Lab 5: Finite-State Machines

Finite state machines (FSMs) are an essential part of theoretical computer science. You learned about them in CptS 317 (Automata and Formal Languages).

In this lab, we'll see how FSMs can also be used to model and build useful programming systems. In particular, we'll use them to build a piece of software to compute useful statistics on a piece of C or C++ code.

This page will be available as

http://www.tricity.wsu.edu/~bobl/cpts360/lab04_fsm/writeup.html

Calling it up in a browser will allow you to cut-and-paste code in the following and save typing. You may also download whole files from the lab link on the course web page.

Every part after Part 1 depends on the part that went right before it, so be sure that part is working before you move on.

`codestats`

"Lines of code" is a common software metric. We talk about how many lines of code a particular project contains. But just what constitutes a line of code? Do blank lines count? Lines that are just comments? Comments are, in general, a good idea. We might want to measure them, too.

The `codestats` program will measure these metrics:

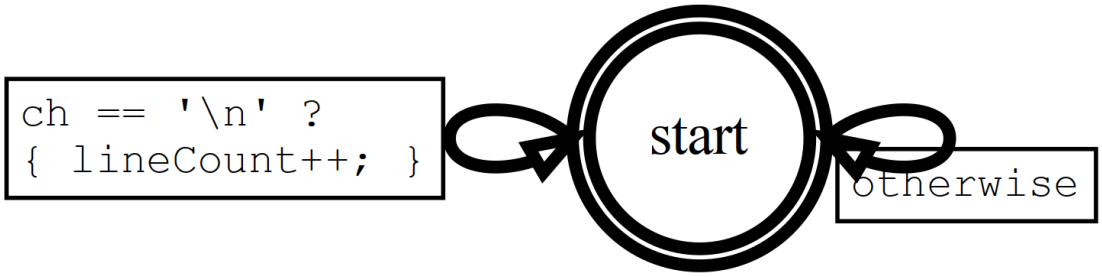
- the total number of lines in a file
- the number of lines that contain code
- the number of C++-style comments
- the number of C-style comments
- the number of C preprocessor directives

We'll use an FSM to find all of these and also to avoid being confused by single-quoted char constants, double-quoted strings, and C preprocessor commands.

We'll make a simplifying assumption about this: The code we examine will be legal C or C++: It should pass through the compiler without errors, although there might be some warnings.

Part 1: Counting Lines

We'll start with a simple FSM to count lines:



This illustrates the notation we'll be using throughout this lab:

- States are given meaningful names. Here, "start" is the only state.
- Most transitions are labeled with the input character or test that causes them, followed by a "?". Here, one transition takes place on a newline input.
- Any actions required by a transition are enclosed in curly braces.
- Usually one transition is labeled "otherwise" and is to be taken if the input character doesn't match something else.
- New states and transitions have thick lines. (In Part 1, everything is new.)

Here's the complete implementation of this FSM:

```
#include <stdio.h>
#include <assert.h> // for assert()

struct CodeStats {
    int lineCount;
};


void codeStats_init(struct CodeStats *codeStats)
{
    codeStats->lineCount = 0;
}


void codeStats_print(struct CodeStats codeStats, char *fileName)
{
    printf("           file: %s\n", fileName);
    printf("     line count: %d\n", codeStats.lineCount);
}


void codeStats_accumulate(struct CodeStats *codeStats, char *fileName)
{
    FILE *f = fopen(fileName, "r");
    int ch;
    enum {
        START,
    } state = START;

    assert(f);
    while ((ch = getc(f)) != EOF) {
        switch (state) {

        case START:
            if (ch == '\n') {
                codeStats->lineCount++;
            }
            break;
```

```
        default:
            assert(0);
            break;
        }
    }
    fclose(f);
    assert(state == START);
}


int main(int argc, char *argv[])
{
    struct CodeStats codeStats;
    int i;

    for (i = 1; i < argc; i++) {
        codeStats_init(&codeStats);
        codeStats_accumulate(&codeStats, argv[i]);
        codeStats_print(codeStats, argv[i]); // no "&" -- see why?
        if (i != argc-1)   // if this wasn't the last file ...
            printf("\n");  // ... print out a separating newline
    }
    return 0;
}
```

Put this in a file called "`codestats_pt1.c`". (You may download it from the lab link.) The code here is complete and needs no modification. All you need to do is create a `Makefile` for `codestats_pt1`. (This lab starts out easy!)

