Конечные автоматы — замечательные инструменты

31 декабря 2020 г.
nullprogram.com/blog/2020/12/31/

Эта статья обсуждалась на Hacker News .

Мне нравится, когда мою текущую проблему можно решить с помощью конечного автомата. Их интересно проектировать и реализовывать, и я очень уверен в правильности. Они, как правило:

1. Представляем минималистичные, аккуратные интерфейсы
2. Требуют немного фиксированных ресурсов
3. Не придерживайтесь мнения о входе и выходе
4. Иметь компактную, лаконичную реализацию
5. Быть легким для рассуждения

Конечные автоматы, возможно, являются одной из тех концепций, о которых вы слышали в колледже, но никогда не применяли на практике. Возможно, вы используете их регулярно. Независимо от этого, вы определенно сталкиваетесь с ними регулярно, от регулярных выражений до светофоров.

Конечный автомат декодера азбуки Морзе

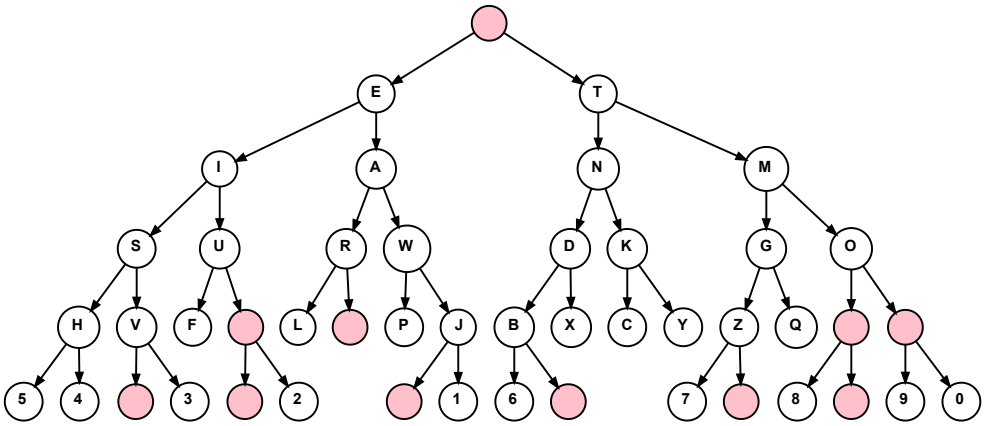
Вдохновленный головоломкой , я придумал этот детерминированный конечный автомат для декодирования кода Морзе . Он принимает точку (`'.'`), тире (`'-'`) или терминатор (0) по одному за раз, продвигаясь по конечному автомату шаг за шагом:

```
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;
    }
}
```

Обычно он компилируется до размера менее 200 байт (включая таблицу), требует всего несколько байт памяти для работы и подойдет даже для самых маленьких микроконтроллеров. Полный исходный листинг, документация и комплексный набор тестов:

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

Конечный автомат имеет форму префикса, а 100-байтовая таблица `t`представляет собой статическое кодирование префикса азбуки Морзе :



Точки перемещаются влево, тире вправо, терминалы выдают символ в текущем узле (состояние терминала). Остановка на красных узлах или попытка взять неуказанное ребро является ошибкой (недопустимый ввод).

Каждый узел в trie — это байт в таблице. У каждой точки и тире есть бит, указывающий, существует ли их ребро. Оставшиеся биты индексируют таблицу символов с 1-й базой (в конце `t`), а «индекс» 0 указывает на пустой (красный) узел. Сами узлы располагаются в виде двоичной кучи в массиве : левые и правые потомки узла в `i`находятся в `i*2+1`и `i*2+2`. Не нужно тратить память на хранение ребер !

