21:14 Counting words with a state machine.

by Robert Graham

In this paper we implement `wc`, the classic Unix word count program, using an asynchronous state machine parser. We implement this twice: first a simplified version supporting ASCII, then a more complete program supporting Unicode UTF-8 encoding. We implement this algorithm in both C and JavaScript. Even the latter is significantly faster than the standard versions of `wc`, such as the GNU Coreutils `wc` that comes with Linux.

Introduction

A parser is software that translates external data into some internal data structures.

At university, they teach you abstract and formal parsers, often in a class that builds a compiler. However, little of that theory is used elsewhere in your coursework. In your networking class, the code they teach uses concrete and ad-hoc parsers, discarding everything you learned in your parser class. While parser theory they teach you is useful, even academics struggle to use it in practice.

In this paper, we do a mix of theory and practice. On one hand, we look at abstract theory of state-machines and deterministic/non-deterministic finite automata. On the other hand, we build the state-machine by hand, banging the bytes together.

The reason this concept is important is demonstrated by Nginx replacing Apache as the dominant web server on the Internet. Apache parses input the legacy way they taught you in networking class. The newer Nginx parses input using a state-machine. This parsing is more scalable, allowing much higher loads on the web server.[39]

In this paper, we demonstrate state-machines by re-implementing the classic Unix command-line program wc. Over the last year, it has been popular for proponents of various languages to re-implement `wc` in order to show that their favorite language can compete with C in performance. In this case, we do this to demonstrate our favorite algorithm is better than existing algorithms, implementing it in two different languages.

These re-implementations are usually incomplete, only parsing ASCII. In this paper, we do a more complete version, correctly parsing UTF-8.

The intent of this article isn't that you should go and parse everything with state-machines. It puts a burden on future programmers trying to read the code, most of whom are unfamiliar with the technique. On the other hand, when performance and scalability are needed, state-machines are a good choice. You probably wouldn't want to use them for `wc` in the real world, as the program doesn't need to be especially fast. We choose `wc` in this paper only because it's a popular benchmark target, the simple thing that more complex endeavors are compared against.

What is WC?

This command-line utility has been part of Unix since time began on the first of January, 1970. As defined in the POSIX standard, it counts the number of lines, words and characters, when the corresponding flag of `-l`, `-w`, and `-c` is set. If no parameters are set, then the default is all three, `-lwc`.

```
$ echo "basic input/output" | wc
       1       2      19
```

We see here that the program has reported one line, two words, and 19 characters. Words are counted by the number of strings of non-spaces separated by spaces. Thus, this example is only two words, not three.

Modern character encodings can use multiple bytes per character, such as UTF-8 or various character sets for Chinese, Japanese, and Korean. In such cases, the `-m` parameter replaces the `-c` parameter, counting the number of multi-byte characters instead of the number of bytes. As we see in these two examples, changing from `-c` to `-m` changes the character count:

```
$ echo わたしは にほんごがすこししか はなせません | wc −lwc
       1       3      67

$ echo わたしは にほんごがすこししか はなせません | wc −lwm
       1       3      23
```

