Finite State Machines
Classes

- Formal Languages
- Finite State Machines
Formal Languages
A natural language is used for human communication

- Spoken, written or gestured
  - e.g. English, French, Mandarin, Klingon

There are rules

- Valid characters
- Valid words
- Valid sentences
- Acceptable idioms

The human brain is pretty good at coping with language errors

Computers, considerably less so
A formal language is used to distinguish precisely what is allowed from what is not.

- Expressed mathematically, often using recursion
  - e.g. valid postfix expressions, valid C++

Similarly to natural languages there are:

- Alphabets
- Words
- Grammars
- But no idioms
An *alphabet* is a finite collection of symbols

- e.g. $\Sigma = \{a, b, c, \ldots, x, y, z\}$ – letters of the alphabet
- e.g. $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ – base ten digits
- e.g. $\Sigma = \{0, 1\}$ – binary digits

A *word* is a finite sequence of alphabet symbols

- Symbols may be repeated
  - e.g. baa, 100, wool, sheep
- Order matters
  - e.g. listen, silent
- The word of length zero is special
A (formal) language is a set of words

- Can be finite
  - e.g. \( L = \{ \text{all valid English words} \} \)
- Or infinite
  - e.g. \( L = \{ \text{all valid decimal numbers} \} \)
- But the words themselves are of finite length

Rules specify which words are valid or invalid

- The rules that describe a language are referred to as a grammar
Like a natural language, use a grammar

- Describe the symbols allowed and the order in which they should appear
  - Usually specified recursively

Examples

- A valid sentence is a noun phrase followed by a verb phrase followed by a subordinate clause
  - A subordinate clause may be composed of the symbol *where* followed by a valid sentence

- A valid postfix expression is either a single number or two valid postfix expressions followed by an operator
A grammar can be represented using *production rules*

- For postfix
  - $E \rightarrow \text{number}$
  - $E \rightarrow EE \text{ operator}$
  - A postfix expression is either a number or two postfix expressions followed by an operator

- We can write algorithms to take input, break it into its components and determine if it is syntactically correct
  - Known as a *parser*
  - Parsing is the process of analyzing a string of symbols that conform to a grammar
Modelling Computation

- Use a *finite state machine* to model the rules of a language
- FSM rules
  - Finite number of states
    - FSM reads one character at a time
    - Next state is determined by looking at the current state and the next input character, and nothing else
    - Each state has at most one transition on any given character
    - Previously read characters may not be read again
  - One state is identified as the *start* state
    - One or more states are identified as *final* states
    - If the last state is a final state – *accept*
    - If the last state is not a final state – *reject*
Finite State Machines
Dead States

- In this presentation
  - Final states outlined in green
  - Start state pointed to by a green arrow
- The yellow state is a dead state
  - Dead states are not final states
  - Dead states cannot be transitioned from
    - They only transition to themselves
  - Dead states are usually not shown
    - In addition transitions that are not shown go to a dead state
- Build an FSM that accepts all words of length 3
  - $\Sigma = \{a, b\}$
- Build an FSM that accepts all decimal integers
  - Disallow leading zeros
  - $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
  - Dead state not shown
    - Any transition from 0 final state
Implement in a simple loop

Algorithm:

```
state = start
while there is still input
    c = next input symbol
    if transition(state, c) exists
        state = transition(state, c)
    else
        reject (or state = dead state)
end while
if state is a final state accept
else reject
```

Implement transitions with a table or a case statement

<table>
<thead>
<tr>
<th>state / c</th>
<th>0</th>
<th>1-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>begin 0</td>
<td>begin 1-9</td>
</tr>
<tr>
<td>begin 0</td>
<td>dead</td>
<td>dead</td>
</tr>
<tr>
<td>begin 1-9</td>
<td>begin 1-9</td>
<td>begin 1-9</td>
</tr>
</tbody>
</table>
Augmenting FSMs

- FSMs can be augmented with other information
- Actions
  - Transitions to a state may be associated with an action
    - Such as the calculation of a value
  - Shown after the input character on the transition arc
    - Typically separated from the input by a /
- Output
  - Transitions may also be associated with output
  - Again, shown after the input character(s) associated with the transition
FSM Actions

- Perform an action during a transition
  - Place actions on transition, following a slash
- What might be a useful action in the FSM to accept integers?
  - The value of the integer
    - A1: val = c
    - A2: val = 10 * val + c
Build an FSM that performs block reduction
  - That reports 0 for each sequence of 0s
  - And 1 for each sequence of 1s
Example
  - 111000010011100011 goes to
  - 1010101
Note that $\epsilon$ represents the empty string
FSM Summary

- FSMs are a mathematical model of computation
  - The behavior of many devices can be modeled by a state machine
    - Vending machines
    - Traffic lights
    - ...
- FSMs can be used to model systems
  - In engineering
  - And computing science
    - To model the behavior of an application