Make it

Code workshop

Parsing comma-separated values

CSU remains a popular format. We show you how to write a program to read it…

In this issue…

▶ WHAT'S COVERED

Even in these days of XML, comma-separated value (CSV) files are still an important way to transfer data from one program or system to another. Providing everyone agrees on the character set used, using them for information transfer is extremely efficient; much more so than XML. However, there are many intricacies involved in dealing with CSV files and, in order to read them efficiently, you have to use a state machine.

Nugget

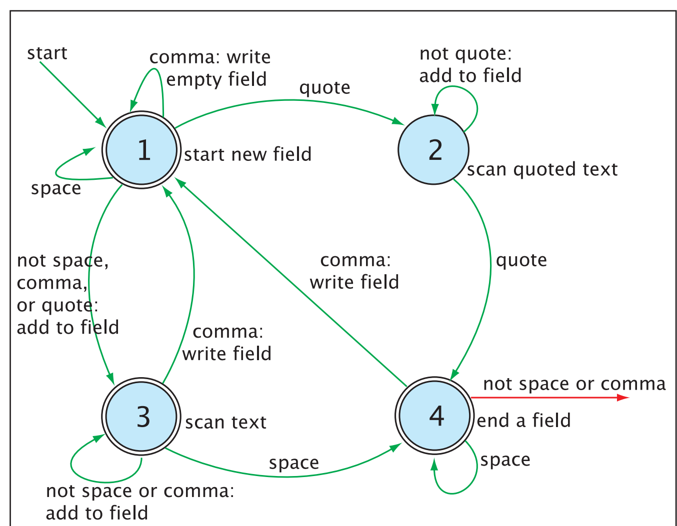
In a CSV file, each line is a separate record. But we can embed fields that span more than one line if we read the file character by character. We first have to expand the state machine to incorporate the processing of a complete file and of each record, each record being delimited by an end of line marker. Now when you read the file character by character, end of line markers denote ends of records.

Comma-separated value (CSV) files are used extensively to transfer data between programs. The latest version of Excel has a way to save data in a CSV file, and transferring data from an mainframe to a PC for analysis is still done using CSV files. Unlike data stored in XML, there are few libraries to help you read or parse CSV data, so sometimes you just have to roll your own parser.

Although on the surface a CSV file is a fairly easy format, there are some intricacies. For starters, a CSV file is purely a text file: no binary data is allowed. Each line in the text file is a separate record. Each record is divided up into text fields, with commas separating each field. By convention, spaces adjacent to a comma are ignored by the parser; they're there just to help a human read the file. If there's no data for a field, there's nothing between the commas.

The major point to realise so far is that field data is just text. The user of the data may 'know' that a particular field is numeric data, for example, but the CSV parser does not. It will read such a numeric field as a string comprising digits, perhaps with a period and sign, and it is up to the user of the data to then convert that string into a binary format.

The next problem is that since all the fields are textual, how do you get a field – an address, for instance – that contains a comma? It's easy: quote the field – that is, put its value inside double quotation marks. That way the parser of the file will read the opening quotation mark then scan and collect characters until it reaches the closing quotation mark. In particular, it won't look for a comma until after the closing quotation mark. It's an error if we run out of characters to read before the closing quotation mark



▲ **A state machine for reading a CSV record that has quoted strings.**

Furthermore, we insist that if a field is to contain a space, it should be enclosed in quotation marks. This way spaces are always ignored, unless they appear inside a quoted field. Since fields cannot contain spaces unless quoted, if we read a space and then a character that isn't a space or a comma, it's an error.

Watch and learn

Here's an example to show you a typical CSV record:

julian, 42, , "May 20, 2007"

It contains four fields: julian, 42, an empty field, and May 20, 2007. Notice that second field is a figure 4 followed by a figure 2 as a string. The parser doesn't convert it to, say, an integer containing the value 42. The fourth field contains a comma and the parser has removed the quotation marks. Finally, notice that the spaces inside the quotation marks are kept but other spaces are ignored.

In order to write some code to parse this CSV record, we devise a state machine. A state machine is a technique used to help solve other problems, especially in parsing. It comprises a set of states that describe the problem space, together with a set of transitions that describe the way to get from one state to another. The state machine has a 'start state', which is where you start the machine off, and one or more 'terminating states', which is where you'd expect to finish. If you finish in a state that is not a terminating state, it's an error.

Fig 1 shows a simplified state machine that parses CSV files. The intent here is that the state machine is used alongside a string containing the CSV record and parses that string. At each state we read the next character from the string and decide what to do about it based on the value of the character. We may transition to another state in doing so.

State 1 is where we start, and can be thought of as the 'about to ▶

PCP258.theory 33 11/6/07 15:11:08

start a new field' state. We read the next character. If it's a space, we ignore the character and return to the same state. If it's a comma, we write out an empty field and return to the same state. If it's a quotation mark, we move to state 2. For any other character, we move to state 3. If we reach the end of the string and there's no next character, it's an error.

State 2 is where we scan the contents of a quoted field. For any character that's not a quotation mark, we add the character into a local string. For a quotation mark, we write out the contents of the string as a field and move to a special state 4.

State 3 is where we scan the contents of a field. For any character that's not a comma or a space, we add the character to a local string. For a space, we ignore it and move to the special state 4. For a comma, we write out the field and return to state 1.