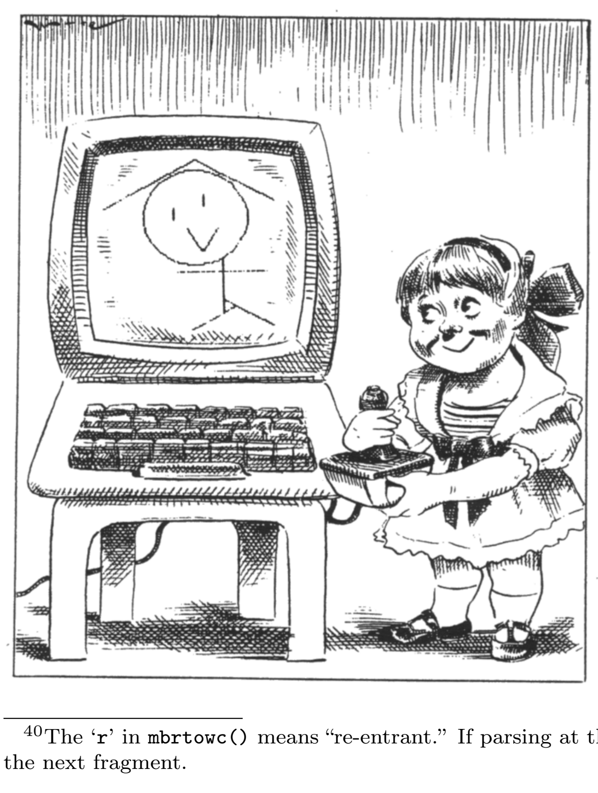
[39] This overstates the importance of just the parsing. Nginx scales better than Apache for a lot of reasons. However, these reasons are all interconnected: if you write an asynchronous server, then state-machine parsers are a much better way of parsing the requests.

How do they implement WC?

There are many versions of the program, such as GNU's Coreutils for Linux, BusyBox, macOS, FreeBSD, OpenBSD, QNX and SunOS. Most implementations count words by counting the number of times a space is followed by a non-space. Think of it as an edge-triggered condition, going from space to non-space. As we'll soon see, this can also be treated as a state-machine with two states.

Parsing words is easy, the hard part is character-sets. We could hard-code ASCII values into our program, such as `0x20` for space and `0x0A` for newline, but this wouldn't work for non-ASCII systems. IBM mainframes that use the EBCDIC character-set will represent a space using `0x40`.

Thus, instead of using hard-coded values these programs use the standard `isspace()` function to test if a character is a space. Recently, many people have re-implemented `wc` in their favorite language to show that they can be just as fast as C. In fact, most of the processing time is spent in the `isspace()` function, so all they really proved is that hard-coded constants like `0x20` in other languages are faster than `isspace()` function calls in C.



The problem is worse for multi-byte character-sets like UTF-8. The program must first parse multiple bytes into wide characters using functions like `mbtowc()` (or `mbrtowc()`),[40] then test if they are a space with `iswspace()`. Re-implementations often do only ASCII. This paper includes two re-implementations, the first for ASCII, the second for UTF-8.

How do we implement it?

Our first version supporting ASCII is shown in Figure 17. In the GitHub project accompanying this article, the program is `wc2o.c`, where the '`o`' stands for "obfuscated C version." This program is pretty darn opaque when trying to figure out how it counts words. On the other hand, it exposes the idea of state-machine parsing.

Line 5 declares the state-machine table consisting of four states. Each state is a row of three transitions. (Table 1)

Line 7 declares a table that will translate bytes. All 256 ASCII values translate into one of three possible values: `word(0)`, `space(1)`, and `newline(2)`. Specifically, the character `0x0A` or '`\n`' translates to `newline(2)`, and the characters '`\b\t\m\v\f`' translate to `space(1)`. All other values translate to `word(0)`. The reason we include this translation step is that so that the state-machine on line 5 is 4×3 states rather than 4×256 states. In our final version, we don't do this translation, and just have large state-machines instead.

Line 15 loops getting the next byte of input, one byte at a time. Calling `getchar()` here for every character is potentially expensive, but we aren't benchmarking this program, just showing the algorithm. In our final version, we read input a buffer at a time instead of a byte at a time.

Line 16 does the state transition, in other words, it parses the input. We translate the byte into one of the three column values, `word(0)`, `space(1)`, or `newline(2)`. We look up that in the current row, then set the next row according to the transition. Thus how in the `was-space(0)` state, if we receive a `non-space(0)` character, we transition to `new-word(2)` state.

Line 17 processes what we parsed. In our case, the processing is trivialized to just counting the number of times we visit each state.