Study this simple example so you don't get lost when we start making modifications. Note in particular how the features of the FSM map to C code:

- Always begin in the "start" state.
- New statistics are:
    - declared as part of the `CodeStats typedef`.
    - set to zero in `codeStats_init()`
    - printed on a line `codeStats_print()`

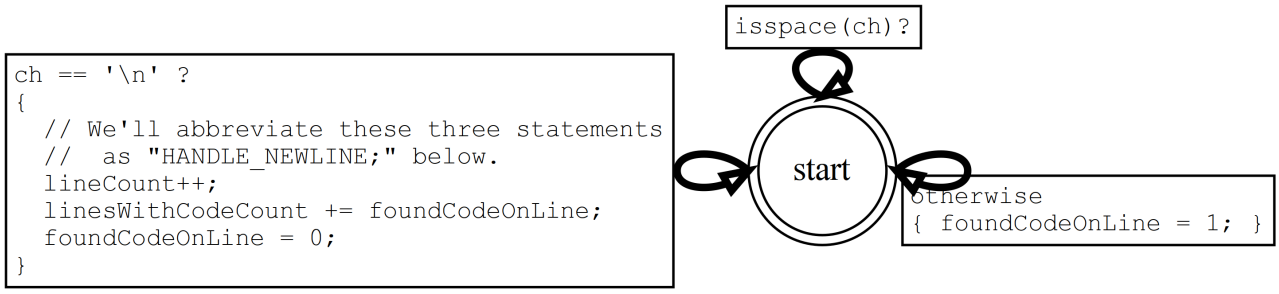
  Modify these three places accordingly whenever you add a new statistic.
- Most of the changes happen in `codeStats_accumulate()`:
    - New states get a new `enum` value at the start of the function.
    - New states get their own `case` branch of the `switch` statement. Be sure every case ends with a `break`. (See why?)
    - Within a state's `case` branch, each transition has its own branch of an `if` (or sometimes `switch`).
    - Statistics are incremented as part of the `codeStats` struct, which is being accessed through a pointer.

- The `assert(0)` in `codeStats_accumulate()` is there in case an unknown state is ever entered. This `assert()` should never be reached, but if it is, the program will crash intentionally.

Be sure to test `codestats_pt1` on a file for which you know the statistics. For this part, "`wc -l`" is a good independent check: both numbers should be the same.

Part 2: Counting Lines of Code

In this next part, we'll enhance the Part 1 result to count lines of code (including comments, for now). We don't need to add any new states, but we do add a new transition and modify the two old ones.



Note how the `foundCodeOnLine` flag works. It's initially zero. As characters in a line come in, we test them with `isspace(3)` to see if they're "whitespace". (Don't use your own set of whitespace characters. There are several unusual ones that `isspace(3)` knows about that you might miss.) If they are whitespace, do nothing, but if they are *not* whitespace we assume the line contains code and we set the `foundCodeOnLine` flag. We only need to know that there's at least one non-whitespace character on the line to count it as code, so we only set `foundCodeOnLine` to 1.

When we reach the `'\n'`, `foundCodeOnLine` will be 1 iff we found any non-whitespace on the line, so we we add it to the (new) `linesWithCodeCount` statistic.