State 4 is where we ignore the spaces at the end of a field before the next comma. For a space we ignore it and stay in the same state; for a comma we write out the field and return to state 1. For any other character it's an error.

The only state that's not a terminating state is state 2. We must read a closing quotation mark when parsing a quoted string, and so we cannot run out of characters in state 2.

Also notice that the transitions not only define how to move from state to state (or from one state to itself), but also define an action to be taken. The actions we have defined include: write out a field, add a character to a field value, ignore the character.

After we've drawn the state machine, the problem is to implement it. Using a procedural language like C, it's fairly easy, and essentially just consists of a big switch statement. Using object-oriented code, it can be a little puzzling, so that's what we'll do. I'll use C# to do it.

The first thing to do is write an interface that defines the behaviour of a state. This one is easy: there's a method called Process that takes in a character, and that returns another state, and there's a property we can read to determine whether the state is a terminating state or not. We'll also have to write an engine class that drives the state machine: it will be in charge of reading characters one by one and calling this Process method.

```
public interface IState {
IState Process(char ch);
bool IsTerminator { get; }
}
public static class CsvStateMachine
{
public static void Execute(string
text, IState startState) {
IState currentState = startState;
foreach (char c in text) {
currentState = currentState.
Process(c);
}
if (!currentState.IsTerminator)
throw new Exception("Done
parsing, final field is not complete");
FieldProcessor.Finish();
}
}
```

The state machine's Execute method uses a utility class called FieldProcessor. All this does is to accumulate characters into a local string (the AddChar method), and then display the string when a field is complete (the Finish method). This could certainly be optimised and enhanced. You could type in the following code to do this:

```
public static class FieldProcessor {
private static string field = String.
Empty;
public static void AddChar(char c) {
field += c;
}
public static void Finish() {
Console.WriteLine('[' + field + ']');
field = String.Empty;
}
}
```

Now we can start filling in the individual state classes. Here's an example to get you going (this is state 1 in the diagram):

```
public class FieldStartState : IState {
public IState Process(char ch) {
switch (ch) {
case ',':
FieldProcessor.Finish();
return this;
case '"':
return new ScanQuotedFieldState();
case ' ':
return this;
default:
FieldProcessor.AddChar(ch);
return new ScanFieldState();
}
}
public bool IsTerminator {
get { return true; }
}
}
```

The Process method looks at the character to process and decides what to do based on its value. If it's a comma, the current field is complete (it'll be empty), and the method then returns this state to the caller. If the character is a quote, Process returns a new quoted field state. If it's a space, it returns this state back. Otherwise it adds the current character onto the field, and then returns a new scan field state.

The other states – ScanFieldState (state 3), ScanQuotedFieldState (state 2), and FieldEndState (state 4) – are implemented by following the state machine diagram much like the one in this article. ■

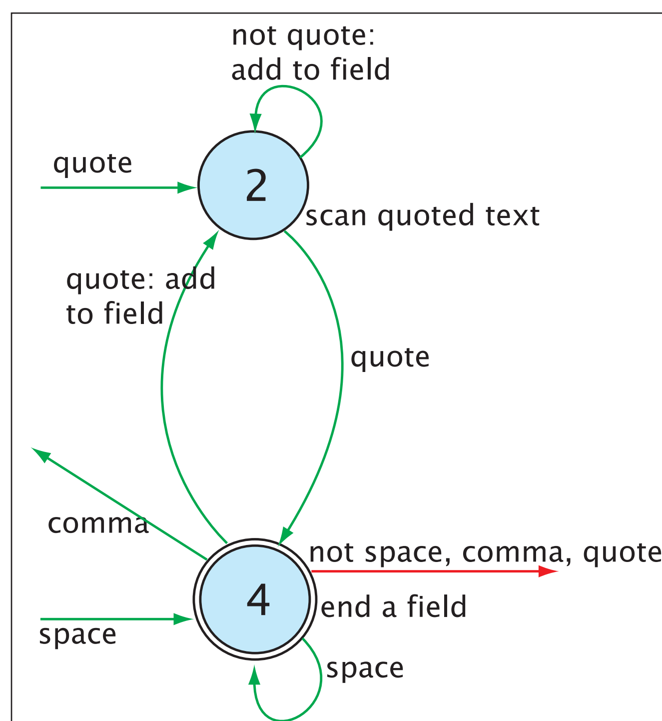
Julian Bucknall is chief technology officer for Developer Express.
feedback@pcplus.co.uk

Spotlight on… Quoted strings that contain a quote

We introduced quoted strings to solve the problem of a field that contains a comma or a space, but what happens if the field should contain a quotation mark of its own? For example, suppose the field has the value Mack "The Knife". How would we now represent that in a comma separated values file?

The solution is to use a quoted string, but double every quotation mark inside the string. So the field in our CSV file would be "Mack ""The Knife""".

Our state machine now needs to change to accept these quotation marks. Fig 2 shows the altered state 4. If the next character when we're in state 4 is a quotation mark, we add a quotation mark to the field and return to state 2. Otherwise state 4 remains the same. ■



▲ **The state machine changes to support quoted strings containing quotes.**

Nugget

There are a few inefficiencies in my code. String concatenation in the FieldProcessor is the first. Each time you add a character to a string, you risk having to grow the memory holding the string. There's a string builder class we can use instead. The second is the creation of new states within the Process methods, as we can just use a single instance of each state. Do this by writing a factory class that dispenses pre-built state objects.

PCP258.theory 34 11/6/07 15:11:10