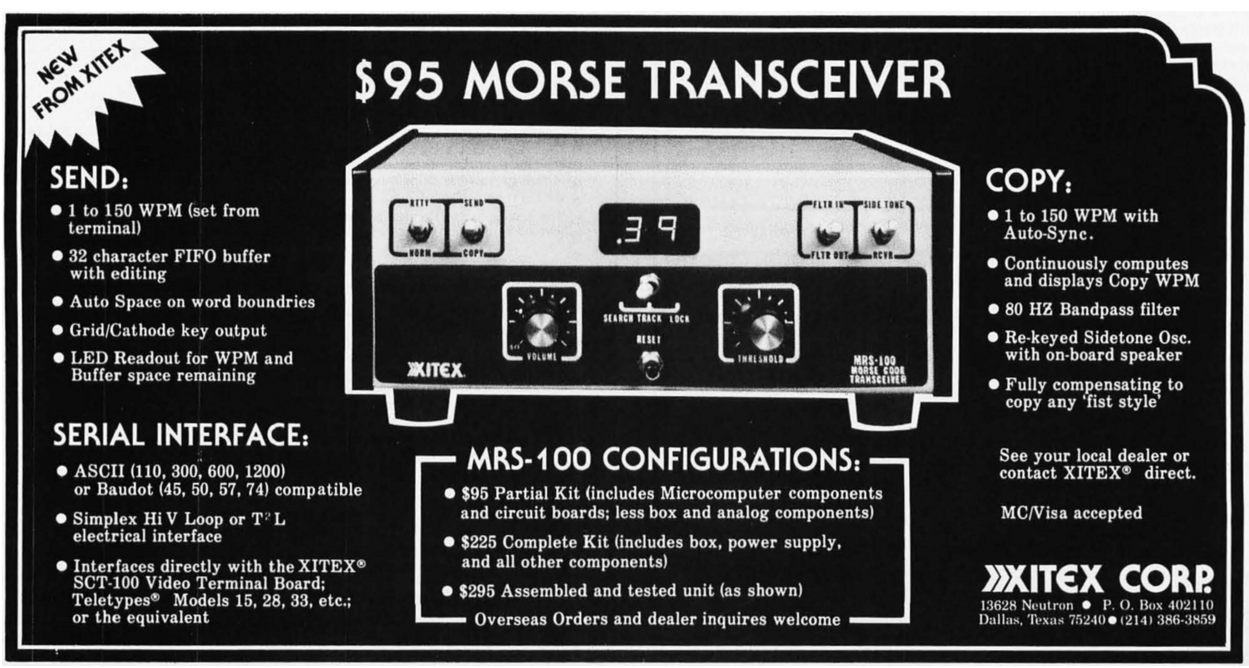
[40] The '`r`' in `mbrtowc()` means "re-entrant." If parsing at the end of a fragment, it saves state before resuming at the start of the next fragment.

```c
#include <stdio.h>
int main(void)
{
    static const unsigned char table[4][3] = {
        {2,0,1}, {2,0,1}, {3,0,1},  {3,0,1}
    };
    static const unsigned char column[256] = {
        0,0,0,0,0,0,0,0,0,1,2,1,1,1,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,
    };

    int state = 0;
    int c;

    while ((c = getchar()) != EOF) {
        state = table[state][column[c]];
        counts[state]++;
    }
    printf("%lu %lu %lu\n", counts[1], counts[2],
           counts[0] + counts[1] + counts[2] + counts[3]);
    return 0;
}
```

Figure 17: `wc2o.c`, an obfuscated word counter for ASCII.

	`word(0)`	`space(1)`	`newline(2)`
`was-space(0)`	`new-word(2)`	`was-space(0)`	`new-line(1)`
`new-line(1)`	`new-word(2)`	`was-space(0)`	`new-line(1)`
`new-word(2)`	`was-word(3)`	`was-space(0)`	`new-line(1)`
`was-word(3)`	`was-word(3)`	`was-space(0)`	`new-line(1)`

Table 1: Simple word-count state machine.



Line 19 prints the results. The number of lines is the number of times we visited the `new-line(1)` state. The number of words is the number of times we visited the `new-word(2)` state. The number of characters is the number of times we visited all the states combined.

