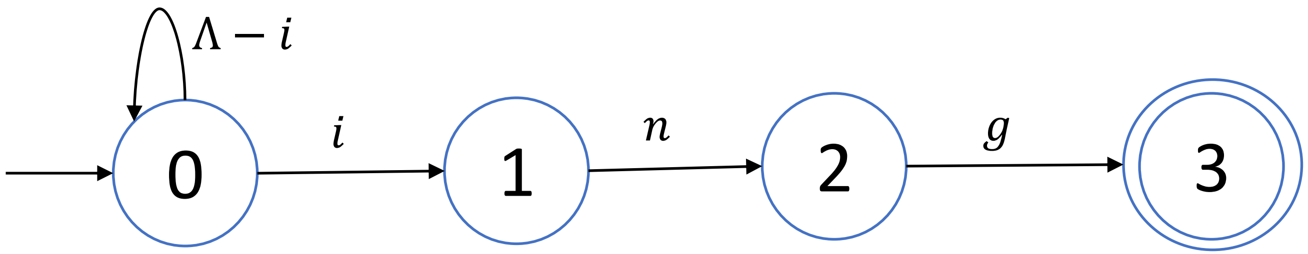
- Sometimes we can formulate a DFA from an NFA by using more “common sense”
- Suppose we want to create an automaton which identifies words that end in suffix “–ing”. For this kind of a problem, an NFA can be constructed rather easily:



  - Here, the symbol Λ means “any character”
  - One would think that this automaton wouldn’t work, because when it encounters an “i”, it can go either to 0 or to 1 – but it does; the automaton goes through all possible paths until the word has been either a) identified or b) deemed unidentifiable.
- How could we construct this into a DFA that does exactly what we want?

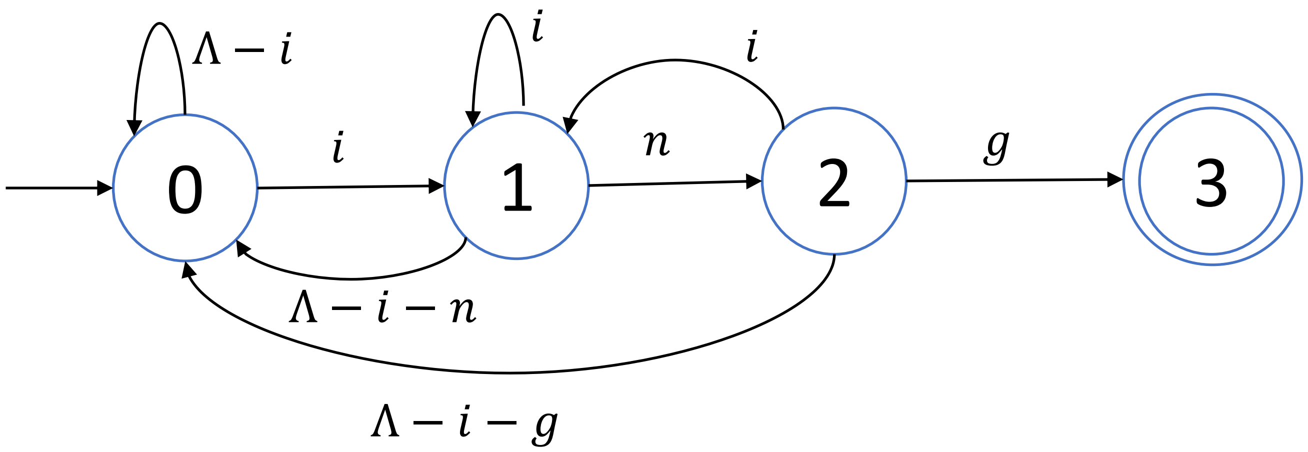
NFA to DFA: Example 2

- First modification is easy: let's remove i from "all characters"
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, "shipping" would cause an error (the first "i" it encounters isn't the one that belongs to the "-ing" suffix)