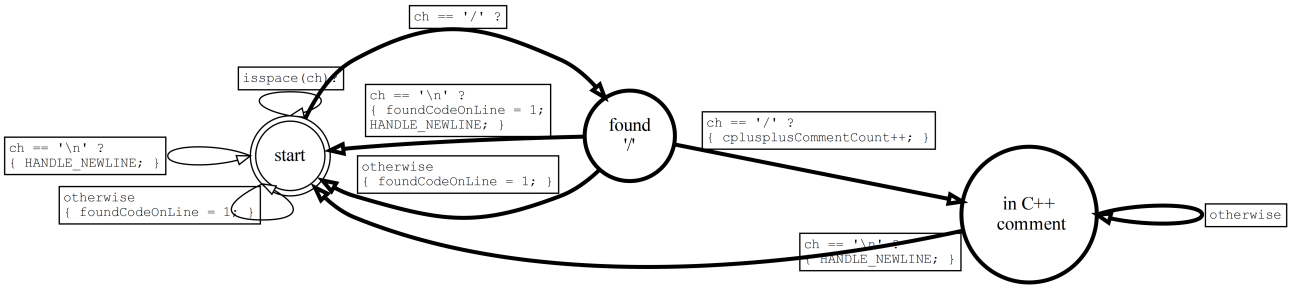
Now you're ready to enhance the code. Copy "`codestats_pt1.c`" to "`codestats_pt2.c`", make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt2` as well as `codestats_pt1` by default.

As we said, there are no new states, but there are some new branches within the `START` state's case branch.

Be sure to test `codestats_pt2` on a file for which you know the statistics.

Part 3: Counting C++-Style Comments

Detecting code isn't enough. We want to distinguish code from comments. The easiest comments to detect are C++-style comments, which start with "`//`" and go to the end of the line. Here's the FSM:



(Note: This and following state diagrams are hard to read on the printout, but if you visit the web page listed above in a browser, you can zoom in and read the fine print more clearly. There are also SVG files in the `"image"` subdirectory that you can zoom in on as much as you like, since they're line drawings and not pixellated.)

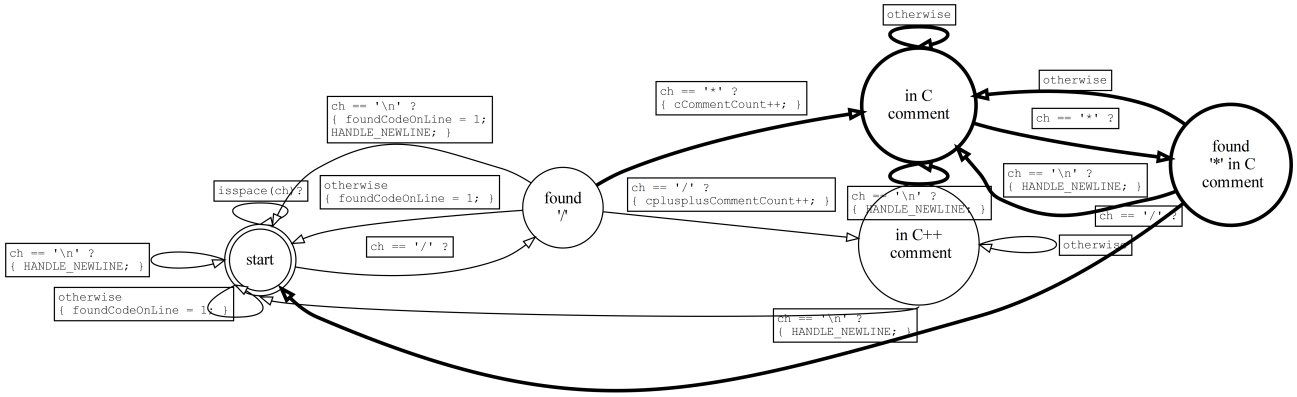
We've added two new states here. The first is entered when we encounter a `'/'` and the second is entered if that's immediately followed by another `'/'`. We can't assume the first `'/'` is leading into a comment. It might be part of an arithmetic expression. This means that we have to do reasonable things if the next character is *not* a `'/'`.

Copy `"codestats_pt2.c"` to `"codestats_pt3.c"`, make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt3` as well as the others by default. Add the new statistic `cplusplusCommentCount`.