Consider reading input whose only letter is 'x'. We start with `state==was-space(0)`. The 'x' is them translated into a column value, `word(0)==column['x']`. The new state becomes `new-word(2) == table[was-space(0)][word(0)]`.

The resulting program produces the same output as the built-in program.

```
$ wc < wc2o.c
 26   74  649

$ ./wc2o < wc2o.c
 26   74  649
```

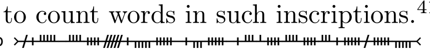
What about Unicode and UTF8?

This program does hard-coded ASCII, a single-byte character set. We need something that can handle multi-byte character-sets, like Unicode and UTF-8.

Unicode is a character-set; UTF-8 is an encoding. Unicode characters, or wide characters or code points, are integers between `0` and `0x100FFFF`. Each code point can be represented by from one to four bytes, as shown in Table 2.

Consider the character `U+1680`, Ogham Space Mark. According to the table, this is encoded as the three-byte sequence `0xE1 0x9A 0x80`.

Ogham is an alphabet from the sixth century for writing Old Irish that survives today as roughly 400 inscriptions on monuments and gravestones. A compliant version of `wc` must count spaces on those monuments. The space mark counts as a space character according to `iswpace(0x1680)`. Thus, we should be able to count words in such inscriptions.[41]

`$ echo`  `| wc -lwm`

`1 4 30`

There are about thirty such Unicode space characters for which `iswspace()` will return true. The libraries for Windows, Linux, and macOS have slight disagreements about this, so what some will recognize as a space will not be recognized as a space on others. However, they all agree that `U+1680` is a space.

There are many invalid UTF-8 sequences. Among those are unnecessarily long, redundant sequences. The table suggests that `0x0A` may also be represented as `0xC0 0x8A` and `0xE0 0x80 0x8A`. Since this can be encoded as simply `0x0A`, the longer sequences are declared to be officially invalid and must be rejected.

Thus, a parser for Unicode must not only consider the basic math as shown in the table, but also recognize spaces and reject invalid sequences.

How do state-machine parsers work?

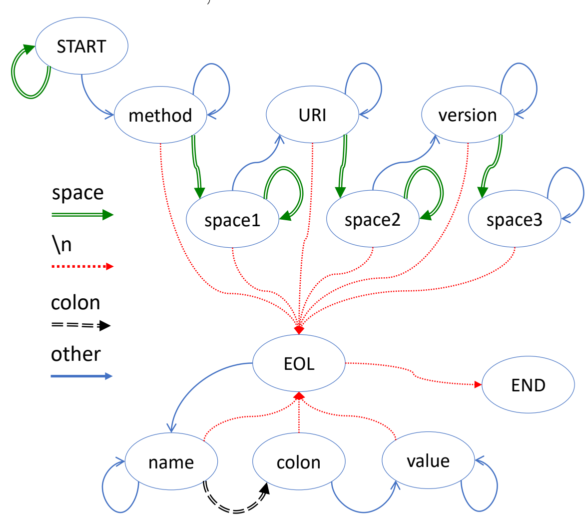
You are familiar with state-machines in other parts of computer-science, such as the famous TCP/IP state-machines. In those state-machines, some sort of event happens that causes a transition from one state to another. For parsers, the event is the next byte of input. Each byte of input is read sequentially, and depending up that byte's value, a transition happens from one state to another.

There are two ways to represent these transitions: either through a big lookup table as in `wc2o.c` on page 73, or with a `switch`/`case` block.

Consider HTTP. A request header looks like the following:

```
GET /index.html HTTP/1.0
Host: www.google.com
User-Agent: Mozilla (actually Chrome)
```

A state-machine that parses it might assign a state to each field, like this.



