The SCANTEX Processor

Version 1.0

1. Introduction. This program takes a TeX file generated by WEAVE and strips out the sections which have not been changed, outputting the changed sections to a second, greatly reduced TeX file. The index, section names, and table of contents are dropped as well.

The program uses a few features of the Modula-2 compiler used in its development (Logitech MS-DOS) that may need to be changed in other installations. System-dependent portions of SCANTEX can be identified by looking at the entries for 'system dependencies' in the index below.

The "banner line" defined here should be changed whenever SCANTEX is modified.

```
define
        banner ≡
        'This␣is␣SCANTEX,␣Version␣1.0'
```

2. The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The TeX input comes from file *TeX_file* and the new TeX output goes to file *out_TeX_file*. Unlike Pascal, Modula-2 does not require the constant, type, and variable sections to be placed here in the program header in a rigidly specified order, but we will do it anyway, since WEB makes it so easy.

```
module scan_TeX; ⟨Import List 4⟩
  const ⟨Constants in the outer block 5⟩
  type ⟨Types in the outer block 6⟩
  var ⟨Globals in the outer block 7⟩
```

3. This procedure initializes the module.

```
procedure initialize;
  var ⟨Local variables for initialization 15⟩
  begin
    ⟨Set initial values 8⟩
    ⟨Initialize the file system 16⟩;
  end initialize;
```

4. A few macro definitions for low-level output instructions are introduced here. All of the terminal-oriented commands in the remainder of the module will be stated in terms of simple primitives. The boxes signify words that must not be forced to uppercase when the program is MANGLED, since Modula-2 is case-sensitive.

```
define pr_char ≡ [␣Write␣]  (* library
        procedure to output a character *)
define pr_string ≡ [␣WriteString␣]  (* library
        procedure to output a string *)
define rd_string ≡ [␣ReadString␣]  (* library
        procedure to input a string *)
define pr_card ≡ [␣WriteCard␣]
        (* library procedure to output a
        cardinal number *)
define new_line ≡ [␣WriteLn␣]  (* a new line *)
define print_string(#) ≡ pr_string(#)  (* put a
        given string to the terminal *)
define read_string(#) ≡ rd_string(#)  (* read a
        given string from the terminal *)
define print_cardinal(#) ≡ pr_card(#, 1)  (* put
        a given cardinal to the terminal, in
        decimal notation, using only as many
        digit positions as necessary *)
define print_ln(#) ≡ pr_string(#); new_line;
        (* put a given string to the terminal,
        followed by a new line *)
define print_char(#) ≡ pr_char(#)  (* put a
        given character to the terminal *)
```

```
⟨Import List 4⟩ ≡
  from [␣InOut␣] import
      pr_string, rd_string, pr_char, pr_card, new_line;
```

See also sections 10 and 11.

This code is used in section 2.

5. Let's define a few constants.

```
⟨Constants in the outer block 5⟩ ≡
  buf_size = 1000;
      (* maximum length of input line *)
  file_name_len = 200;  (* length of a file name *)
```

This code is used in section 2.

6. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no real errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

We don't really have to worry about errors too much in this particular program because the input is machine-generated (by WEAVE). The error likeliest to occur is failure during file opens.

```
define mark_harmless ≡
        if history = spotless then
          history ← harmless_message;
        end ;
define mark_error ≡ history ← error_message
define mark_fatal ≡ history ← fatal_message
define err_print(#) ≡ print_ln(#); mark_error;
```

⟨ Types in the outer block 6 ⟩ ≡

```
error_level = (spotless, harmless_message,
    error_message, fatal_message);
```

This code is used in section 2.

7. ⟨ Globals in the outer block 7 ⟩ ≡

```
history: error_level;  (* how bad was this run? *)
```

See also sections 12 and 18.

This code is used in section 2.

8. ⟨ Set initial values 8 ⟩ ≡

```
history ← spotless;
```

See also section 17.

This code is used in section 3.

9. Some implementations may wish to pass the value of the *history* variable to the operating system so that it can be used to govern whether or not other programs are started. The *doscall* procedure passes a program status value back to DOS. We use *fatal_error* to terminate the program abnormally.

```
define print_fatal_message ≡
            print_string('(That␣was␣a␣fatal␣');
        print_ln('error,␣my␣friend.)')
define fatal_error(#) ≡ mark_fatal; print_ln(#);
        print_fatal_message; doscall("4C, history);
```

⟨ Terminate program, converting *history* to program exit status 9 ⟩ ≡

```
doscall("4C, history);
```

This code is used in section 24.

