Error Detecting Changes to TANGLE

R. M. Damerell
Royal Holloway & Bedford College

This article describes a proposal to improve the diagnostics of TANGLE by making it detect certain types of errors. No corresponding change is proposed for WEAVE, on the theory that users are unlikely to want to weave a program until the worst of the bugs have been fixed. I would be much obliged if members of the TeX community could say whether these changes are desirable. If the response is favourable, then I propose to make copies of the modified TANGLE freely available. The changes described in this article apply to version 2.5 of TANGLE, as supplied by Stanford.

Change 1: Missing Start of Module

By far the most frequent and troublesome error that I make when writing WEB programs is that of omitting the string @␣ that should separate two successive modules. This error usually damages the PASCAL output. Consider the following fragment, copied from TANGLE:

```
@<Globals in the outer block@>=
@!history:spotless..fatal_message;

@ @<Set initial values@>=
history:=spotless;
```

Now suppose the user omits the @␣ between these modules. Then the whole replacement text of the second module will be inserted into the first. If this error is not detected, then probably the user will try to compile the damaged output. The likeliest result is that the compiler will become confused and print lots of error reports that bear no clear relation to the cause of the trouble. There seems to be no obvious way to detect this type of error every time, but the changes here proposed will detect it with a fairly high probability.

```
@<Globals in the outer block@>=
  try_loc: integer;

@ @<Was an '@@' missed here?@>=
try_loc := loc;
while (buffer[try_loc] = " " )
  and (try_loc < limit) do incr(try_loc);
if    (buffer[try_loc] = "+" )
  and (try_loc < limit) then incr(try_loc);
while (buffer[try_loc] = " " )
  and (try_loc < limit) do incr(try_loc);
if    (buffer[try_loc] = "=" ) then
  begin err_print
    ('! Nested named modules. Missing @?');
      @.Nested named modules@>
  @<Show list checkpoint@>;
  end
```

Figure 1. *Code to Detect Missing @␣. The procedure* scan_repl *is changed to execute this code when it has just read a module name and* loc *is pointing to the character that follows the* @> *at the end of the name.*

First, I have changed TANGLE so that it enforces an extra rule of syntax: if the replacement text of one module includes a call of another module, then that call may not be immediately followed (on the same line) by = or +=. This rule is not really valid for WEB programs: somebody might want to say:

```
if @<some expression@> = 0 then ...
```

but this construction does seem unlikely in PASCAL. Also the test can be bypassed by putting the ' = 0' onto the next line.

Figure 1 shows the code that performs the test. TANGLE has just read a module name, say NNN, within the replacement text on another module, say MMM. Then loc should be pointing to the next character after the @> that ends the module name NNN. TANGLE must then examine the next few characters in the file without losing its position.

That module must be called by inserting

```
@<Was an '@@' missed here?@>;
```

immediately before the call of `app_repl` which follows the label `module_name` in procedure `scan_repl`.

Change 2: Showing a checkpoint

The next change is suggested by the fact that several errors, which are currently detected by `TANGLE`, nearly always occur as symptoms of a missing `@␣`. A typical example is "Identifier conflict." Here is how this happens. Suppose the programmer omits an `@␣` between two modules. Then `TANGLE` will read the TeX part of the next module as if it were PASCAL. When it reads a sentence like "If this fails, then ...," the "If" conflicts with PASCAL's `if`. The obvious change here is to alter the error message to warn the user of this possibility. Also, I have inserted a call of

```
@<Show last checkpoint@>;
```

immediately after the code that generates the message. The messages "`@d/@f/@p ignored in PASCAL text`" also seem to occur frequently as symptoms of a missing `@␣`, and I have treated them in the same way.

When `TANGLE` thinks it has found a missing `@␣`, we would like to tell the user where it ought to have been. The best I can do is to indicate a range of line numbers. The actual error is that two consecutive modules, say `MMM` and `NNN`, have been run together; the formal error is an illegal construction within the PASCAL code of module `MMM`. So `TANGLE` will say where it thinks the PASCAL code of that module started.

