Finite State Machine (FSM)

Takes a sequence of symbols from some alphabet as input.

Produces a sequence of symbols from some alphabet as output.

Has a finite set of (computational) states, with one start state.

Has a set of state transitions (computational steps):
Current state & input symbol determine output symbol & next state.

Can be given as

- a table, with a row per state and a column per input symbol, where each entry gives output and next state;
- a labeled directed graph, with a node per state, and an arrow between nodes labeled by input and output per transition.

Also known as Deterministic Finite Automaton (DFA).
Input alphabet = output alphabet = \{ 0, 1, \# \}

The number is input with its \textit{least significant bit} first.

<table>
<thead>
<tr>
<th>Output, Next State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>0</td>
</tr>
<tr>
<td>( A^*(\text{Carry 1}) )</td>
<td>1, ( B )</td>
</tr>
<tr>
<td>( B ) (\text{Carry 0})</td>
<td>0, ( B )</td>
</tr>
<tr>
<td>( C ) (Done)</td>
<td>( C )</td>
</tr>
</tbody>
</table>

19 = 10011_2
20 = 10100_2

How to increment a \textit{decimal} number by one? Multiply by 3?
How to check divisibility by 7? How to add two numbers?

*Start state
FSM has limited computational power

There exists no FSM

1. to increment an (arbitrary sized) number by one, when the number is offered with its *most significant digit* first;

2. to reverse the order of a sequence of symbols;

3. to multiply two (arbitrary sized) numbers;

4. to recognize a sequence of properly nested parentheses.

Reason: a FSM has finite storage.
Pushdown Automaton (PDA)

Like a FSM, but it also has an unbounded stack.

The stack stores a sequence of symbols from the stack’s alphabet.

Besides the input symbol and current state, also the symbol at the top of the stack determines which transition the PDA takes.

Besides producing output and going to the next state, the action with a transition can also push a symbol onto the top of the stack, or pop the top symbol off the stack.
PDA has limited computational power

A PDA can do everything that a FSM can do, and even more:

- reverse a sequence of symbols
- recognize a sequence of properly nested parentheses

There exists no PDA

- to multiply two numbers
- etc.

Reason: a PDA has infinite storage with very limited access
Turing Machine (TM)

Like a FSM, but it also has a two-way infinite read-write tape.

The tape is divided into cells that each hold one symbol.

It has a read/write head that is positioned at a specific tape cell.

The current state and tape symbol under the R/W head determines the next state.

As action, it also can write a symbol at the head position and move the head one cell left/right.
Turing Machine that finds a 1 among 0s

Transition $X / Y Z$ means

'when reading $X$, overwrite by $Y$ and move the R/W head in direction $Z' \ (R = \text{right}; \ L = \text{left})."
Formal Grammar to Generate a Formal Language

- Set of **terminal symbols**
- Set of **non-terminal symbols**

One non-terminal symbol is designated as start symbol.

- Set of **production rules**, each of the form
  
  sequence of symbols → sequence of symbols

Grammar *generates* a set of sequences of symbols (formal language):

1. Start with the sequence consisting of just the start symbol.
2. Repeatedly replace subsequence $t$ by $u$ for production rule $t \rightarrow u$.
3. Terminate when the sequence contains terminal symbols only.
Formal Grammar: Example 1

Terminals: \{ a, b \}. Non-terminals: \{ S, T \}, start symbol \( S \)

Production rules:

1. \( S \rightarrow aT \)
2. \( T \rightarrow b \)
3. \( T \rightarrow bS \)

Production example:

\[
S \quad \frac{1}{\rightarrow} \quad aT \quad \frac{3}{\rightarrow} \quad abS \quad \frac{1}{\rightarrow} \quad abaT \quad \frac{3}{\rightarrow} \quad ababS \quad \frac{1}{\rightarrow} \quad ababaT \quad \frac{2}{\rightarrow} \quad ababab
\]