10. If we are going to use *doscall* from the Logitech library we have to import it from the module *system*.

⟨ Import List 4 ⟩ +≡

```
from system import doscall;
```

11. File Handling. Here we define the symbols for use with file handling.

```
define lookup ≡ ␣Lookup␣
        (* library procedure to open a file *)
define close ≡ ␣Close␣
        (* library procedure to close a file *)
define failure(#) ≡ (#.res ≠ done)
        (* last file operation sucessful ? *)
define abort_if_open_error(#) ≡
      if failure(#) then
        print_string('unable␣to␣open␣');
        fatal_error(filename);
      end ;
define open_input_file(#) ≡ lookup(#, filename,
        false); abort_if_open_error(#)
define open_output_file(#) ≡ lookup(#, filename,
        true); abort_if_open_error(#)
define close_file(#) ≡ close(#);
define end_file ≡ eof
define null_char ≡ ␣nul␣
define end_of_line(#) ≡ (ch = eol)
define end_of_file(#) ≡ (#.end_file)
define read_char ≡ ␣ReadChar␣
define write_char ≡ ␣WriteChar␣
define input_char(#) ≡ read_char(#, ch);
define output_char(#) ≡ write_char(#, ch)
define read_ln(#) ≡
      while ¬end_of_line(#) do
        input_char(#);
      end ;
define write_ln(#) ≡ write_char(#, eol);
define text_file ≡ ␣File␣
```

⟨ Import List 4 ⟩ +≡

```
from ␣FileSystem␣ import lookup, Response,
    read_char, write_char, text_file, close;
from ascii import eol, null_char;
```

12. Input goes into an array called *buffer*.

⟨ Globals in the outer block 7 ⟩ +≡

```
buffer: array [0 .. buf_size] of char;
TeX_file, out_TeX_file: text_file;
```

13. The *input_ln* procedure brings the next line of input from the specified file into the *buffer* array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false*. Under normal conditions, we will never reach true end of file, for reasons discussed in later sections, but we will handle it anyway. Trailing blanks are ignored and the global variable *limit* is set to the length of the line. The value of *limit* must be strictly less than *buf_size*.

```
procedure input_ln(var f : text_file): boolean;
        (∗ inputs a line or returns false ∗)
  var final_limit: [0 .. buf_size];
        (∗ limit without trailing blanks ∗)
    ch: char;  (∗ current input character ∗)
    line_pres: boolean;
        (∗ temporary result of procedure ∗)
  begin
    limit ← 0; final_limit ← 0;
    if end_of_file(f) then
      line_pres ← false
    else
      input_char(f);
      while ¬end_of_line(f) do
        if ch = null_char then
          return false
        end ;
        buffer[limit] ← ch; inc(limit);
        if buffer[limit − 1] ≠ ´␣´ then
          final_limit ← limit
        end ;
        if limit = buf_size then
          read_ln(f); dec(limit);
          err_print(´!␣Input␣line␣too␣long´);
          return true;
        end ;
        input_char(f);
      end ;
      read_ln(f); limit ← final_limit;
      line_pres ← true;
    end ;
    return line_pres;
  end input_ln;
```

14. The *output_ln* procedure writes the next line of output from the *buffer* array to the specified file.

```
procedure output_ln(var f : text_file);
        (∗ outputs a line ∗)
  var ch: char;  (∗ current output character ∗)
    temp: [0 .. buf_size];
  begin
    if limit > 0 then
      for temp ← 0 to limit − 1 do
        ch ← buffer[temp]; output_char(f);
      end ;
    end ;
    write_ln(f);
  end output_ln;
```

15. We define *filename* local to the initalization procedure because it is used only during file open.

⟨Local variables for initialization 15⟩ ≡
filename: **array** [0 .. *file_name_len* − 1] **of** *char*;

This code is used in section 3.

16. In this section we open both of the files.

```
define next_file(#) ≡ filename[0] ← ´␣´;
        print_ln(#); read_string(filename);
        new_line; print_ln(filename); new_line;
```

```
⟨Initialize the file system 16⟩ ≡
  next_file(´TeX␣file:´);
  open_input_file(TeX_file);
  next_file(´output␣TeX␣file:´);
  open_output_file(out_TeX_file);
```

This code is used in section 3.