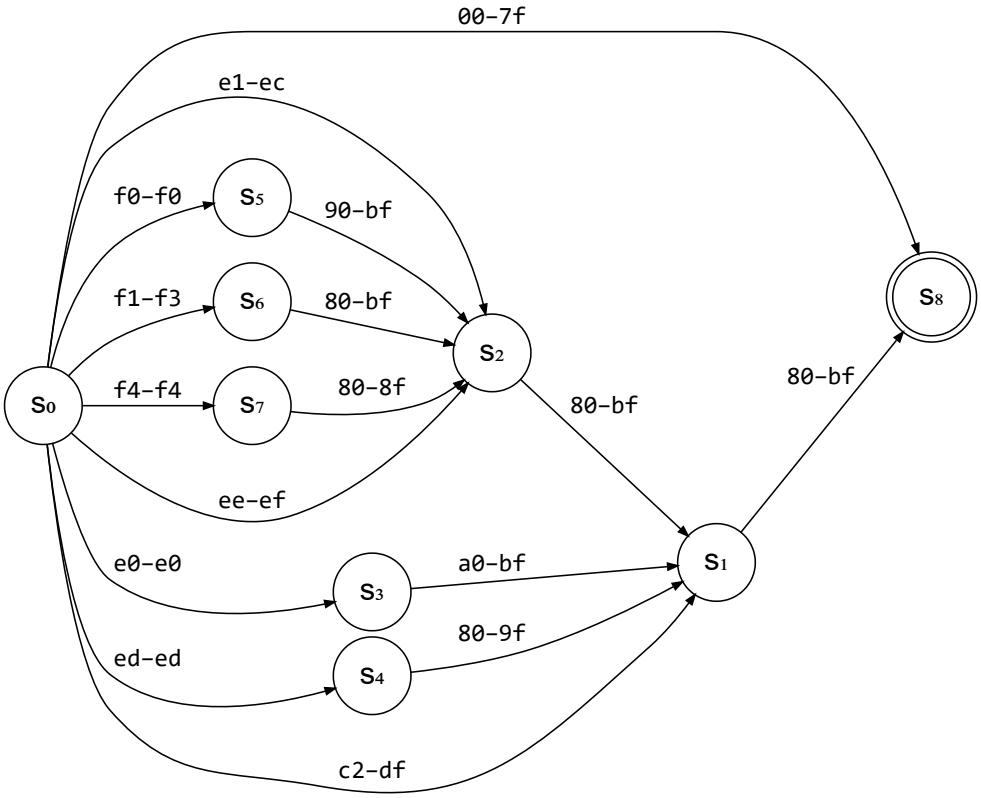
Поскольку в C, к сожалению, нет множественных возвращаемых значений, я использую знаковый бит возвращаемого значения для создания своего рода типа суммы. Отрицательное возвращаемое значение — это состояние, поэтому состояние внутренне инвертируется перед использованием. Положительный результат — это символьный вывод. Если ноль, ввод был недействительным. Только начальное состояние неотрицательно (ноль), что нормально, поскольку по определению невозможно перейти к начальному состоянию. Никакой `c`ввод не приведет к плохому состоянию.

В исходной задаче терминалы отсутствовали. Несмотря на то, что это *конечный автомат* , `morse_decode`это чистая функция. Вызывающий может сохранить свою позицию в trie, сохранив целое число состояния и попробовав разные входы из этого состояния.

Конечный автомат декодера UTF-8

Классический декодер UTF-8 — это гибкий и экономичный декодер UTF-8 Бьорна Хёрмана . Он упаковывает весь конечный автомат в относительно небольшую таблицу с помощью хитрых трюков. Это, несомненно, мой любимый декодер UTF-8.

Я хотел попробовать сделать это сам, поэтому я перевывел тот же канонический автомат UTF-8:



Затем я закодировал эту диаграмму непосредственно в гораздо большую (2064 байта), менее элегантную таблицу, слишком большую, чтобы отобразить ее здесь в строке:

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

Однако компромисс заключается в том, что исполняемый код становится меньше, быстрее и снова не имеет ветвлений (случайно, клянусь!):

```
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;
}
```

Как и декодер Бьорна, есть аккумулятор кодовых точек. *Реальный* конечный автомат имеет 1 109 950 конечных состояний и гораздо больше ребер и узлов. Аккумулятор — это оптимизация для отслеживания того, какое ребро было взято в какой узел, без необходимости представлять такое чудовище.