Generated language = \{ \( (ab)^n \mid n \geq 1 \) \}

© 2011, T. Verhoeff @ TUE.NL
Formal Grammar: Example 2

Terminals: \( \{ a, b \} \). Non-terminals: \( \{ S \} \), start symbol \( S \)

Production rules:

1. \( S \rightarrow ab \)
2. \( S \rightarrow aSb \)

Production example:

\[
\begin{align*}
S & \rightarrow 2 \rightarrow aSb \rightarrow 2 \rightarrow aaaSbb \rightarrow 1 \rightarrow aaabbb
\end{align*}
\]

Generated language = \( \{ a^n b^n \mid n \geq 1 \} \)
Formal Grammar: Example 3

Terminals: \{a, b, c\}. Non-terminals: \{S, T\}, start symbol S

Production rules:

1. \( S \rightarrow aTSc \)
2. \( S \rightarrow abc \)
3. \( Ta \rightarrow aT \)
4. \( Tb \rightarrow bb \)

Production example:

\[
\begin{align*}
S & \quad 1 \rightarrow \underline{aT}S\underline{c} \\
& \quad 2 \rightarrow \underline{aT}ab\underline{c}c \\
& \quad 3 \rightarrow \underline{aaT}b\underline{c}c \\
& \quad 4 \rightarrow a\underline{a}b\underline{b}c\underline{c}
\end{align*}
\]

Generated language = \{ \( a^n b^n c^n \) | \( n \geq 1 \) \}
Chomsky Hierarchy of Grammars

- **Unrestricted Grammar** (Type 0): no restrictions

- **Context-Free Grammar** (Type 2):

  *left-hand side* of each production rule consist of a *single non-terminal symbol*

- **Regular Grammar** (Type 3): like context-free grammar, but also

  *right-hand side* of each production rule is restricted:

  - either empty, or
  - single terminal symbol, or
  - single terminal symbol followed by single non-terminal symbol
Classification of Example Grammars

- Grammar 1 is regular
- Grammar 2 is context-free, and not regular
- Grammar 3 is unrestricted, and not context-free
Classification of Languages

<table>
<thead>
<tr>
<th>A language is called . . .</th>
<th>if it can be generated by a . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Regular Grammar</td>
</tr>
<tr>
<td>Context-Free</td>
<td>Context-Free Grammar</td>
</tr>
<tr>
<td>Recursively Enumerable</td>
<td>Unrestricted Grammar</td>
</tr>
</tbody>
</table>

- Note: Different grammars can define the same language.

- Regular ⊂ Context-Free ⊂ Recursively Enumerable

- Context-free, non-regular grammar can generate regular language

- Unrestricted non-context-free grammar can generate context-free language
Classification of Example Languages

- \( \{ (ab)^n \mid n \geq 1 \} \) is regular
- \( \{ a^n b^n \mid n \geq 1 \} \) is context-free, but not regular
- \( \{ a^n b^n c^n \mid n \geq 1 \} \) is recursively enumerable, but not context-free
Automata as Language Recognizers

By marking some states as accepting states an automaton can be turned into a recognizer:

An automaton is said to accept a sequence when it terminates in an accepting state after processing that sequence as input.

An automaton is said to recognizes a language, if it accepts exactly the sequences of that language (no more, and no less).

- Finite State Machines recognize Regular Languages.
- Pushdown Automata recognize Context-Free Languages.
- Turing Machines recognize Recursively Enumerable Languages.
There are many different computational mechanisms, such as:

- Finite State Machine (FSM)
- Pushdown Automaton, with a single stack (PDA)
- Turing Machine, with a two-way infinite read/write tape (TM)

These also differ in computational power.

A formal language is a set of sequences of symbols. A formal grammar defines (generates) a formal language:

- Regular Grammar (recognizable by FSM)
- Context-Free Grammar (recognizable by PDA)
- Unrestricted Grammar (recognizable by TM)
References


http://en.wikipedia.org/wiki/Pushdown_automaton

http://en.wikipedia.org/wiki/Turing_machine

http://en.wikipedia.org/wiki/Computability


http://en.wikipedia.org/wiki/Formal_grammar