17. Here we initialize most of the variables. The *output_enabled* flag is initialized to *true* so that the lines in the header of the WEAVE-generated TEX file, known as "limbo text", are picked up in addition to the changed sections.

```
⟨Set initial values 8⟩ +≡
  TeX_line ← 0; out_TeX_line ← 0; limit ← 0;
  buffer[0] ← ´␣´; input_has_ended ← false;
  output_enabled ← true;
```

18. ⟨Globals in the outer block 7⟩ +≡

```
TeX_line: cardinal;  (∗ the number of the current
        line in the main TEX file ∗)
out_TeX_line: cardinal;  (∗ the number of the line
        in the output TEX file ∗)
limit: [0 .. buf_size];  (∗ the last character position
        occupied in the buffer ∗)
input_has_ended: boolean;
        (∗ there is no more input ∗)
output_enabled: boolean;
        (∗ we are copying input lines to output ∗)
```

19. Main Input Loop. This is the main processing loop of SCANTEX. We simply read lines until end of file. The *get_line* procedure will determine the setting of the *output_enabled* flag. If set, we copy the line to the output file.

```
⟨Read the input 19⟩ ≡
  while ¬input_has_ended do
    get_line;
    if output_enabled then
      output_ln(out_TeX_file); inc(out_TeX_line);
    end ;
  end ;
```

This code is used in section 24.

20. The *get_line* procedure is called to read in the next line and scan it. We will output an "I'm alive!" dot to the terminal every 100 input lines and a new line every 2000.

```
procedure get_line;   (* inputs the next line *)
  var keep_looking: boolean; temp_index: cardinal;
  begin
    input_has_ended ← ¬input_ln(TeX_file);
    if input_has_ended then
      output_enabled ← false; return ;
    else
      inc(TeX_line);
      if (TeX_line mod 100) = 0 then
        print_char(´.´);
        if (TeX_line mod 2000) = 0 then
          new_line;
        end ;
      end ;
      ⟨Scan the line 21⟩;
      buffer[limit] ← ´␣´;
    end ;
  end get_line;
```

21. In this section we determine whether the current line is the beginning of a section ('\M' or '\N' at beginning of line, followed immediately by a section number) and, if so, whether the section has been modified ('*.' following the number). We update the *output_enabled* flag according. Additionally, the index section ('\inx') is considered end of file. If it is detected, we set the flag *input_has_ended* to terminate the program and set *output_enabled* to false to keep the \inx command from being copied to the output file.

```
define numeric_digit_at(#) ≡ ((buffer[#] ≤ ´9´)
        ∧ (buffer[#] ≥ ´0´))
define third_char_matches(#) ≡
        (buffer[temp_index + 2] = #)
define second_char_matches(#) ≡
        (buffer[temp_index + 1] = #) ∧
        third_char_matches
define char_matches(#) ≡ (buffer[temp_index] =
        #)
define three_chars_match(#) ≡ char_matches(#)
        ∧ second_char_matches
```

```
⟨Scan the line 21⟩ ≡
  temp_index ← 1;
  if (limit > 3) ∧ (buffer[0] = ´\´) then
    if (char_matches(´M´) ∨ char_matches(´N´)) ∧
          numeric_digit_at(2) then
      ⟨Search for '\*.'; set output_enabled if
          found 22⟩;
    elsif three_chars_match(´i´)(´n´)(´x´) then
      output_enabled ← false;
      input_has_ended ← true;
    end ;
  end ;
```

This code is used in section 20.

22. Starting at the first digit of the section number, search for '*.', which indicates that this is a changed section. Discontinue the search if '*.'is found or the current position is no longer a numeric digit, which means we have moved past the section number without finding it.

```
⟨Search for '\*.'; set output_enabled if found 22⟩ ≡
  output_enabled ← false; keep_looking ← true;
  temp_index ← 3;
  while (¬output_enabled) ∧ keep_looking do
    output_enabled ←
        three_chars_match(´\´)(´*´)(´.´);
    keep_looking ← numeric_digit_at(temp_index);
    inc(temp_index);
  end ;
```

This code is used in section 21.

23. The command to generate the table of contents ('\con') is normally the last line in a TeX file generated by WEAVE. Part of its function is to terminate TeX gracefully by generating a '\bye' command or equivalent after generating the contents. Since we are dropping the '\con' command, we must issue the '\bye' command directly, just before closing the input and output files.

```
⟨Add '\bye' command to end of output and close
      both files 23⟩ ≡
  buffer[0] ← ´\´; buffer[1] ← ´b´;
  buffer[2] ← ´y´; buffer[3] ← ´e´; limit ← 4;
  output_ln(out_TeX_file); inc(out_TeX_line);
  close_file(TeX_file); close_file(out_TeX_file);
```

This code is used in section 24.

24. Main Program. This is the main program.

```
begin
  print_ln(banner); initialize;
  ⟨Read the input 19⟩;
  ⟨Add '\bye' command to end of output and
      close both files 23⟩;
  ⟨Print statistics about line counts 26⟩;
  ⟨Print the job history 25⟩;
  ⟨Terminate program, converting history to
      program exit status 9⟩;
end scan_TeX.
```

25. Here we simply report the history to the user.

```
⟨Print the job history 25⟩ ≡
  case history of
  spotless: print_ln(´(No␣errors␣were␣found.)´)
      |
  harmless_message:
      print_string(´(Did␣you␣see␣the␣´);
    print_ln('warning␣message␣above?') |
  error_message:
        print_string('(Pardon␣me,␣but␣I␣think␣I␣');
    print_ln('spotted␣something␣wrong.') |
  fatal_message: print_fatal_message
end ; (∗ there are no other cases ∗)
```

This code is used in section 24.