Be sure to test `codestats_pt3` on a file for which you know the statistics.

Part 4: Counting C-Style Comments

We next count C-style comments, which begin with `"/*"` and end with `"*/"`. Here's the FSM:



They are a little trickier than C++-style comments. The latter are always ended by a newline, but it's possible to encounter a `'*'` within a C-style comment without it signifying the end of a comment. We therefore have to do the same kind of two-state process to get out of the comment that we did to get into it.

Copy `"codestats_pt3.c"` to `"codestats_pt4.c"`, make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt4` as well as the others by default. Add the new statistic `cCommentCount`.

Be sure to test `codestats_pt4` on a file for which you know the statistics.

Part 5: Dealing with Strings

So far, so good. But what about this code:

```
#include <stdio.h>

int main(void)
{
    printf("/* this is not really a C comment! */\n");
```

```
    return 0;
}
```

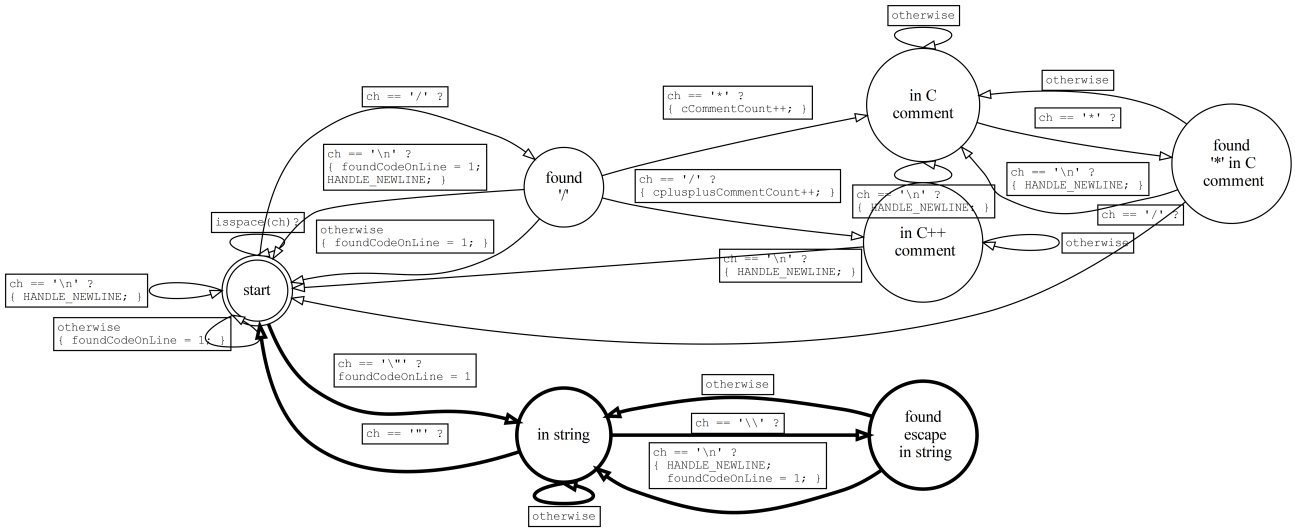
or this:

```
#include <stdio.h>

int main(void)
{
    printf("// this is not a C++ comment, either!\n");
    return 0;
}
```

(These are downloadable from the web page as `part5a_confuse_c_comment.c` and `part5b_confuse_cpp_comment.c`, respectively.)

`codestats_pt4` would detect comments in these where there were none. We need to enhance our code to ignore C double-quoted strings. Here's the FSM:



The tricky part here is the use of the escape character `'\'` within the string. When we encounter a `'\'` while scanning a string, we add an additional state which simply ignores the next character, even if it's `'"'`. In this way, we'll continue to scan the string until the closing (un-escaped) `'"'`.

Copy `"codestats_pt4.c"` to `"codestats_pt5.c"`, make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt5` as well as the others by default.

