State machines are wonderful tools

December 31, 2020
nullprogram.com/blog/2020/12/31/

This article was discussed on Hacker News.

I love when my current problem can be solved with a state machine. They're fun to design and implement, and I have high confidence about correctness. They tend to:

1. Present minimal, tidy interfaces
2. Require few, fixed resources
3. Hold no opinions about input and output
4. Have a compact, concise implementation
5. Be easy to reason about

State machines are perhaps one of those concepts you heard about in college but never put into practice. Maybe you use them regularly. Regardless, you certainly run into them regularly, from regular expressions to traffic lights.

Morse code decoder state machine

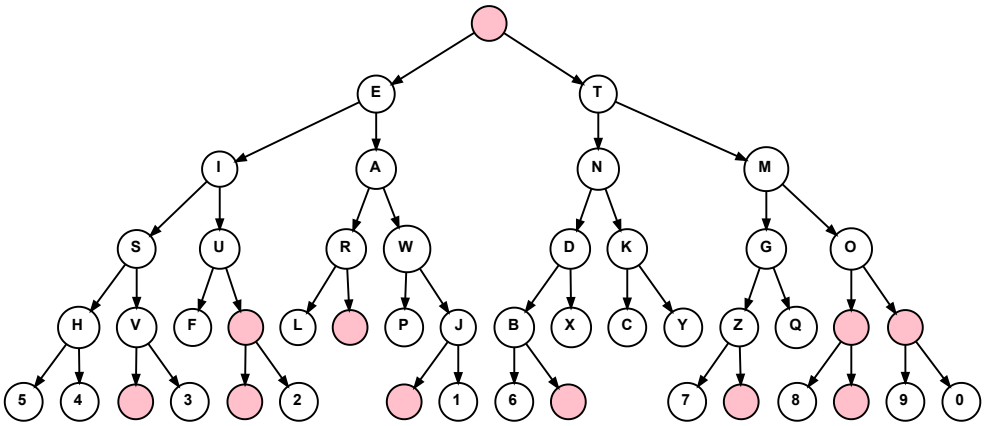
Inspired by a puzzle, I came up with this deterministic state machine for decoding Morse code. It accepts a dot (`'.'`), dash (`'-'`), or terminator (0) one at a time, advancing through a state machine step by step:

```
int morse_decode(int state, int c)
{
    static const unsigned char t[] = {
        0x03, 0x3f, 0x7b, 0x4f, 0x2f, 0x63, 0x5f, 0x77, 0x7f, 0x72,
        0x87, 0x3b, 0x57, 0x47, 0x67, 0x4b, 0x81, 0x40, 0x01, 0x58,
        0x00, 0x68, 0x51, 0x32, 0x88, 0x34, 0x8c, 0x92, 0x6c, 0x02,
        0x03, 0x18, 0x14, 0x00, 0x10, 0x00, 0x00, 0x00, 0x0c, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x1c, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x00, 0x00, 0x24,
        0x00, 0x28, 0x04, 0x00, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35,
        0x36, 0x37, 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46,
        0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50,
        0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5a
    };
    int v = t[-state];
    switch (c) {
    case 0x00: return v >> 2 ? t[(v >> 2) + 63] : 0;
    case 0x2e: return v &  2 ? state*2 - 1 : 0;
    case 0x2d: return v &  1 ? state*2 - 2 : 0;
    default:   return 0;
    }
}
```

It typically compiles to under 200 bytes (table included), requires only a few bytes of memory to operate, and will fit on even the smallest of microcontrollers. The full source listing, documentation, and comprehensive test suite:

https://github.com/skeeto/scratch/blob/master/parsers/morsecode.c

The state machine is trie-shaped, and the 100-byte table `t` is the static encoding of the Morse code trie:



Dots traverse left, dashes right, terminals emit the character at the current node (terminal state). Stopping on red nodes, or attempting to take an unlisted edge is an error (invalid input).

Each node in the trie is a byte in the table. Dot and dash each have a bit indicating if their edge exists. The remaining bits index into a 1-based character table (at the end of `t`), and a 0 "index" indicates an empty (red) node. The nodes themselves are laid out as a binary heap in an array: the left and right children of the node at `i` are found at `i*2+1` and `i*2+2`. No need to waste memory storing edges!