26. ⟨ Print statistics about line counts 26 ⟩ ≡

```
  new_line; print_ln('Line␣count␣statistics:');
  print_cardinal(TeX_line);
  print_ln('␣lines␣in␣input␣TeX␣file');
  print_cardinal(out_TeX_line);
  print_ln('␣lines␣in␣output␣TeX␣file');
```

This code is used in section 24.

27. Index.

⟨ Add '\bye' command to end of output and close both files 23 ⟩ Used in section 24.
⟨ Constants in the outer block 5 ⟩ Used in section 2.
⟨ Globals in the outer block 7, 12, 18 ⟩ Used in section 2.
⟨ Import List 4, 10, 11 ⟩ Used in section 2.
⟨ Initialize the file system 16 ⟩ Used in section 3.
⟨ Local variables for initialization 15 ⟩ Used in section 3.
⟨ Print statistics about line counts 26 ⟩ Used in section 24.
⟨ Print the job *history* 25 ⟩ Used in section 24.
⟨ Read the input 19 ⟩ Used in section 24.
⟨ Scan the line 21 ⟩ Used in section 20.
⟨ Search for '*.'; set *output_enabled* if found 22 ⟩ Used in section 21.
⟨ Set initial values 8, 17 ⟩ Used in section 3.
⟨ Terminate program, converting *history* to program exit status 9 ⟩ Used in section 24.
⟨ Types in the outer block 6 ⟩ Used in section 2.

Fonts

Blacker Thoughts

John S. Gourlay
Ohio State University

Like many owners of write-white laser printers, I found a few months ago that the "cm" series of Computer Modern fonts, as distributed, is unacceptably faint on my Xerox 2700. I began my search for more suitable METAFONT parameter settings with the "conjectural" settings for QMS printers, which share the same print engine as the 2700. I was immediately disappointed, however. Printed, the new bitmaps were acceptably black, but they didn't look anything like the Computer Modern in *Computer Modern Typefaces*, and not even very much like the original bitmaps printed on a write-black Canon engine.

Laser printers work by producing patterns of electric charge on a piece of paper. The charge attracts particles of black "toner," which eventually forms a permanent printed image. Write-black laser printers start with an uncharged piece of paper and in effect use a laser to place spots of charge on the paper. Write-white laser printers start with a fully charged piece of paper and then use a laser to remove the charge in places where the final image should remain white. In both cases the round spot produced by the laser is slightly larger than a pixel so that no gaps are left between spots in solid regions of black or white. For this reason, lines drawn on a write-black laser printer tend to be slightly thicker than one would expect given their width in pixels, and lines drawn on a write-white printer tend to be slightly thinner (the "white lines" are thicker).

The plain base file of METAFONT anticipates this kind of systematic difference between printers by providing a parameter called *blacker* whose value can be added to the thickness of pen strokes to compensate for any thinning inherent in the printing process. After some experimentation with various settings of *blacker* I decided empirically that the higher I made the value of *blacker* the smaller I found such lowercase letters as o and e to become. Also decreasing were the sizes of the bowls of such letters as p and b, the widths of m, n, and the lower part of h. The overall impression was that the "x-height" of the font was decreasing as *blacker* increased. At the conjectured setting of *blacker* = .75, the effect was great enough to make the font look entirely different and much less legible than the model in *Computer Modern Typefaces*.

Once I saw the problem it wasn't hard to see why it was happening. Looking at the METAFONT code for the roman lowercase o, one can see that it is drawn with a variable-width pen moving along a path through four points at the character's top, left, bottom, and right. Concentrating on point 1, the top point of the o, the relevant METAFONT statements are

$$penpos_1(vair, 90);$$

and

$$y_{1r} = h + \mathrm{vround}\, 1.5oo;$$

The first says that the pen at point 1 has a nib of width *vair* and it is held vertically with the "right" edge of the nib at the top. The second says that the right (or top) edge of the pen should be at a distance $h + \mathrm{vround}\, 1.5oo$ from the baseline. The parameter *blacker* figures into this because the pen width, *vair*, increases as *blacker* increases. Since the location of the top edge is fixed, an increase in *blacker* causes the whole pen to move down, and all the extra width appears at the bottom edge of the pen stroke. The same thing happens at the sides and bottom of the o, so the overall effect of an