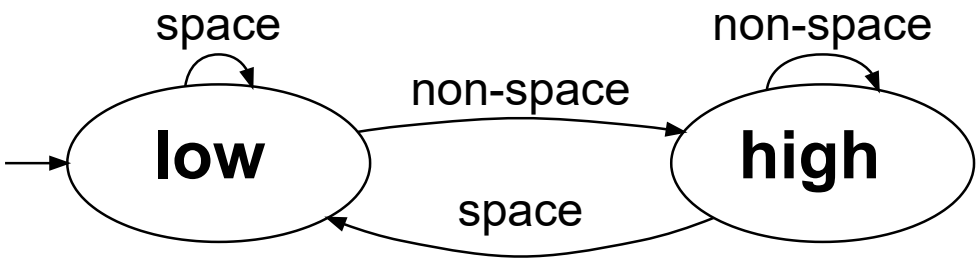
Несмотря на огромный стол, я им очень доволен.

Конечный автомат подсчета слов

Вот еще один конечный автомат, который я придумал некоторое время назад для подсчета слов по одной кодовой точке Unicode за раз, при этом учитывая различные виды пробелов Unicode. Если ваши входные данные — байты, то вставьте это в указанный выше конечный автомат UTF-8, чтобы преобразовать байты в кодовые точки! Этот использует переключатель вместо таблицы поиска, поскольку таблица будет разреженной (т. е. пусть компилятор сам разберется).

```
/* 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;
    }
}
```

Я особенно доволен механизмом перехода состояний, *активируемым фронтом* . Знак состояния отслеживает, является ли «сигнал» «высоким» (внутри слова) или «низким» (вне слова), и поэтому он учитывает нарастающие фронты.



Счетчик *технически* не является частью конечного автомата (хотя в конечном итоге он переполняется по практическим причинам, на самом деле он не «конечен»), а скорее представляет собой внешний счетчик переходов конечного автомата из низкого состояния в высокое, что является фактическим полезным выводом.

Задача читателя : найти простой и эффективный способ закодировать все эти кодовые точки в виде таблицы, а не полагаться на то, что сгенерирует компилятор `switch`(цепочка ветвлений, таблица переходов?).

Корутины и генераторы как конечные автоматы

В языках, которые их поддерживают, конечные автоматы могут быть реализованы с использованием сопрограмм, включая генераторы. Мне особенно нравится идея синтезированных компилятором сопрограмм в качестве конечных автоматов, хотя это редкое удовольствие. Состояние подразумевается в сопрограмме при каждом yield, поэтому программисту не нужно управлять им явно. (Хотя часто этот явный контроль является мощным!)

К сожалению, на практике это всегда кажется неуклюжим. Следующий код реализует конечный автомат подсчета слов (хотя и не совсем в стиле Python). Генератор возвращает текущий подсчет и продолжается путем отправки ему другой кодовой точки:

```
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
```

Однако церемония генератора доминирует в интерфейсе, поэтому вы, вероятно, захотите обернуть ее во что-то более красивое — в таком случае, на самом деле, нет смысла использовать генератор в первую очередь:

```
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
```

Та же идея в Lua, который, как известно, имеет полные сопрограммы:

```
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
```

За исключением первоначальной подготовки сопрограммы, по крайней мере `coroutine.wrap()` скрывает тот факт, что это сопрограмма.

```
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
```

Дополнительные примеры

Наконец, еще пара примеров, которые не стоит здесь подробно описывать. Сначала машина состояний сворачивания регистра Unicode:

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

Это просто интерфейс для поиска в официальной таблице складных корпусов . Это был эксперимент, и я, *вероятно,* не буду использовать его в реальной программе.

Во-вторых, я уже упоминал свой кодер и декодер UTF-7 . Это не очевидно из интерфейса, но внутренне это просто конечный автомат для кодера и декодера, что и позволяет ему «делать паузу» между любой парой байтов ввода/вывода.