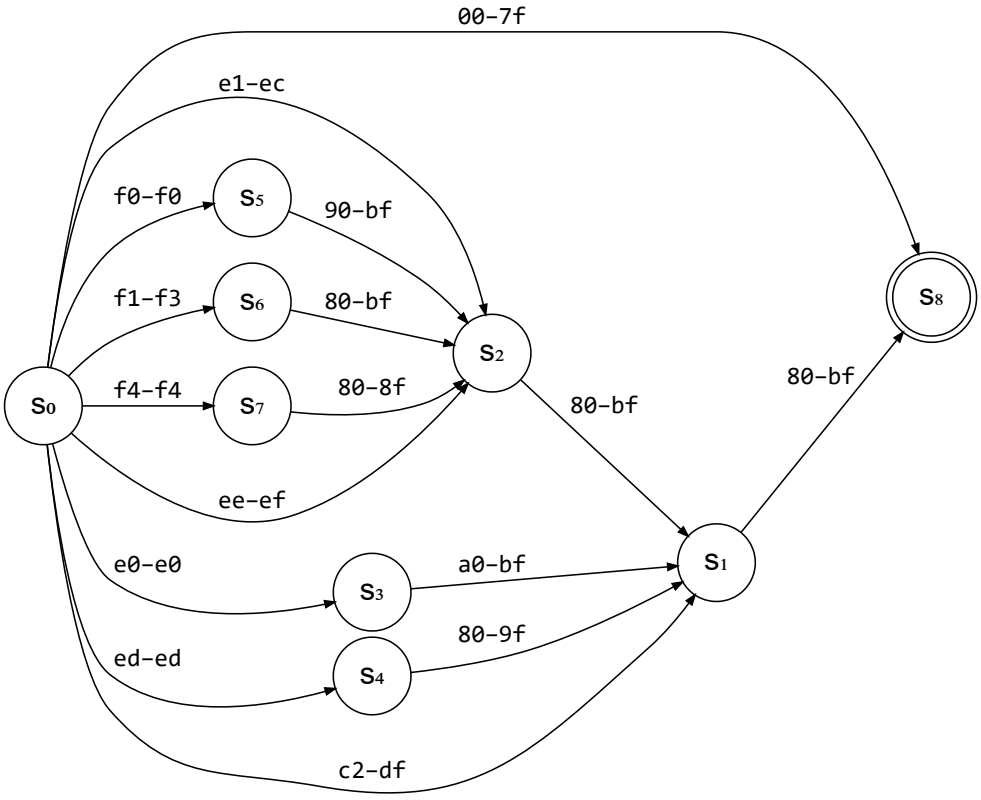
Since C sadly does not have multiple return values, I'm using the sign bit of the return value to create a kind of sum type. A negative return value is a state — which is why the state is negated internally before use. A positive result is a character output. If zero, the input was invalid. Only the initial state is non-negative (zero), which is fine since it's, by definition, not possible to traverse to the initial state. No `c` input will produce a bad state.

In the original problem the terminals were missing. Despite being a *state machine*, `morse_decode` is a pure function. The caller can save their position in the trie by saving the state integer and trying different inputs from that state.

UTF-8 decoder state machine

The classic UTF-8 decoder state machine is Bjoern Hoehrmann's Flexible and Economical UTF-8 Decoder. It packs the entire state machine into a relatively small table using clever tricks. It's easily my favorite UTF-8 decoder.

I wanted to try my own hand at it, so I re-derived the same canonical UTF-8 automaton:



Then I encoded this diagram directly into a much larger (2,064-byte), less elegant table, too large to display inline here:

https://github.com/skeeto/scratch/blob/master/parsers/utf8_decode.c

However, the trade-off is that the executable code is smaller, faster, and branchless again (by accident, I swear!):

```
int utf8_decode(int state, long *cp, int byte)
{
    static const signed char table[8][256] = { /* ... */ };
    static const unsigned char masks[2][8] = { /* ... */ };
    int next = table[state][byte];
    *cp = (*cp << 6) | (byte & masks[!state][next&7]);
    return next;
}
```

Like Bjoern's decoder, there's a code point accumulator. The *real* state machine has 1,109,950 terminal states, and many more edges and nodes. The accumulator is an optimization to track exactly which edge was taken to which node without having to represent such a monstrosity.

Despite the huge table I'm pretty happy with it.

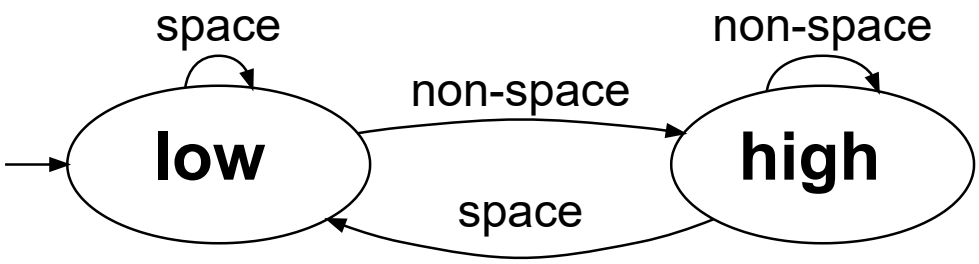
Word count state machine

Here's another state machine I came up with awhile back for counting words one Unicode code point at a time while accounting for Unicode's various kinds of whitespace. If your input is bytes, then plug this into the above UTF-8 state