41 If the editors have done their job right, you should be able to copy/paste this from the online PDF document and reproduce these results. It works on macOS, Linux, and on Windows using the PowerShell `Measure-Object` commandlet, when the locale is set to Unicode. —Rob

Scalar Unicode Value	First Byte	Second	Third	Fourth
00000000 00000000 0xxxxxxx	0xxxxxxx			
00000000 00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
00000000 zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 2: UTF-8 Bit Distribution, Unicode 6.0

As we receive the bytes of an HTTP request, we enter the `method` state the first time we receive a non-space character. We remain in the `method` state until we receive a space character, at which point we transition to the `space1` state.

In C, we might process each byte of input with a function like the following `switch`/`case` logic:

```
int http_parse(int state, unsigned char c,
                ...){
  switch (state) {
    ...
  case METHOD: /*GET, POST, HEAD, ... */
    if (c == '\n') {
      ...
      return EOL;
    } else if (isspace(c)) {
      ...
      return SPACE1;
    } else {
      ...
      return state; /* no change in state */
    }
    ...
  }
}
```

Most major web servers that aren't Apache use this method. Nginx calls this state `sw_method`, which you can see in the open-source online.[42]

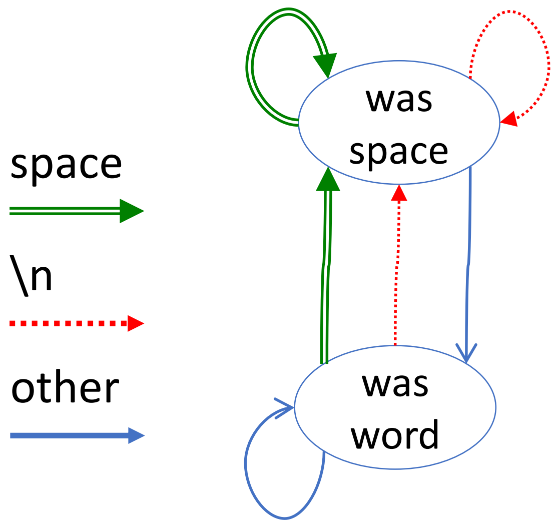
You can test on a live network whether a web server is parsing requests using a state-machine. Send a request to the server consisting of `GET`, followed by five billion spaces and only then the rest of the request. If the server acts like Apache buffering a complete header, then it'll run out of buffer space. If instead the server acts like Nginx and parses input with a state machine, it'll happily keep reading spaces as long as it's in that state. If the connection terminates prematurely, it'll be because of a timeout instead of running out of buffers. (It takes a while to send five gigabytes.)

[42] See near line 159 of `ngx_http_parse.c`.

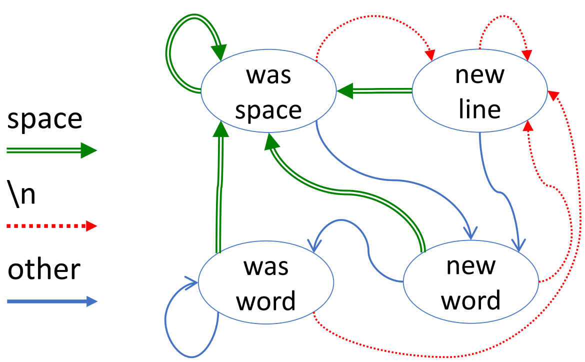
This example uses a switch/case block of code to handle the transitions. In our state-machines for counting words, we use a lookup table instead. A third choice is to use a mixture. The program `masscan`, for example, does a lot of parsing of such protocols like FTP, SMTP, X.509, and so. It uses a mixture of `switch` statements and lookup tables.

How can we construct a state-machine for word-counting?

Most implementations of `wc` effectively use a machine with two states which can be represented with the following diagram. Note that they aren't designed explicitly as a state-machine, but that's effectively how the code works.



