0. Introduction: Processing Text

A lot of Unix programming involves working with text.

Unix text processing programs read data in, do something with the data, and write data out. Text files have structure, the program has to extract meaning and content from that structure. For example, the train schedule file has the following structure:

a. Each line represents a train stopping at a station.

b. Each record contains six pieces of information.

c. The pieces of information are separated by semicolons.

d. The pieces of information are identified by a tag and an equals sign.

A program that wants to get meaningful content from this file has to be able to process lines, fields, tags, and values. Spotting these components requires identifying the different sections of each line. A programming idea that is extremely helpful for spotting sections of a pattern is the idea of modes or states.

We begin with an example.

1. Capitalizing Schedule Data

Consider typical lines of schedule data:

```
TR=052;dir=i;day=m-f;TI=19:11;stn=braintree;Line=kingston
TR=53;dir=o;day=m-f;TI=08:56;stn=salem;Line=newburyport
```

The names of the stops and the names of the lines are all stored as lower case strings. When we present data to users, it would look better if the names had upper case initial letters.

The program capitalize2.c reads text input, capitalizes any letter that follows a non-letter, and writes the resulting text to standard output. Here is the code:

```c
#include <stdio.h>
#include <ctype.h>

#define NOCHANGE_MODE 0
#define UPPERCASE_MODE 1

int main()
{
    int c;
    int mode = NOCHANGE_MODE;

    while( ( c=getchar() ) != EOF ){
        if ( mode == UPPERCASE_MODE ){
            if ( isalpha(c) ){
                c = toupper(c);
                mode = NOCHANGE_MODE;
            }
        } else if ( mode == NOCHANGE_MODE ){
            if ( c == '=' || c == ' ' || c == '/' )
                mode = UPPERCASE_MODE;
        }
        putchar(c);
    }
}
```

This program processes input one character at a time. When the program reads an equals sign, a space, or a
slash, it sets the variable called mode to UPPER_CASE_MODE to remind itself to capitalize the next alphabetic character it reads. Then, after processing an alphabetic character, it sets mode back to NOCHANGE_MODE.

A useful way to think about this behavior is to say the program operates in two modes: nochange and uppercase. In nochange mode, the program reads in characters and writes them out. In uppercase mode, the program reads in characters, converts them to upper case, then writes them out. The program switches mode based on data it sees. When the program sees an equals, space, or slash, the program changes into uppercase mode. When it sees an alphabetic character, it changes back to nochange mode.

Some programs operate in more than two modes.

2. Example 2 - Word Count

The Unix tool called wc counts words, characters, and lines in a file. Counting characters and lines is easy, counting words is easily done with a program that uses a mode variable. Consider input that looks like:

```plaintext
this is a test/ of some words to-be-counted.
```

The following code can handle the wide spaces and punctuation.

```c
#include <stdio.h>
#include <ctype.h>

/* wordcount.c - count words in input */
#define IN_A_WORD 1
#define BETWEEN_WORDS 2

int main()
{
    int c;
    int mode = BETWEEN_WORDS;
    int numwords = 0;
    while( (c=getchar()) != EOF )
    {
        if ( mode == BETWEEN_WORDS )
            if ( isalpha(c) )
                numwords++;
            mode = IN_A_WORD;
        else if ( mode == IN_A_WORD )
            if ( !isalpha(c) )
                mode = BETWEEN_WORDS;
        printf("%d\n", numwords);
    }
    printf("%d\n", numwords);
}
```

This machine has two modes. We usually call these states. The code has two conditional blocks. The first block handles one state; the second block handles the other state. In each state, the program examines the current character and takes action based on that character. The action can be to modify a variable (as in `numwords++`) and/or to change state.

That is the basis of state machines. In each state, the program examines the current character and takes action.

Trace through this code processing the sample data. The program spots the beginning of a word by keeping track of whether it is processing a word or if it is between words. If the program is between words, then the appearance of an alphabetic character indicates the beginning of a new word. Once the program has seen that initial letter and has incremented the word counter, it sets the mode so it knows it is already in a word. In that mode, alphabetic characters are considered just part of the same word. When it finds a non-alphabetic character, it changes the mode to reflect the fact it is no longer processing a word.
3. Example 3 - Real Life Example: Passing Mode

On some roads one sees signs that say "No Passing". Later on one sees a sign saying "Passing Allowed". As you drive along that road, you change from one mode to another. When you are in Passing Mode, you respond to slow drivers by passing them. When you are in NoPassing Mode, you respond to slow drivers by following along behind them. The signs cause you to change from one mode to another.

