Development of a State Machine for ALCODE

There is now an implementation of a state machine embedded within ALCW. The drivers for this are these:

a) Currently processing text sources throws up several issues:
1) The text-processor dialog tool requires a complex description of where records split, and sometimes there is not a clear definitive statement of where that happens, it then requires a deep understanding of where data items are within the 'record' and those that are outside the 'record'.
2) The text-processor requires a precise definition of a precise input, we have seen many examples of where that input cannot be relied upon, and the smallest diversion from any given example causes the processing to fail.
3) The text-processor is susceptible to the input file's encoding. There are some strange – non-ISO – encodings that appear for ALCW.
4) The text-processor is designed to 'hide' the complexities of regular expressions, and yet most of the matching lines needed within the text-processor have to be hand-crafted as [quite complex] regular expressions.
5) The processing mechanism when embedded within the text-processor is invisible to the run-of-code as described by the ALCW code file. It is not obvious when reviewing the ALCW code how record fields are sought and assigned.

b) The eventual solution is to use regular expressions to match lines of text input and then a series of global variables – holding interim values as processing proceeds – then testing those before deciding to emit a record – infilling values processed earlier. This is a scheme that – while valid – is prone to errors, particularly when input is provided that is not expected. That is, there is a likelihood that input not seen or expected – while valid input – is incorrectly processed. Further, the scheme is not robust enough to capture error conditions, nor flexible enough to correctly process [previously unseen] valid input.

c) The text-processing scheme is a generalisation of any possible text input, yet it has to distinguish between a tab delimited layout and a character position layout – it would never be possible to process one input type defined for another, yet the fundamental principle [that there is a delimiter of some sort] could be easily handled by a regular expression – which is why there is now a tendency to use a set of 'custom regular expressions' rather than the full power of the text-processor.

The ALCW State-Machine

The state-machine as implemented in ALCW is an amalgamation of a 'structured variable' (an array of property/value pairs) and a callable function. It maintains a single 'state-value', and that value can be thought of as a sting. Within it, it has definitions of how a given state will transit to another state – when a clause is true. On a successful transition the machine will [optionally, if the function exists] call a function.

Consider this as a [trivial] example. We aim to examine if a string of tokens contains a greeting. Suppose a string is “Hello World”; we will define the state-machine like this (further explanation of syntax later):

state Greeting **step** null **then** HaveHello **when** “Hello”;
state Greeting **step** HaveHello **then** HaveGreeting **when** “World”;

This says, starting from the null state (the initial state) transit to the state HaveHello if the state-machine matches with the string “Hello”. It also says, if the current state is HaveHello then transit to the state HaveGreeting if the machine matches “World”.

The form that causes the state-machine to test and transit is this:

Greeting->”string”;

So let us suppose we issue these two statements:
Greeting->”Hello”;
Greeting->”World”;
Then the final state of the machine will be HaveGreeting because it did transit from the initial state to HaveHello because it matched “Hello”, it then did transit from HaveHello to HaveGreeting because it was in the state HaveHello (therefore looking for the next token to be “World”) and it matched “World”.

What would have happened in this case if the input string had been “Hello to the World”.
Well it would still result in the final state being HaveGreeting. We will step through that:
We start from the initial state (null), we match the first token “Hello” and move the state to HaveHello. We then receive the token “to”. This does not match the pattern for the state HaveHello which is a match in “World” – so the state remains HaveHello. We then receive the token “the”, again this is not matched with “World” so the state remains the same. We then (finally) receive “World”, this does match the pattern for the current state of HaveHello and therefore transits to the state HaveGreeting. Of course, this might be exactly what you intended; it is both robust and flexible enough to ‘understand’ a greeting when it gets one!
If, on the other hand, you wanted to ensure you only considered a valid greeting to be “Hello World” then you would change the definitions like this:

state Greeting **step** null **then** HaveHello **when** “Hello”;
state Greeting **step** HaveHello **then** HaveGreeting **else** null **when** “World”;

Notice the **else** clause in the state definition for HaveGreeting.

So stepping through the previous “Hello to the World” example: again we transit from the initial state to HaveHello when we receive “Hello”. We then receive the token “to”; our current state is HaveHello and this says transit to HaveGreeting if the token is “World”, otherwise (the else clause) revert to the initial state (null). We then receive the token “the”; the current state is now null and it does not match the transition to HaveHello (which expects “Hello”), we then receive the token “the” and the same situation, finally we receive “World” and this will also not match the only transition from the null state to HaveHello which only expects the token “Hello”. The final state of the machine is therefore the null or initial state.

You might see that the string: “Hello, Hello, Hello World” would successfully transit to the state HaveGreeting because although it would reset itself several times with consecutive “Hello”s and commas, it would finally match the tokens “Hello World”.

On any successful transit from one state to another the state-machine calls the function (in this case Greeting). This provides the opportunity to test the state and take an action. It might look like this:

```
function Greeting() begin
  console(“We have a state of {Greeting?}”);
  end
```

The function (the callback) has no parameters. Notice the new operator ‘?’, this fetches the current state. The callback function is called after the state changes.

So on the receipt of the string “Hello World”, the result would be two console lines looking like this:

```
We have a state of HaveHello
We have a state of HaveGreeting
```

It might be more useful to extend the call-back like this, for example:

```
function Greeting() begin
  case (Greeting?) begin
    when “HaveGreeting” console(“And a hello to you also”);
    end
  end
```

So this ignores the transit states other than HaveGreeting.