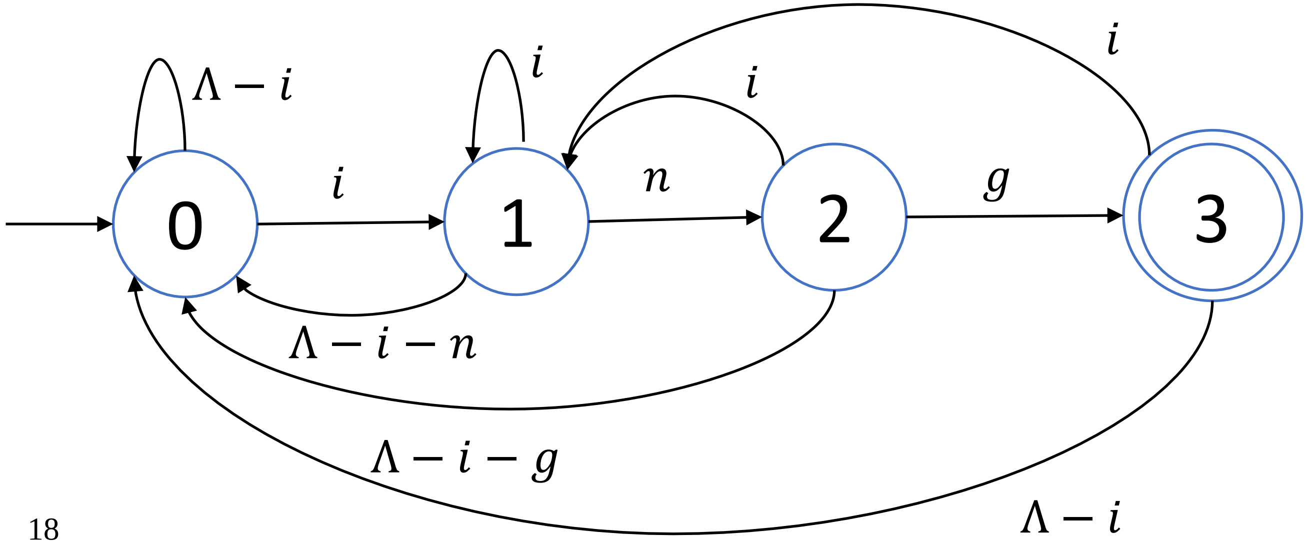
NFA to DFA: Example 2

- First modification is easy: let's remove i from "all characters"
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, "shipping" would cause an error (the first "i" it encounters isn't the one that belongs to the "-ing" suffix)
- Second modification: enable going backwards in the automaton
  - Does it work now? No, because it doesn't detect whether the word *ends* in –ing. (For example, "ringer" or "upbringing" would be problematic – depending on the setup of the automaton.)



NFA to DFA: Example 2

- First modification is easy: let's remove i from "all characters"
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, "shipping" would cause an error (the first "i" it encounters isn't the one that belongs to the "-ing" suffix)
- Second modification: enable going backwards in the automaton
  - Does it work now? No, because it doesn't detect whether the word *ends* in –ing. (For example, "ringer" or "upbringing" would be problematic – depending on the setup of the automaton.)
- 3rd modification
  - Back loops from state 3
- Now it works!



String search using regular expressions

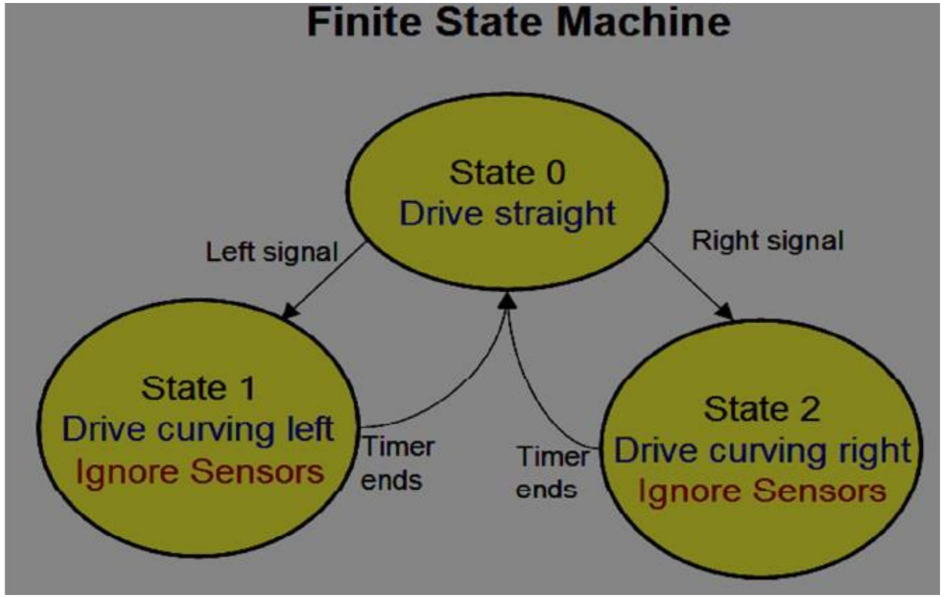
- In previous examples we used automata to search for strings that fulfilled our given conditions
- Instead of using an automaton, we can describe these strings using *regular expressions* (regex) – the most effective way to represent any language
- We've already encountered some of these before, but let's dive a bit deeper now:
  - Asterisk: a* = 0 to infinite number of concurrent a's
  - Plus: a+ = 1 to infinite number of concurrent a's
  - Question mark: ab?c = zero or one b's (so, "abc" and "ac" are accepted)
  - Wildcard (dot): a.b = the dot can be any character
  - Boolean OR: a|b = a or b
  - Parentheses: (abb|bab)a = "abba" OR "baba"
  - Curly braces: a{3,5} = 3 to 5 pcs of concurrent a's

