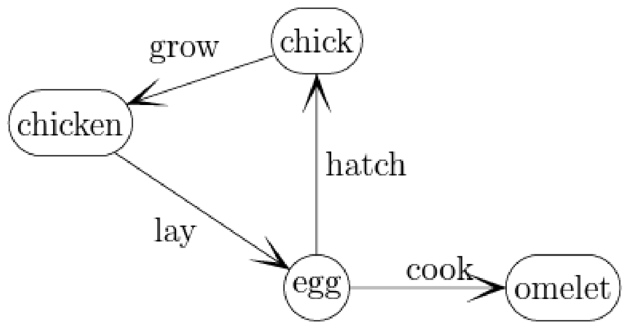
Chapter 19-State Diagrams

Tuesday, January 10, 2023
10:41 PM

State Diagrams:

A type of directed graph, in which the graph nodes represent states and labels on the graph edges represent actions.

Example:

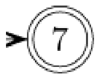


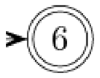
(mm yummy)

Walks/paths are a little different for state diagrams.
(For example, you cannot go from omelet to chicken, despite them being connected)

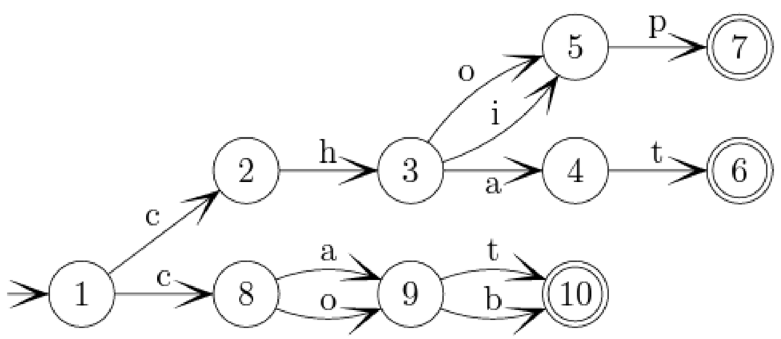
The Walk in graphs contains both a sequence of *nodes* and a sequence of *edges*.
The walk in state diagrams have a sequence of **states** and a sequence of **actions**.
It's often important to include both sequences,

Sometimes the actions are not marked, but it can be easily inferred from the change in state.
End states are marked with a double ring, examples below:





Phone Lattices:
State diagrams that change state based on letters inputted and end states are possible words.



(end state possibilities are: {chat,chop,chip,cat,cot,cab,cob}, and states 2 and 8 can be merged if you want)

How to Represent/Model State Diagrams (or functions in general):
Think about a set of states, and a set of actions, and then some way to map out how actions result in state changes.
So an input would look something like (*state*, *action*), and the output would be a **set** of *states*.
(output is a set because we might output an empty set, or more than one state in some case)

The tricky part is to figure out an efficient way to map out how actions result in state changes (the transition function).
One approach is to build a 1D array, with the *x*th element representing state *number x,* and the element (state) contains a list of actions possible from that state, together with the new states for each action.
This is called the *adjacency list* style.

There is another way, called a *hash function*.
They map the state/action pairs to a small integers, and each state/action pair gets a position in a 1D array, and each position contains a list of new states.
(Though not relevant to the class, it'd still be nice to dig deep into hash functions and explain it here, but um… I prefer my 8 hours of sleep)

Dynamic Programming:
Merging states is a really good thing, since it saves both time and space.
When we generate states to explore different paths or end states, if we build a record (index) of states we've already generated, we can detect when we generate the same state (or reach the beginning of the same sequence) for the

second time. We can then use results from our previous work. This is called ***Dynamic Programming***.

Consider the following pseudocode as example:

```
cyclic()
    y = 0
    x = 0
    while (y < 100)
         x = remainder(x+1,4)
         y = 2x
```

Instead of making a bunch of states, we could just use

$$\begin{tikzcd}
{} \arrow[r] & x=0,y=0 \arrow[r] & x=1,y=2 \arrow[r] & x=2,y=4 \arrow[r] & x=3,y=6 \arrow[lll, bend right]
\end{tikzcd}$$

However, you need to keep in mind that sometimes such patterns are much harder to spot, or simply nonexistent.
A system can also have a very large number of states, and sometimes infinite (for example, the game of life can have an infinite number of alive cells for certain start configurations).