In the `wc2o.c` program, we changed this to a machine with four states. This is the table with three types of transitions and four states:

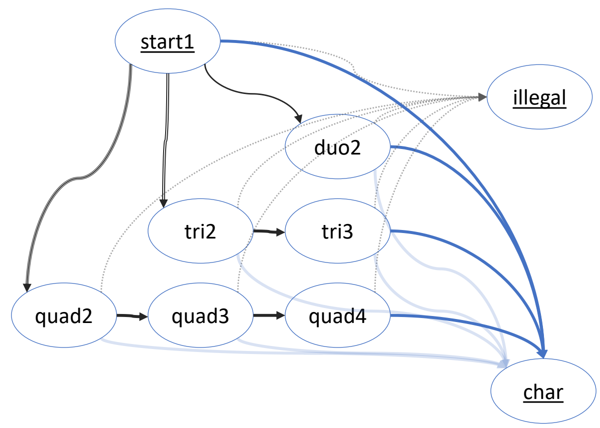


We did this in order to be overly clever in how we were going to process the data.

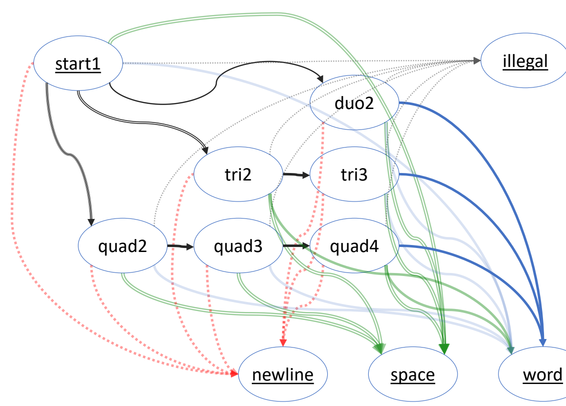
Remember, there's two things going on here. One step is parsing the input. The next step is processing the results. Thus, we first need to parse out things like words, characters, and newlines. Then we need to process this information, which for word-counting, is done by counting each time we enter a state. We've cleverly collapsed the processing into a simple operation.

The lesson here isn't that parsers can completely trivialize processing as we've done here, but instead that we often add artificial states to benefit later processing.

Now let's talk about UTF-8. Using the original table as a guide, we might construct a state-machine for parsing 1-byte sequences, 2-byte sequences called a "duo," 3-byte sequences "tri," and 4-byte sequences "quad."



However, our needs are simpler. We don't need to parse out the code point and test with `iswspace()` but can instead include that functionality within the state-machine itself, where the output is one of four values: `word`, `space`, `newline`, or `illegal`.



There are, in fact, more states than just this. Instead of a simple path for 3-byte characters, we must add additional states that recognize 3-byte characters that result in spaces. This creates a table of roughly thirty states that's too complex to draw here.

Instead, here are snippets of the code that take an existing table and adds states for characters like `U+1680` Ogham Space Mark. It clones existing states that follow the same path, but at the end marks the character as a `space` instead of a `word`:

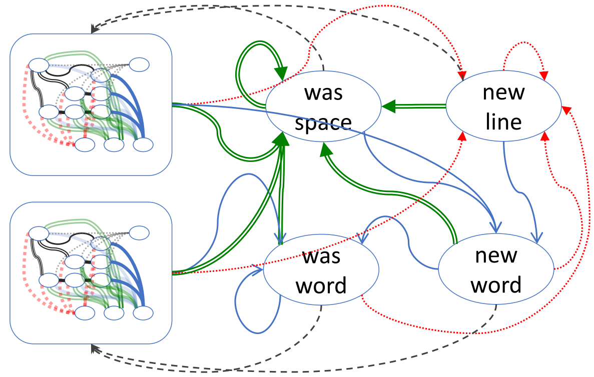
```
/* clone existing states */
memcpy(table[TRI2_E1], table[TRI2],
        sizeof(table[0]));
memcpy(table[TRI3_E1_9a], table[TRI3],
        sizeof(table[0]));
/* link in new states */
table[0][0xE1] = TRI2_E1;
table[TRI2_E1][0x9a] = TRI3_E1_9a;
table[TRI3_E1_9a][0x80] = SPACE;
```

What this code is doing is exactly what generic regex code would do. All we are doing here is creating manually what regex libraries would do based upon expressions. What we are doing here is manual optimization for concepts that exist abstractly.