Capturing items from the match

Captured items from a regular expression matched in a successful transition are assigned to a property of the state-machine. The greeting example might expect strings of the form “Hello World” or “Hello Earth”, or perhaps hello anywhere. The set of state-machine definitions might therefore be extended like this:

```
state Greeting step null then HaveHello when “Hello”;
state Greeting step HaveHello then HaveGreeting else null when
“(?P<planet>[\w]{1,})”;
```

Then the state-machine assignments:

```
Greeting->”Hello”;
Greeting->”Earth”;
```

Our call-back function can further be extended to output the captured string in this way:

```
function Greeting() begin
  case (Greeting?) begin
    when “HaveGreeting” console(“And a hello from {Greeting.planet} to you also”);
    end
  end
```

Causes the output: “And a hello from Earth to you also”.

All the properties of the state-machine can be extracted in the usual way:

for string properties $p **in** Greeting console(“Greeting has {$p} as {Greeting.$p}”);

Which causes the output: “Greeting has planet as Earth”.

Formal Syntax of the state-machine

To define a state rule:

```
state <state_machine_name>
   step <current_state>
   then <transit_to>
   else <transit_back>
   when <pattern>
```

A <state_machine_name> is a name consisting of letters, digits and the under-bar character – as would be valid for a function name.

The terms <current_state>, <transit_to>, and <transit_back> are also names – as would be valid for a property name.

The term <pattern> is a string for matching, including a regular expression (but void of delimiters).

There is a block rule syntax for defining multiple rules for a given state-machine:

```
state <state_machine_name> begin
  step <current_state> then <transit_to> else <transit_back> when <pattern>
  step <current_state> then <transit_to> else <transit_back> when <pattern>
  end
```

There are special forms of <current_state>:
The null keyword represents the initial state. The ‘*’ symbol represents any state.
The <transit_to> and <transit_back> can also have the null term.

This rule:
  state m **step** null **then** s1 **when** “pattern”;

Says, in the state-machine ‘m’, go to the state s1 from the initial or null state when matched on “pattern”.

This rule:
  state m **step** * **then** s2 **when** “pattern”;
Says, in state-machine ‘m’, whatever the current state, go to the state s2 when matched on “pattern”.

Notice that both the <transit_to> and <transit_back> clauses can be omitted.

So this rule:
  state m **step** s1 **when** “pattern”;
Says, in state-machine ‘m’, when matched with “pattern” remain in state s1 (and as there is no else clause then the rule has no transit effect at all – however it is useful to capture elements from the pattern when that is a regular expression).

To cause a transition:

The state-machine is caused to evaluate a transition with a statement of this form:

<state_machine_name> -> “input string”;

The form can be used as a stand-alone statement, or as a Boolean term, that is the statement returns a Boolean true or false – true when the statement caused a matched transition:

if <state_machine_name> **->**”Hello” **then** console(“Matched on hello”);

To test status:

The operator ‘?’ has already been shown in the examples above, it returns the current state:

<state_machine_name> ?

console(“Currently in state {Greeting?}”);

Advanced Functions:

The statement ‘extends’ is an advanced form that is used (often) in common code. It has two functions 1) to provide a known alias to a state-machine and 2) to extend the rules of a “base” or “parent” state-machine.

state <state_machine_name> **extends** <a_other_machine_name>;

For example:

state ProxyGreeting **extends** Greeting;

But more specifically (and more usefully) like this:

state ProxyGreeting **extends** $v;

Where $v would evaluate to the string “Greeting”.

Profile Example

This is a working example for the CSS_MBNATRSU_ETL profile:

The state machine is defined like this:

```
state MBNA begin
  step null then DateLine when
                    "(?P<Posted_Date>\d\d\d\d-\d\d-\d\d).*?(?P<TranDate>\d\d\d\d-\d\d-\d\d)$";

  step * then RecordLine when
"(?P<Program_Name>[^\t]{1,})[\t]{1,1}(?P<Transaction_Group>[^\t]*?)[\t]{1,1}(?P<Count>[\d]{1,})
[\t]{1,1}(?P<Sum>[\d\-]{1,})";

  step RecordLine then CheckLines when "(?P<RecCount>[\d]{1,})";
  end
```

These rules look initially for a line of this form:
```
2013-02-04 09:25:55.999    1000000000000190331,… 2013-02-01
```
Moving the state to ‘RecordLine’, it will then read lines of the form:
```
BAC Virgin Travel EUR Standard   01 Load    13    2163.93
```
Which represent a record (the call-back, shown below, will cause these records to be offered for insertion), until it reads a line of the form:
```
22
```
Which terminates the load process.

The loader is implemented by the common code 'load_from_state' with these properties set:

```
  $p.custom_regx = null;
  $p.use_state = "MBNA";
  $p.emit_on_state = "RecordLine";
```

Inside of the common code the call-back is implemented thus (for information only):

```
state ProxyLoad extends $p.use_state;
#_load_from_state = $p;
for all properties $line_no in $lines begin
    $line = $lines.$line_no;
    ProxyLoad->$line;
    end

function ProxyLoad() begin
  case (ProxyLoad?) begin
    when #_load_from_state.emit_on_state begin
       for string properties $field in ProxyLoad $record.$field = ProxyLoad.$field;
        if loader_insert($record, #_load_from_state.checksum_field,
                                  #_load_from_state.use_db_table,
                                  #_load_from_state.file_field_lookup,
                                  #_load_from_state.file_add_fields,
                                  #_load_from_state.file_field_process,
                                  #_load_from_state.file_not_null_fields) is true
          #file_lines = #file_lines+1;
      end
    end
  end
```