The world is full of situations where you operate in different modes. Often, specific inputs trigger changing from one mode to another. In the driving example, the inputs are visual input in the form of signs.

4. Programming Finite State Machines

Capitalize and wordcount are examples of finite state machines. Each has two states. They implement finite state machines. Many programming problems are easily solved by writing a finite machine. To do so: (a) list the set of states the program can be in, (b) describe, for each state, how the program behaves when in that state, (c) describe what inputs cause the program to shift from one state to another.

5. Visualizing Finite State Machines

FSMs are often drawn as a set of circles and arrows. Each circle represents one state, and each arrow represents a transition between states. The capitalization program shown above could be represented as:

As an exercise, add a new state called 'done' and add arrows to show how the program reaches that state.

6. States as Functions

The code for wordcount.c above uses an if..else if.. structure to handle the two states of the program. In a more complex machine, the sequence of if..else if..else if else... can make the function very long. One way to modularize your state machine is to make the handling of each state its own function. Here is a wordcount program that uses functions:

```c
#include <stdio.h>
#include <ctype.h>
/* wordcount-funcs.c - count words in input: word: seq of non-space */
int skip_blanks(); /* state: between words */
int read_word(); /* state: in a word */
int main()
{
    int numwords = 0;
    while( 1 )
    {
        if ( skip_blanks() == EOF ) /* read through spaces */
            break;
        numwords++;
        if ( read_word() == EOF ) /* read til end of word */
            break;
    }
    printf("%d\n", numwords);
    return 0;
}
```
// * skip_blanks: consume inter-word spaces until EOF or a non-space
//   rts: the terminator (EOF or the non-space char )
*/
int skip_blanks()
{
    int c;
    while( 1 ) {
        c = getchar();
        if ( c == EOF || !isspace(c) )
            return c;
    }
}

// * read_word: read until EOF or a space.
//   rts: the terminating char
*/
int read_word()
{
    int c;
    while( 1 ) {
        c = getchar();
        if ( c == EOF || isspace(c) )
            return c;
    }
}

In this program, the main function is shorter and clearer. Input text consists of alternating inter-word space and words. The program flow reflects that alternation. First the program reads in a sequence of spaces (space, tab, or newline), then the program reads in a word, then the program reads in the spaces, then a word, then spaces, ....

One elegant feature of this design is that being in a state is exactly the same as being in a function. The program alternates function calls to reflect the alternation of states, and that alternation of states reflects the alternation of types in input.

Each time the program switches to the read_word state, the counter goes up by one. The program simply counts the number of times it enters the words state (i.e. calls the read_word() function.)

The main loop can be made even shorter as:

    int main()
    {
        int numwords = 0;
        while( skip_blanks() != EOF ){
            numwords++;
            read_word();
        }
        printf("%d\n", numwords);
        return 0;
    }

Why does this program work as well as the longer version shown before?

7. State Machines and Switch Statements

Another technique for programming FSMs is to use a single function with a switch statement. Here is one:
# include <stdio.h>
# include <ctype.h>

/* wordcount.c - count words in input */

#define IN_A_WORD 1
#define BETWEEN_WORDS 2

main()
{
    int c;
    int mode = BETWEEN_WORDS;
    int numwords = 0;

    while( (c=getchar()) != EOF )
    {
        switch ( mode )
        {
            case BETWEEN_WORDS:
                if ( isalpha(c) ){
                    numwords++;
                    mode = IN_A_WORD;
                }
                break;

            case IN_A_WORD:
                if ( !isalpha(c) ){
                    mode = BETWEEN_WORDS;
                }
                break;
        }
    }
    printf("%d\n", numwords);
}

This is similar to the first example with the conditional blocks, but is faster (switch jumps directly to the correct section) and cleaner (there are fewer curly brace blocks. Also, the state names are neatly lined up. A complex state machine can be coded this way without being difficult to read or maintain.