Now let's combine our UTF-8 state-machine parser with our word-count state-machine parser. There's two ways of doing this. The obvious way is to feed the output of one as input to the other. The other way is to combine the two into a single state-machine.

This is a multiplicative process. That means replicating one state-machine for every state in the other state-machine. Again, let's talk regex theory. There are two ways of representing such a thing. One way increases computation, what we call an NFA or non-deterministic finite automata, which is what would happen if we fed the output of one as the input to the next. The other way keeps computation the same but increases the size of the table. This is a DFA or deterministic finite automata. As you build complex regexes, you cause either computation to explode or memory to explode. In this case, we've chosen DFA, so memory explodes.

Thus, where one state-machine needs 35 states and the other just four, that means the combination may needs as many as $4 \times 35 = 140$ states. However, we are going to do a small trick. The `was-space` and `new-line` states are clones of each other, as are `was-word` and `new-word`. Thus, we only need to double rather than quadruple the UTF-8 state-machine. This produces something that may be represented like:



The Final Code

The final code is in `wc2.c`. It's a few hundred lines so is not included in this article but is instead available on GitHub.[43]

The complicated part that takes hundreds of lines is where it builds that state-machine table. This results in a table roughly with 70 states (rows), and 256 columns, where each column represents the transition that will happen when a byte of input is received.

Once we've built the table, we simply process chunks of input analogous to the following. The actual code looks slightly different, with the inner loop separated into a `parse_chunk()` function.

```
unsigned counts[MAX_STATE];
//Get the next chunk of input.
length = fread(buf, 1, sizeof(buf), fp);
//For all bytes in that chunk,
for (i=0; i<length; i++) {
    //Get the next byte.
    c = buf[i];
    //Do the state transition.
    state = table[state][c];
    //Do the counting.
    counts[state]++;
}
//Report the results.
word_count = counts[NEW_WORD];
line_count = counts[LINE_COUNT];
char_count = counts[0] + counts[1]
           + counts[2] + counts[3];
```

[43] `git clone https://github.com/robertdavidgraham/wc2 || unzip pocorgtfo21.pdf wc2.zip`

Benchmarks

For benchmarks, I started with the file `pocorgtfo-18.pdf`.[44] This is big (92-million bytes), but also has the nice property of being unfriendly to parsers.

However, this turned out to be a bad choice, or at least an awkward one. Illegal characters cause performance problems in the `mbtowc()` and `iswspace()` functions at the heart of existing programs. There were also big performance differences with legal text, depending upon whether it was ASCII or Unicode, random letters/spaces, all spaces, or all non-spaces.

To better understand the existing `wc` in GNU Coreutils, I benchmarked a bunch of 92-million-byte files. The files are:

A random sequence of UTF-8 non-spaces and spaces, `utf8.txt`. A random sequence of spaces and non-spaces in 7-bit clean ASCII, `ascii.txt`. The letter `x` repeated 92 million times, `word.txt`. The space character repeated 92 million times, `space.txt`.

The command-line utility time was used, using the userland time, in seconds.

Filename	UTF-8	ASCII
`pocorgtfo18.pdf`	5.171	1.104
`utf8.txt`	2.257	0.765
`ascii.txt`	2.280	1.098
`word.txt`	0.712	0.643
`space.txt`	0.499	0.424

We see a roughly ten-fold performance difference of the existing GNU `wc` depending upon input. Parsing a bunch of illegal characters as Unicode takes 5.17 seconds, but parsing a file containing only ASCII space characters takes 0.42 seconds.

Now for our program. We've written two versions, `wc2.c` and `wc2.js`, in C and JavaScript respectively. Comparing our results for just the UTF-8 mode, we see that both state machine implementations are faster than the original.