Change 3: Semicolon-Else

Another frequent error is that of writing a program containing the sequence ` ; else `, which PASCAL does not allow. This is easy to do, as the semicolon and the `else` often come from different modules. Although there is absolutely no reason why you cannot say ` ; else ` in `WEB` programs, it still seems desirable that `TANGLE` should detect this, as (on our machine, at any rate) `TANGLE` runs between 5 and 10 times as fast as the PASCAL compiler. The test applied here is very crude; it will not recognise `ELSE` or `Else` or the output from ` el@& se `.

The module to detect the change, listed in Figure 3, gets called by inserting

```
@<Semi else test@>;
```

```
@p procedure scan_module;
  label continue, done, exit;
  var p: name_pointer;
  {module name for the current module}
  begin incr(module_count);
    @<Scan the \(definition part...@>;
    last_line := line;                       — insert
    was_changing := changing;                — insert
    @<Scan the \PASCAL\ part...@>;
    exit:
    last_line := 0;                          — insert
  end;

  :

@<Globals in the outer block@>=
last_line: integer;
was_changing: boolean;

@ @<Show last checkpoint@>=
if last_line = 0 then
  print_ln(' (not in PASCAL)')
else
  begin
  print_nl('PASCAL part of module',
    ' began at line ' , last_line);
  if was_changing then
    print(' in change file');
  print_ln(' ');
  end

@<Set initial values@>=
last_line := 0; {where PASCAL started}
was_changing := false;
```

Figure 2. *Code to Show Checkpoint. The lines added to the* `scan_module` *procedure track the starting line of the latest piece of PASCAL code. Notice the checkpoint is reset at the beginning of the program and after the end of the PASCAL part of each module. The module* `@<Show last checkpoint@>` *displays the saved checkpoint; references to this module are added following the error reporting code for errors that are often caused by a missing start of module.*

immediately after the label `reswitch` in procedure `send_the_output`.

Change 4: Error reports to a file

Another change that I have found very useful is to make TANGLE write its error reports onto a file as well as the terminal. With a split screen editor, one can work through the error and source files in parallel. This is much easier than writing the errors on paper as they appear. Essentially, you say:

```
    @d print(#)== begin write(term_out,#);
       write(errorfile,#); end
```

but several refinements are needed to prevent routine messages from getting into the error file. There seems to be no point in giving details here as they are long and messy and system dependent.

Conclusion

In designing these changes, I have tried to ensure that the new version of TANGLE will be compatible with the old. So when TANGLE thinks it has found an error, it merely prints an error report without making any attempt to correct the supposed error. The missing @␣ test is fairly effective: there are only about 10 places in WEAVE.WEB where you can omit an @␣ without it being detected. The semicolon-else test is also effective, provided that the user always writes else in lower case. The modified TANGLE seems to run about 2% slower than the old version; but users will save more machine time by not trying to compile bad PASCAL programs. I believe that these changes will significantly reduce the time that programmers have to spend in removing trivial errors from WEB programs.

```
@<Globals in the outer block@>=
point_else: name_pointer;
semi_last: boolean; {output was semicolon}
@ The first step is to put an '|else|' into the hash
table,
to be used in later comparisons.
@<Initialize the input system@>=
  buffer[0] := "e"; buffer [1] := "l";
  buffer[2] := "s"; buffer [3] := "e";
  id_first := 0;
  id_loc :=4;
  point_else := id_lookup(normal);
  buffer[0] := " ";
  semi_last := false;

@ Then |send_the_output| must test
for ' ; else ', ignoring intervening
comments or white space.

@<Semi else test@>=
  if   (cur_char = begin_comment)
    or (cur_char = join)
    or (cur_char = module_number)
    or (cur_char = 0)
    or (cur_char = force_line)
    or (brace_level > 0)
  then do_nothing
  else if cur_char = ";" then
    semi_last := true
  else if semi_last
   and (cur_char = identifier)
   and (cur_val = point_else)
    then
      begin err_print
       ('! semicolon-ELSE found');
         @.semicolon-ELSE found@>
      semi_last := false;
      end
  else semi_last := false;
```

Figure 3. *Code to check for semicolon - else combination. A reference to the module* `@<Semi else test@>` *is added to the main loop of* `send_the_output`, *following the* `reswitch` *label, to check the program as it is expanded and output.*