Regular expressions and automata

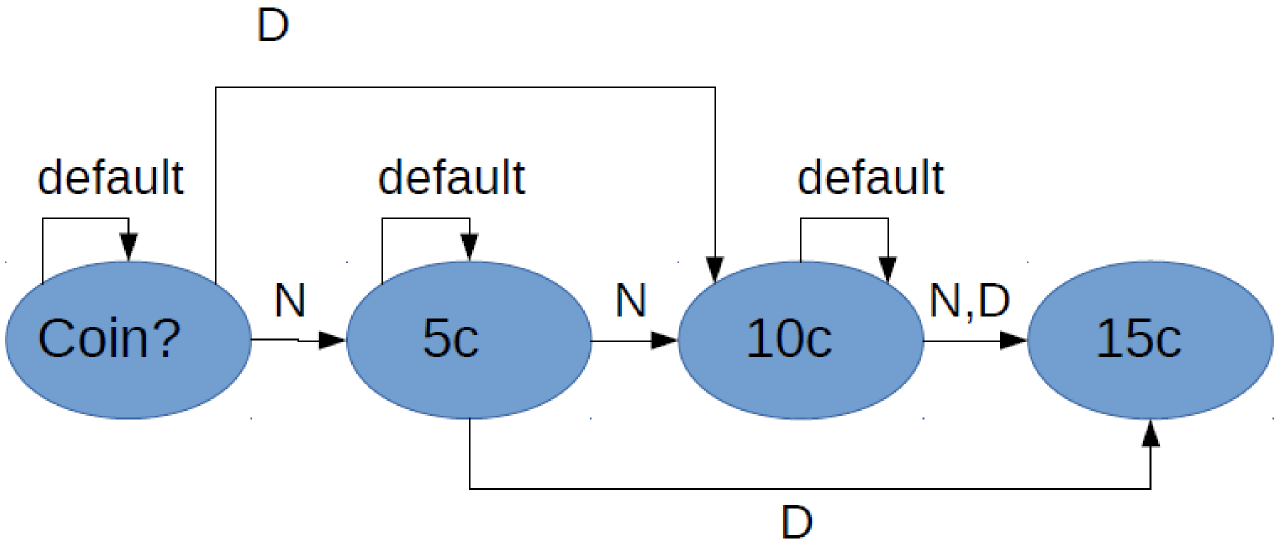
- Regular expressions are the simplest way to define a search string
- All regular expressions can be converted to an automaton
- This conversion is actually done by first creating an NFA from the regular expression and then converting that to a DFA
- Regular expressions are widely used in programming language grammars, some search engines & text processors ("find & replace")
  - Not Google, though - since the larger the database, the more resource-intensive their use is
- Hence, knowing how to use these is a nice skill to have
- Really good site to practice: https://www.regexpal.com/
  - Allows the user to give a test string and then check in real-time how many matches the given regular expression produces

Practical automata examples

- Lane assist in a car
  - Turn signal changes state
  - In states 1 and 2, lane detection sensors are ignored



- In the old days there were vending machines that sold Coca-Cola for 15 cents a bottle (nowadays inflation has caught up)
  - default = no money added
  - D = dime (10 cents)
  - N = nickel (5 cents)
  - Accept state = 15c
  - Note! No change given