Be sure to test `codestats_pt5` on a file for which you know the statistics.

Part 6: Dealing with `char` Constants

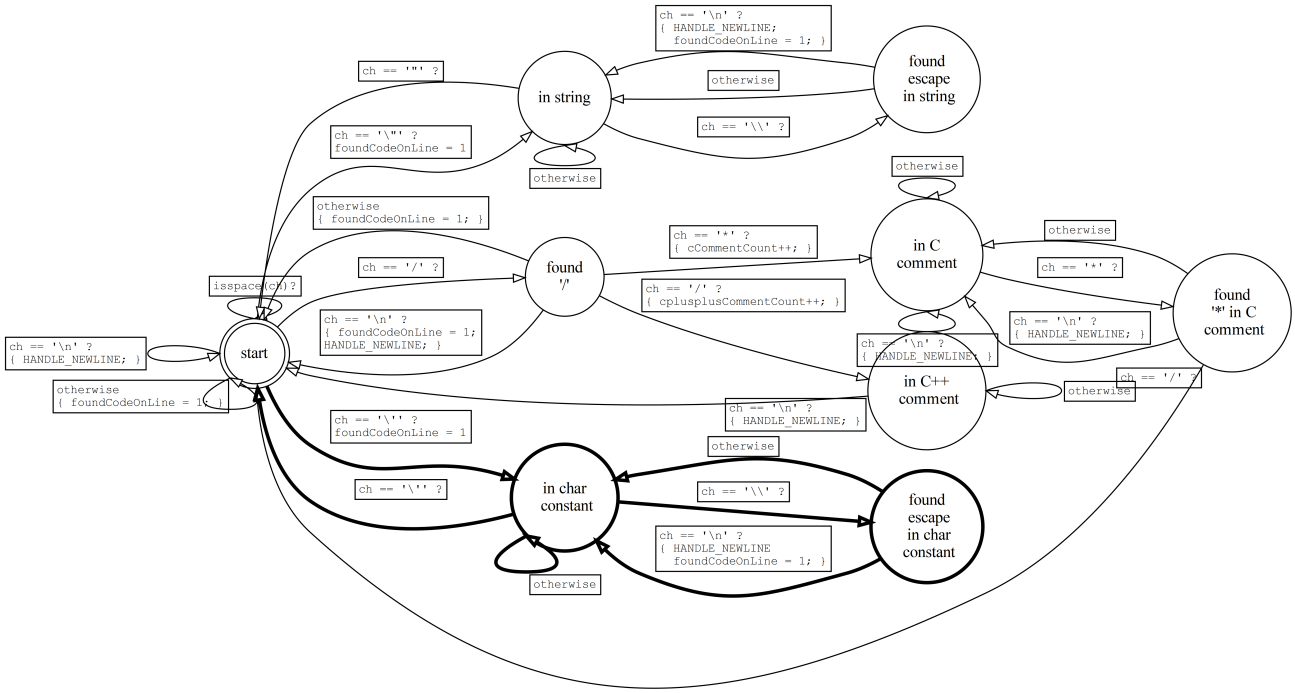
Just as we skipped (doubly-quoted) strings, we must also skip (singly-quoted) `char` constants. It may not seem obvious at first, but consider that the following code is perfectly legal (if somewhat nonsensical) C:

```
#include <stdio.h>

int main(void)
{
    char *s = "\
";
    // these are legal (but uncouth) C
    printf("%x\n", '//');
    printf("%x\n", '/**/');
    return 0;
}
```

(This is downloadable from the web page as `part6_weird_char_constants.c`.)