machine to convert bytes to code points! This one uses a switch instead of a lookup table since the table would be sparse (i.e. let the compiler figure it out).

```
/* State machine counting words in a sequence of code points.
 *
 * The current word count is the absolute value of the state, so
 * the initial state is zero. Code points are fed into the state
 * machine one at a time, each call returning the next state.
 */
long word_count(long state, long codepoint)
{
    switch (codepoint) {
    case 0x0009: case 0x000a: case 0x000b: case 0x000c: case 0x000d:
    case 0x0020: case 0x0085: case 0x00a0: case 0x1680: case 0x2000:
    case 0x2001: case 0x2002: case 0x2003: case 0x2004: case 0x2005:
    case 0x2006: case 0x2007: case 0x2008: case 0x2009: case 0x200a:
    case 0x2028: case 0x2029: case 0x202f: case 0x205f: case 0x3000:
        return state < 0 ? -state : state;
    default:
        return state < 0 ? state : -1 - state;
    }
}
```

I'm particularly happy with the *edge-triggered* state transition mechanism. The sign of the state tracks whether the "signal" is "high" (inside of a word) or "low" (outside of a word), and so it counts rising edges.



The counter is not *technically* part of the state machine — though it eventually overflows for practical reasons, it isn't really "finite" — but is rather an external count of the times the state machine transitions from low to high, which is the actual, useful output.

Reader challenge: Find a slick, efficient way to encode all those code points as a table rather than rely on whatever the compiler generates for the `switch` (chain of branches, jump table?).

Coroutines and generators as state machines

In languages that support them, state machines can be implemented using coroutines, including generators. I do particularly like the idea of compiler-synthesized coroutines as state machines, though this is a rare treat. The state is implicit in the coroutine at each yield, so the programmer doesn't have to manage it explicitly. (Though often that explicit control is powerful!)

Unfortunately in practice it always feels clunky. The following implements the word count state machine (albeit in a rather un-Pythonic way). The generator returns the current count and is continued by sending it another code point:

```
WHITESPACE = {
    0x0009, 0x000a, 0x000b, 0x000c, 0x000d,
    0x0020, 0x0085, 0x00a0, 0x1680, 0x2000,
    0x2001, 0x2002, 0x2003, 0x2004, 0x2005,
    0x2006, 0x2007, 0x2008, 0x2009, 0x200a,
    0x2028, 0x2029, 0x202f, 0x205f, 0x3000,
}

def wordcount():
    count = 0
    while True:
        while True:
            # low signal
            codepoint = yield count
            if codepoint not in WHITESPACE:
                count += 1
                break
        while True:
            # high signal
            codepoint = yield count
            if codepoint in WHITESPACE:
                break
```

However, the generator ceremony dominates the interface, so you'd probably want to wrap it in something nicer — at which point there's really no reason to use the generator in the first place:

```
wc = wordcount()
next(wc)  # prime the generator
wc.send(ord('A'))  # => 1
wc.send(ord(' '))  # => 1
wc.send(ord('B'))  # => 2
wc.send(ord(' '))  # => 2
```

Same idea in Lua, which famously has full coroutines:

```
local WHITESPACE = {
    [0x0009]=true,[0x000a]=true,[0x000b]=true,[0x000c]=true,
    [0x000d]=true,[0x0020]=true,[0x0085]=true,[0x00a0]=true,
    [0x1680]=true,[0x2000]=true,[0x2001]=true,[0x2002]=true,
    [0x2003]=true,[0x2004]=true,[0x2005]=true,[0x2006]=true,
    [0x2007]=true,[0x2008]=true,[0x2009]=true,[0x200a]=true,
    [0x2028]=true,[0x2029]=true,[0x202f]=true,[0x205f]=true,
    [0x3000]=true
}

function wordcount()
    local count = 0
    while true do
        while true do
            -- low signal
            local codepoint = coroutine.yield(count)
            if not WHITESPACE[codepoint] then
                count = count + 1
                break
            end
        end
        while true do
            -- high signal
            local codepoint = coroutine.yield(count)
            if WHITESPACE[codepoint] then
                break
            end
        end
    end
end
```

Except for initially priming the coroutine, at least `coroutine.wrap()` hides the fact that it's a coroutine.

```
wc = coroutine.wrap(wordcount)
wc()  -- prime the coroutine
wc(string.byte('A'))  -- => 1
wc(string.byte(' '))  -- => 1
wc(string.byte('B'))  -- => 2
wc(string.byte(' '))  -- => 2
```

Extra examples

Finally, a couple more examples not worth describing in detail here. First a Unicode case folding state machine:

https://github.com/skeeto/scratch/blob/master/misc/casefold.c

It's just an interface to do a lookup into the official case folding table. It was an experiment, and I *probably* wouldn't use it in a real program.

Second, I've mentioned my UTF-7 encoder and decoder before. It's not obvious from the interface, but internally it's just a state machine for both encoder and decoder, which is what it allows it to "pause" between any pair of input/output bytes.