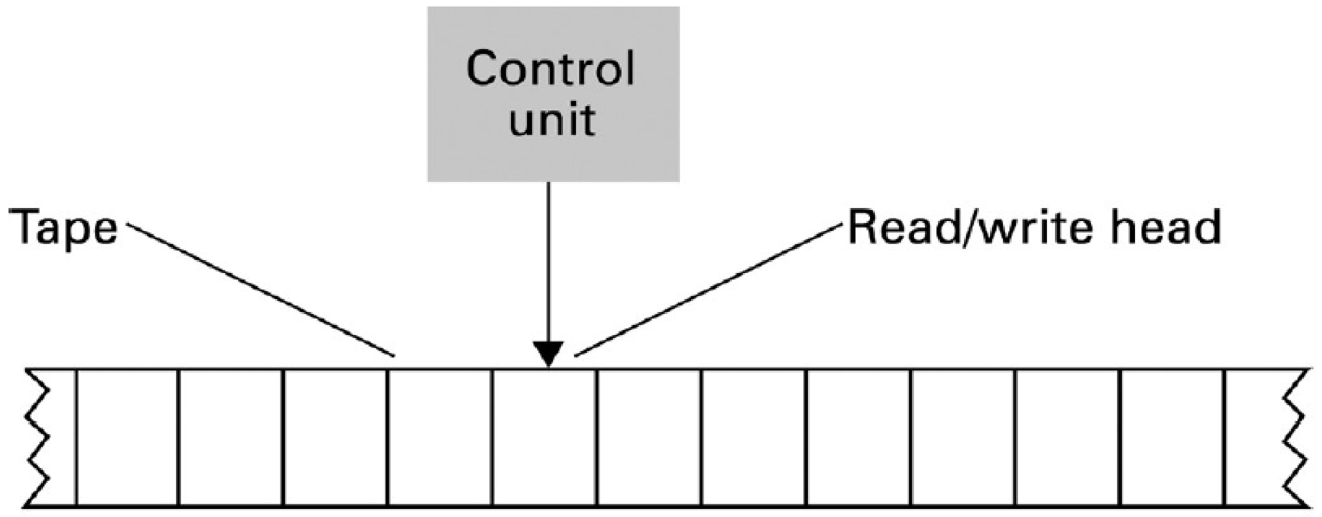


Turing machine

- State machines were quite primitive automata; they didn’t have memory, so the transition was only dependent on the current state and next input character
- If we expand our automaton by adding a memory, we end up in a primitive model of a computer called *Turing machine* (according to Alan Turing, 1936)
  - Actually there’s a “middle version” called pushdown automata (PDA) in between these; a PDA doesn’t have memory, but it employs a stack
- Memory of a Turing machine is a tape, which can be both read and written on
- This is done via a read/write head, which can read the tape one character at a time
  - After operation, read/write head can be moved one step at a time to the left or right
- The write-possibility enables us to also modify the input – while in a DFA, the input could only either be accepted or rejected

Structure of a Turing machine

- A Turing machine is a simple mathematical model of computation
  - Tape is infinitely long and it has been divided to cells
  - A tape cell can contain any symbol from the symbol group (alphabet of the machine)
  - Control unit reads and/or writes the symbols on tape cell by cell
  - Control unit can move the read/write head left (L), right (R) or stay (S) in place



How Turing machine works

- Calculation always starts from initial state and ends in final state
- Calculation consists of steps made by the control unit
- A step consists of
  - Reading the cell on the tape
  - Writing on the cell on the tape
  - Moving the read/write head (or tape – some authors think that the tape moves)
  - Changing the state
- Early computers were basically Turing machines
  - Memory could only be used in specific order
- Nowadays modern computers use RAM, which can be read or written in any order
  - So, modern computers are more agile than Turing machines
- Still, a Turing machine can perform all calculations that a computer does!