Yes, the multi-char constant is weird, but C permits them (with a warning). `codestats` can easily deal with them, however, in much the same way it dealt with string constants, including the escape. Here's the FSM:

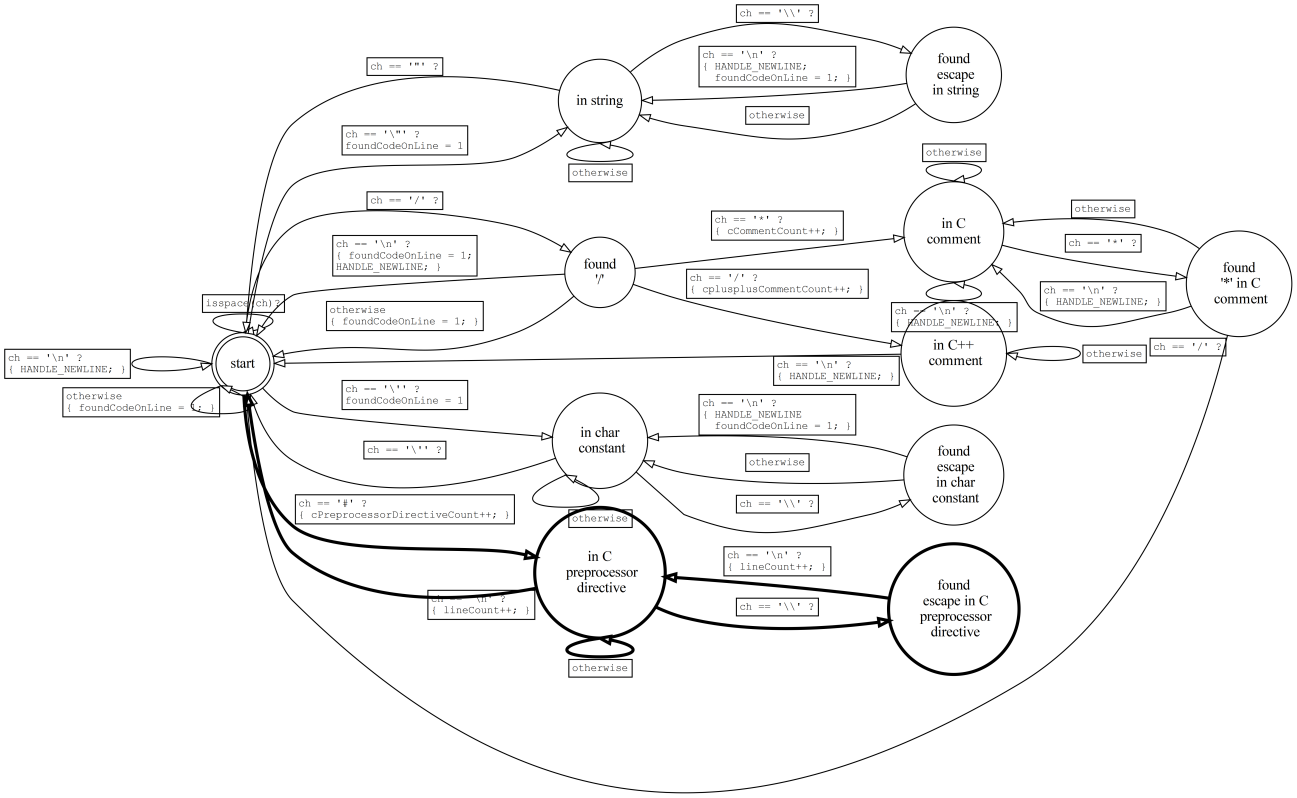


Copy "`codestats_pt5.c`" to "`codestats_pt6.c`", make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt6` as well as the others by default.

Be sure to test `codestats_pt6` on a file for which you know the statistics.

Part 7: Count C Preprocessor Directives

Finally, we enhance `codestats` to count (but otherwise ignore) C preprocessor directives. These are comparatively easy: They start with a `'#'` (not inside a string) and end at the end of the line, although multiple lines are permitted if all but the last ends with `'\'`. Here's the FSM:



Copy "`codestats_pt6.c`" to "`codestats_pt7.c`", make the above changes to the latter, and enhance the `Makefile` to build `codestats_pt7` as well as the others by default.

Be sure to test `codestats_pt7` on a file for which you know the statistics.