Indirect Function Calls

There are three language structures that can cause different execution paths based on a value of an expression.

IF-THEN-ELSE
The most basic is: if <Boolean value> then <stmt> else <stmt>
Where the value can have multiple values the if-then-else structure is easily constructed like:

```
if <Boolean value> then <stmt>
else if <Boolean value> then <stmt>
else if <Boolean value> then <stmt>
…
else <default stmt>
```

However, this structure soon becomes unreadable and unmanageable.
See the note on 'dangling else problem' below.

CASE

Something more readable, and more efficient in the underlying generated code is the case statement, like this:

```
case <a value> begin
   when <a value is this value> <stmt>
   when <a value is this value> <stmt>
   when <a value is this value> <stmt>
   <default stmt>
   end
```

However, this form will still become unmanageable with a lot of 'when' clauses, however if has the advantage over the if-then-else form in that the when clause can be an expression that is evaluated only as needed.

INDIRECT FUNCTIONS

The indirect function allows a function name to be evaluated and then called. So if we consider the collection of an event value or the title of a triggering profile, then we can construct a handling function for each of the potential values received. The calling code can be left unchanged as it is just a task of writing the handling function.

Here is the basic form of an indirect function call:

```
$afn = "my_fnct";
$x = $afn(1,2);

function my_fnct($p, $q) begin
   return $p+$q;
   end
```

In other words make a function call via a variable, where the variable is the string name of the function call required to be called.

The reason for this, consider the profile that is triggered by any number of masters, if we know the name of the (master) triggering profile(s), then we can set up a variable based on that name to call the handling function.

So:

```
// The following code need not be changed again
$uploaded = detect_trigger();
if $uploaded is not false begin
   $call_handler = "handle_" + $uploaded.TriggerByTitle;
   If $call_handler exists then
      $result = $call_handler($uploaded.FilePath)
  Else
      $result = default_handler($uploaded.FilePath
   end
```

Then you can have functions:

```
function handle_cimb_etl($path)

function handle_mcc_etl($path)

function default_handler($path) // error case or default action
```

And the appropriate function will be called based on what profile triggered this profile. Add handling functions as new values become available.

The Dangling Else problem.

The **dangling else** is a problem in computer programming in which a seemingly well-defined statement can become ambiguous. In many programming languages one may write conditionally executed code in two forms: the if-then form, and the if-then-else form:

if a **then** s
if a **then** s1 **else** s2

This gives rise to an ambiguity in interpretation whenever an if-then form appears as `s1` in an if-then-else form:

if a **then if** b **then** s **else** s2

In this example, `s` is unambiguously executed when `a` is true and `b` is true, but one may interpret `s2` as being executed when `a` is false (thus attaching the else to the first

if) or when `a` is true and `b` is false (thus attaching the else to the second if). In other words, one may see the previous statement as either of the following expressions:

if a **then** (**if** b **then** s) **else** s2
or
if a **then** (**if** b **then** s **else** s2)

Languages solve the problem (often) by ensuring that the if-then clause is explicitly terminated – often by 'endif'. The alcode solution is to terminate the if-then clause with a ';' (semicolon) just like all other statements can be terminated. A semicolon after the if-then clause simple says there is no else clause to follow.

So this should be read as unambiguous:

if a **then if** b **then** s; **else** s2;

Which is:

if a **then** (**if** b **then** s) **else** s2;