Definition of a Turing machine

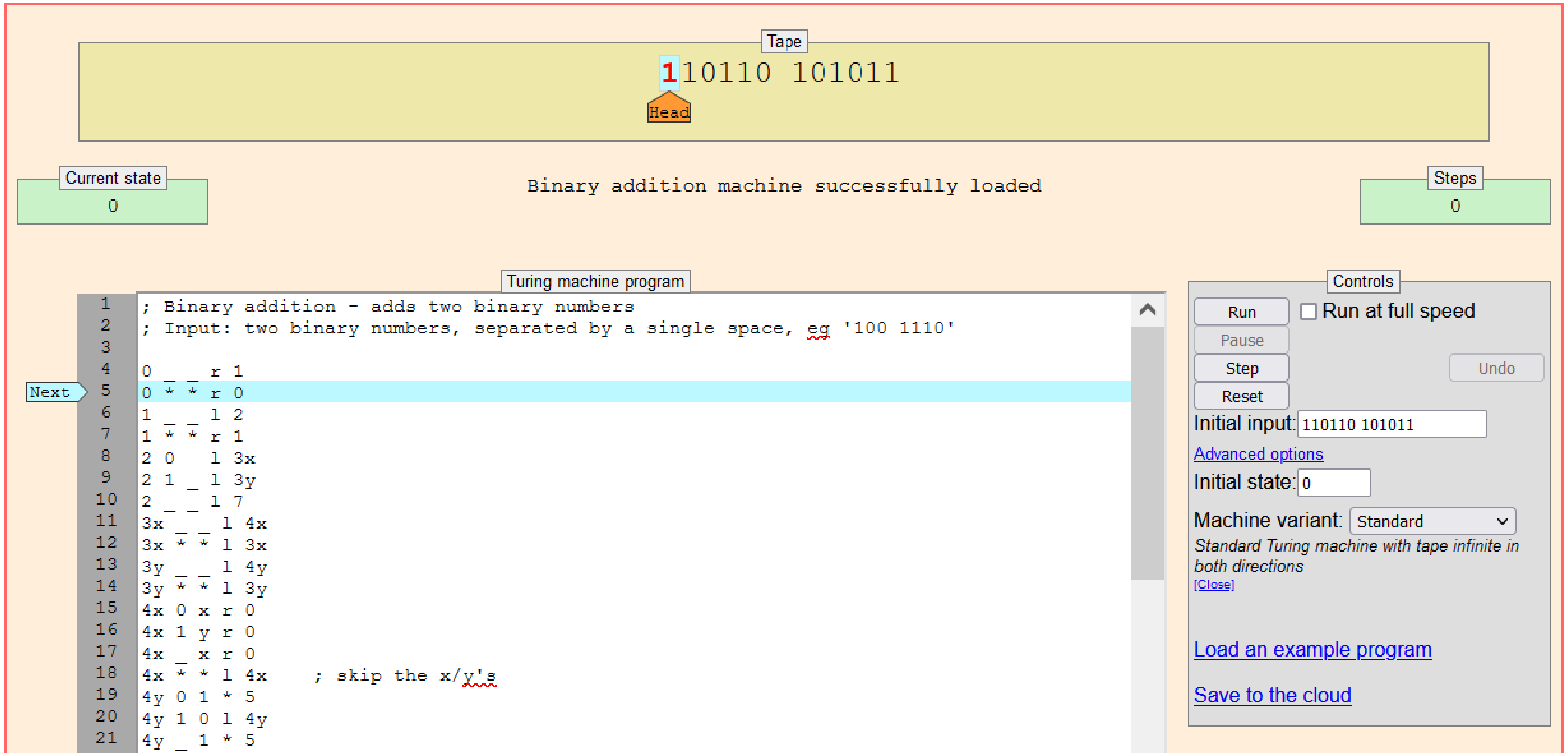
- A Turing machine M is defined by a 7-tuple $M = (Q, T, I, \delta, b, q_0, q_F)$:
  - Q = a finite group of states
  - T = a group of tape symbols
  - I = the set of input symbols (Note: $I \subseteq T$)
  - δ = a transition function that specifies the transitions $Q \times T \rightarrow Q \times T \times \{L, S, R\}$
  - b = blank symbol
  - q_0 = initial state (Note: $q_0 \in Q$, naturally)
  - q_F = set of final states (Note: $q_F \subseteq Q$, naturally)
- Example of a transition: $q_1, x \rightarrow q_2, y, L$
  - Meaning: if we're currently in state q_1 and the symbol on tape is x
  - Procedure in this case: write symbol y on tape, move the read/write head left, switch to state q_2

Morphett Turing simulator

- Behavior of different Turing machines can be investigated using a Turing simulator
- There are many of these online, but this Morphett's version seems like the best: https://morphett.info/turing/turing.html
- Learn how to use this simulator by trying out some of the example programs
- Some things to notice:
  - In Morphett, state transformations syntax is different - it specifies the transitions in order: (current state, symbol on tape, symbol written on tape, head move direction, new state to enter)
  - So, for example, the previous transition $q_1, x \rightarrow q_2, y, L$ in Morphett would be $q_1\ x\ y\ L\ q_2$ (separated only by one spacebar)
  - Default initial state is 0, but this can be changed from ”Advanced options”
  - Head position can be specified using an asterisk (*) in the input

Morphett Turing simulator

- Use “Step” button in order to see step by step how the machine proceeds
- On the right machine shows the step number
- Try different inputs!