Filename	GNU `wc`	`wc2.c`	`wc2.js`
`pocorgtfo18.pdf`	5.172	0.145	0.501
`utf8.txt`	2.277	0.142	0.502
`word.txt`	0.716	0.139	0.496
`space.txt`	0.498	0.142	0.501

The first property of the `wc2` programs is that they are constant time, regardless of the type of input. The slight variations in time are due to inaccuracies using time as a benchmark tool.

The second property of the programs is that they are faster. Even at its fastest, the GNU program is over 3 times slower than our program. At roughly 0.5 seconds, even our JavaScript program matches the GNU program at its fastest. Given worst case input, our program is twenty-five times as fast.

What this shows is that state-machine parsers tend to be both fast, but also robust when given illegal input. When you look at what `mbrtowc()` and `iswspace()` must do in order to guard against malicious input, you'll see that the code is quite dangerous. In contrast, how the state-machine parses malicious input is inherently safe.

Finally, just to make sure our UTF-8 parsing is correct, we see that it produces the same results as before, finding four words and 30 characters, given 88 bytes of input:

```
$ echo ᚛ᚋᚐᚉ ᚈᚓᚎᚐᚅᚓ ᚐᚃᚓ ᚐᚈᚆᚒᚉᚒᚈᚐᚓᚋᚓᚅ᚜"" | ./wc2 -lwm
1 4 30
```

What does asynchronous mean?

We've talked a lot about state machines but not what it means for them to be asynchronous. Asynchronous means that reading input is completely independent of parsing, that the parser doesn't influence how input is read.

To understand the difference between asynchronous and traditional methods, consider other implementations of `wc`. They often read input using a `getword()` or `getline()` function.[45] This combines some parsing with reading input. In other words, instead of reading input in a fixed manner, like 64k buffers, the amount read depends upon parsing. Conversely, how the parser is constructed depends upon how input is read. Each influences the other.

Let's say that we want to write a version that can word-count thousands of files at once, simultaneously. Using the traditional method of combining reading with parsing, you'd have to spawn thousands of threads. Using the asynchronous technique, you can use a single thread. Using AIO APIs, the operating system will deliver the next chunk of data as it arrives from the disk. AIO APIs read fixed

[44] There are two different versions of `pocorgtfo18.pdf` with a SHA1 of `191b636f80d0c74164ec9d9b3544decdaa2b7df5`. These experiments describe the version with an MD5 of `84c49ffee3fffebed5875a162e43bb1d`, not an MD5 of `f5879ccb9570ec8def41-c36854021b4e`.

[45] See OpenBSD's `wc.c`, near line 210.

sized blocks, like 64k; you can't choose the size of blocks depending upon the parsed contents. When data is received, it is dispatched to the appropriate copy of the state for parsing each file. We just need an 8-bit integer for every file to hold all the per file parser state.

This would be a silly thing to do with files but is an important thing for networking services. Apache is broken and can't scale beyond 10,000 concurrent TCP connections because it struggles with 10,000 threads in the system. All its major competitors use a single thread (or single thread per CPU core) and handle things asynchronously.

Again, the parser can't influence data reception. The network stack simply receives packets from the other side, whatever size of packets those might be. The parser must handle the case where it receives exactly as much data as it was expecting, or too much data, or not enough data.

Conclusion

There have been many posts over the last year of people implementing `wc` in their favorite language, such as Haskell or Python. In this paper, instead of a different programming language, we've chosen a fundamentally different algorithm, that of an asynchronous state-machine parser. We've implemented the same algorithm in both C and JavaScript, to show that the speed is property of the algorithm instead of the language.

Instead of a simplified problem of just handling ASCII, we've demonstrated the algorithm using the difficult problem of UTF-8 encodings. Given something bizarre like Ogham text, we still produce the same answer as compliant `wc` programs. While only the UTF-8 encoding is implemented, the concept extends to any character-set, including the CJK (Chinese, Japanese, Korean) multi-byte character-sets.

Such state-machine parsers are costly in terms of code maintainability: most programmers are unfamiliar with them. However, they have clear advantages for writing scalable, secure code for modern Internet applications.