Basic Components — Semantics

Steven Zeil

Sep. 7, 2001

Contents

Basic Components of Programming Languages

1. History
2. Classification
3. Translation
4. Semantics

---

4 Semantics

- What kinds of things can programming languages do?
- Are some programming languages more powerful than others?

Simplifying assumptions:

1. programs read and write character streams
2. one input stream, one output stream

4.1 Automata

Describe fundamental levels of computing power in terms of a set of simple machines:

- Finite State Automata
- Push-Down Automata
- Linear Bouded Automata
- Turing Machine

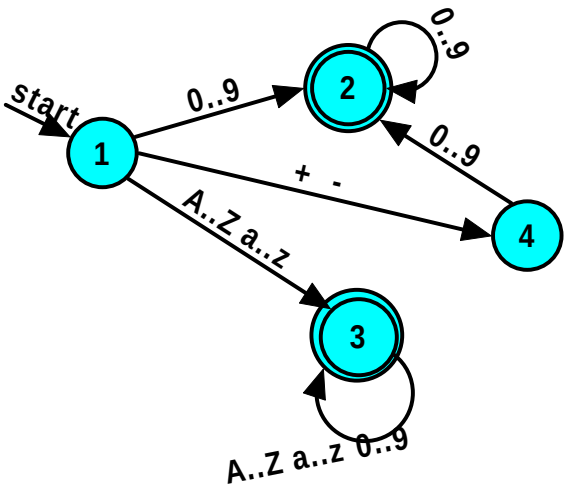
4.1.1 Finite State Automata

(a.k.a. **finite state machines**)

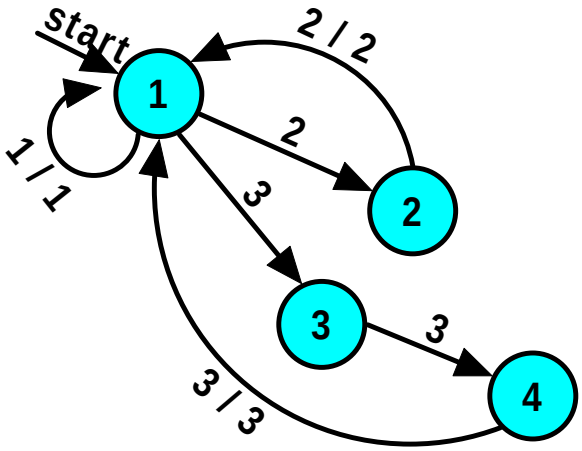
- a set of **states**
  - one designated as a **start** state
  - one or more designated as **accept** states
- linked by **transitions**
- each transition labelled with input chars and, optionally, an output string

---

A finite state machine with output:



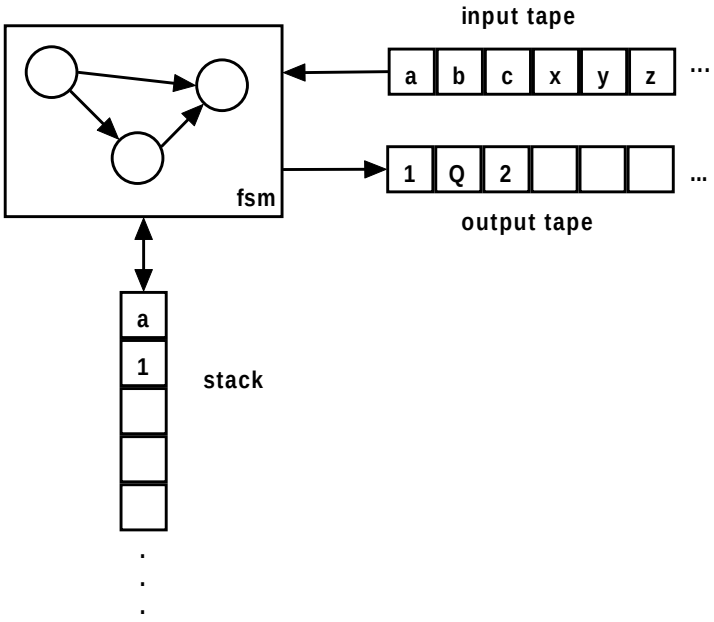
A finite state machine with accept states:



4.1.2 Push-Down Automata

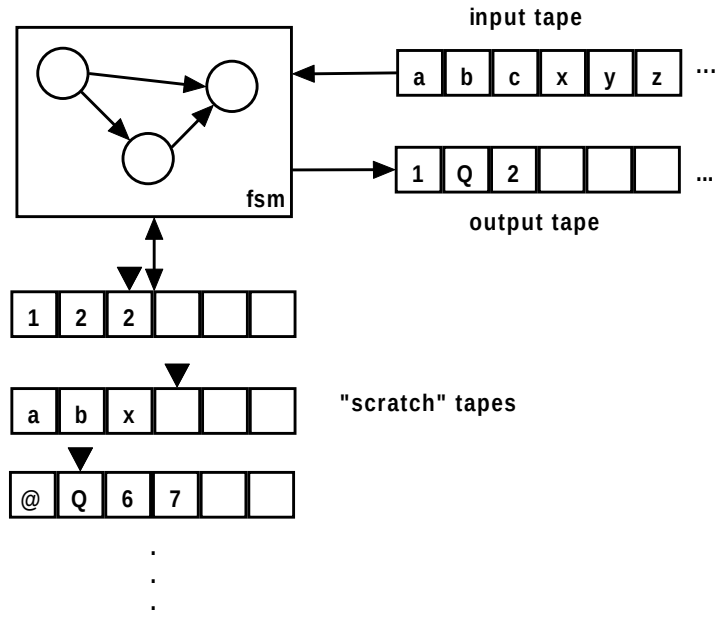
Add a stack to an FSA:

- Allow FSA transitions to depend upon top char on stack as well as next input char
- Allow FSA output to push/pop 1 char on stack
- Input tape read left to right



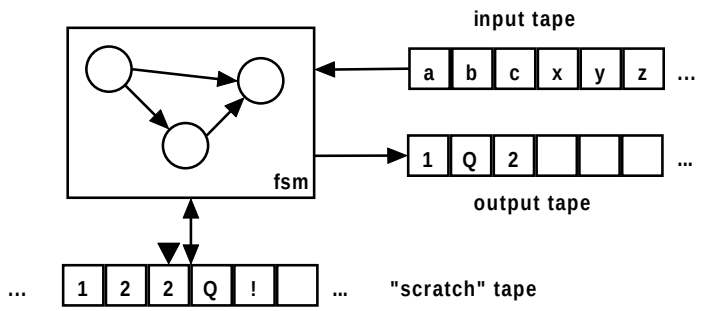
4.1.3 Linear Bouded Automata

- Replace PDA's stack by any finite number of finite-length "scratch" tapes
  - FSA can read and write chars to these tapes
  - FSA can move "read head" on scratch tapes forward or backwards



4.1.4 Turing Machine

- Replace PDA's stack by an infinite-length "scratch" tape
  - FSA can read and write chars to the scratch tapes
  - FSA can move "read head" on scratch tape forward or backwards



Alan Turing in 1936 discovered a **universal** Turing Machine, a machine that could

- read a description of any other Turing machine program from its input tape
- and could then emulate that other Turing machine

4.2 Church's Thesis

> Any computation scheme can be simulated by a Turing machine.
>
> A. Church, 1936

"Computation scheme" corresponds roughly to the idea of "anything that can be accomplished by an algorithm."

Supporting evidence:

- Easy to write Turing machines in most HLL's.
- TM's can be used to emulate a "Random Access Machine", an automaton that resembles a Von Neuman architecture (with infinite memory!)

Turing Complete Languages

A programming language is said to be **Turing complete** if it is powerful enough to allow someone to write a Turing machine emulator.

- Most HLL's are Turing complete
- And so are all equally powerful in the "Turing sense"

FSA also see a lot of use in programming

- specification, especially of protocols
- modelling and analysis
- many "data-driven" applications are implemented as FSA's
  - Implementing a FSA is simple

```
char c;
int state = 0;
while (1) {
  cin >> c;
  switch (state) {
    case 0:
      switch (c) {
        case 'a': state = 1; break;
        case 'b': state = 3; break;
        ⋮
      } break;
    case 1:
      switch (c) {
        ⋮
  }
}
```

```
struct StateDescription {
  string output;
  int nextstate;
};

StateDescription** fsa;
  ⋮
int state = 0;
while (1)
{
  char c;
  cin >> c;
  cout << fsa[state][c].output;
  state = fsa[state][c].nextstate;
}
```

4.3 Automata & Grammars

If we

- forget about the output, and
- concentrate on the set of strings that would leave an automaton in an "accept" state

---

Finite State	regular
Push-Down	context free
Linear Bounded	context sensitive
Turing	type 0

- Automatic scanner generators work by
  - accepting lexeme description written as regular expressions,
  - generating a FSA that rocognizes those regular expressions
  - generating code to emulate that FSA

- One of the more powerful techniques for parsing context-free grammars is called LR parsing.
  - Used by many automatic parser generators
  - Algorithm employs a FSA and a stack