Example 1

- What does this Turing machine do? (Starts from right side of input)

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

Example 1

- What does this Turing machine do? (Starts from right side of input)
  - After a couple of simulations, we see that it adds 1 to the input (binary addition: 101 → 110)

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

Example 2

- What does this Turing machine do? (starts from right side of input)

$M = (Q, T, I, \delta, b, q_0, q_f)$

$Q = \{1, 2, 3, H\}$

$T = \{0, 1, \_\}$

$I = \{0, 1\}$

$b = \_$

$q_0 = 1$

$q_f = H$

$q_i, x \rightarrow q_j, y, \{L, S, R\}$

$$
\delta = \begin{array}{l}
1, \_ \rightarrow 1, \_, L \\
1, 0 \rightarrow 2, 0, L \\
1, 1 \rightarrow 2, 1, L \\
2, \_ \rightarrow 3, \_, R \\
2, 0 \rightarrow 2, 0, L \\
2, 1 \rightarrow 2, 1, L \\
3, \_ \rightarrow H, \_, S \\
3, 0 \rightarrow 3, 0, R \\
3, 1 \rightarrow 3, 1, R
\end{array}
$$

Example 2

- What does this Turing machine do? (starts from right side of input)
  - Nothing much – it seems to search for the nearest blank space that has a number on its right side, and then comes back
  - Note: tape symbols are not altered in any transition!

$M = (Q, T, I, \delta, b, q_0, q_f)$

$Q = \{1, 2, 3, H\}$

$T = \{0, 1, \_\}$

$I = \{0, 1\}$

$b = \_$

$q_0 = 1$

$q_f = H$

$q_i, x \rightarrow q_j, y, \{L, S, R\}$

$\delta =$

1, _ → 1, _, L
1, 0 → 2, 0, L
1, 1 → 2, 1, L
2, _ → 3, _, R
2, 0 → 2, 0, L
2, 1 → 2, 1, L
3, _ → H, _, S
3, 0 → 3, 0, R
3, 1 → 3, 1, R

Example 3

- What does this Turing machine do? (starts from left side of input)

$M = (Q, T, I, \delta, b, q_0, q_f)$

$Q = \{1, 2, 3, 4, 5, 6, H\}$

$T = \{0, 1, \_\}$

$I = \{0, 1\}$

$b = \_$

$q_0 = 1$

$q_f = H$

$\delta =$
1, _ → H, _, S
1, 0 → 2, 0, S
1, 1 → 2, 0, S
2, _ → 5, _, L
2, 0 → 3, 0, L
2, 1 → 4, 1, L
3, _ → 6, 0, R
3, 0 → 6, 0, R
3, 1 → 6, 0, R
4, _ → 6, 1, R
4, 0 → 6, 1, R
4, 1 → 6, 1, R
5, _ → H, _, S
5, 0 → H, 0, S
5, 1 → H, 0, S
6, _ → 2, _, R
6, 0 → 2, 0, R
6, 1 → 2, 1, R

Example 3

- What does this Turing machine do? (starts from left side of input)
  - Needs a couple of simulations to understand
  - Machine treats the input as a number with a sign (two's complement)
  - It takes the absolute value of the input and then multiplies it by two

$$M = (Q, T, I, \delta, b, q_0, q_f)$$
$$Q = \{1, 2, 3, 4, 5, 6, H\}$$
$$T = \{0, 1, \_\}$$
$$I = \{0, 1\}$$
$$b = \_$$
$$q_0 = 1$$
$$q_f = H$$

δ =
1, _ → H, _, S
1, 0 → 2, 0, S
1, 1 → 2, 0, S
2, _ → 5, _, L
2, 0 → 3, 0, L
2, 1 → 4, 1, L
3, _ → 6, 0, R
3, 0 → 6, 0, R
3, 1 → 6, 0, R
4, _ → 6, 1, R
4, 0 → 6, 1, R
4, 1 → 6, 1, R
5, _ → H, _, S
5, 0 → H, 0, S
5, 1 → H, 0, S
6, _ → 2, _, R
6, 0 → 2, 0, R
6, 1 → 2, 1, R

Try these yourself! All these 3 examples have been converted to Morphett code in the .txt file that can be found in Moodle. Just copy & paste the Turing machine in Morphett and experiment with different inputs!

Thank you for listening!