Pascal pretty-printing: an example of "preprocessing within TEX"

Jean-luc Doumont
JL Consulting, Watertorenlaan 28, B-1930 Zaventem, Belgium
j.doumont@ieee.org

Abstract

Pretty-printing a piece of Pascal code with TEX is often done via an external preprocessor. Actually, the job can be done entirely in TEX; this paper introduces PPP, a Pascal pretty-printer environment that allows you to typeset Pascal code by simply typing `\Pascal` ⟨*Pascal code*⟩ `\endPascal`. The same approach of "preprocessing within TEX" — namely two-token tail-recursion around a `\FIND`-like macro — can be applied easily and successfully to numerous other situations.

Introduction

A pretty-printed piece of computer code is a striking example of how the typeset form can reveal the contents of a document. Because the contents are rigorously structured, an equally rigorous typeset form helps the reader understand the logic behind the code, recognize constructs that are similar, and differentiate those that are not. Not surprisingly, many programming environments nowadays provide programmers with a pretty-printed representation of the code they are working on. In the typesetting world, TEX seems an obvious candidate for a pretty-printing environment, thanks to its programming capabilities and its focus on logical — rather than visual — design.

The current standard for typesetting Pascal code with TEX seems to be TGRIND, a preprocessor running under UNIX. Useful as it may be, TGRIND also has limitations. While it can recognize reserved words, it does little to reflect logical content with indentation. In fact, it indents by replacing spaces in the original file by fixed `\hskip`'s. Of course, it can be used on the result produced by an ASCII-oriented pretty-printer, which generates the right number of spaces according to logical contents.

Alternatives to TGRIND are either to develop a dedicated preprocessor — a computer program that takes a piece of Pascal code as input and produces a TEX source file as output — or to do the equivalent of the preprocessing work within TEX. The first solution is likely to be faster, hence more convenient for long listings, but requires an intermediate step and is less portable. The second, by contrast, is rather slow, but also quite convenient: pieces of Pascal code can be inserted (`\input`) *as is* in a TEX document, or written directly within TEX. This solution is portable (it can run wherever TEX runs), requires no intermediate step (it does its job whenever the document is typeset), and, like other sets of macros, can be fine-tuned or customized to personal preferences while maintaining good logical design.

This article describes briefly the main features and underlying principles of PPP, a Pascal pretty-printing environment that was developed for typesetting (short) pieces of Pascal code in engineering textbooks. It then discusses how to use the same principles of "preprocessing within TEX" to quickly build other sets of macros that gobble up characters and replace them with other tokens, to be further processed by TEX. The complete PPP macro package will soon be found on the CTAN archives.

Of course, there are other ways of tackling the issue, with either a broader or a narrower scope. Structured software documentation at large can benefit from the literate programming approach and corresponding tools, with TEX or LATEX as a formatter — a discussion beyond the scope of this paper. Occasional short pieces of code, on the other hand, can also be typeset verbatim or with a few *ad hoc* macros, for example a simple tabbing environment, as shown by Don Knuth (1984, page 234). For additional references, see also the compilation work of Piet van Oostrum (1991).

Main features of macros

Basic use. PPP works transparently; you do not need to know much to run it. After `\input`ing the macros in your source, all you do is write

`\Pascal`
⟨*Pascal code*⟩
`\endPascal`

in `plain.tex` or

```
\begin{Pascal}
⟨Pascal code⟩
\end{Pascal}
```

in LaTeX, where ⟨*Pascal code*⟩ can be an `\input` command.

The PPP package then pretty-prints the corresponding Pascal code; by default, it

- typesets reserved words in boldface;
- indents the structure according to syntax (identifying such constructs as **begin** ... **end** and **if** ... **then** ... **else** ...;);
- typesets string literals in monospaced (`\tt`) font;
- considers comments to be TeX code and typesets them accordingly.

The Appendix illustrates these features.

Comments. Recognizing comments as TeX code is particularly powerful: side by side with a rather strict typeset design for the program itself, comments can be typeset with all of TeX's flexibility and power. Besides for adding explanatory comments to the program, this possibility can be used to fine-tune the layout. Extra vertical space and page breaks can be added in this way. Such comments can even be made "invisible", so no empty pair of comment delimiters shows on the page.

> Accessing TeX within comments suffers a notable exception, though. Pascal comments can be delimited with braces, but Pascal compilers *do not match braces*: the first opening brace opens the comment and the first closing brace closes the comment, irrespective of how many other opening braces are in between. As a consequence, braces cannot be used for delimiting TeX groups inside Pascal comments (the result would not be legal Pascal code anymore). Other TeX delimiters must be used; by default, PPP uses the square brackets '`[`' and '`]`'.

Program fragments. PPP was taught the minimum amount of Pascal syntax that allows it to typeset Pascal code; it is thus not a syntax-checker. While some syntax errors (such as a missing **end**) will cause incorrect or unexpected output, some others (such as unbalanced parentheses) will be happily ignored.

However, the package was designed for inserting illustrative pieces of code in textbooks, including *incomplete* programs. PPP has facilities for handling these, though it needs hints from the author as to what parts are missing. These hints basically consist in supplying — in a hidden form — the important missing elements, so PPP knows how many groups to open and can then close them properly.

Customization. PPP is dedicated to Pascal. Though you can use the same underlying principles (see next section) in other contexts, you cannot easily modify PPP to pretty-print very different programming languages. There is, however, room for customizing the pretty-printing, and this at several levels.

At a high level, you can use the token registers `\everyPascal`, `\everystring`, as well as `\everycomment` to add formatting commands to be applied, respectively, to the entire Pascal code, to the Pascal string literals, and to the Pascal comments. If you want your whole Pascal code to be in nine-point roman, for example, you can say

```
\everyPascal{\ninerm
             \baselineskip=10pt⟨etc.⟩}
```

If you would rather use '`(`' and '`)`' instead of '`[`' and '`]`' as TeX grouping delimiters in Pascal comments, you can say

```
\everycomment{\catcode`\(=1 \catcode`\)=2}
```

Similarly, if you wish to reproduce the comments verbatim rather than consider them as TeX code, you can say

```
\everycomment{\verbatimcomments}
```

At an intermediate level, you can add reserved words by defining a macro with the same name as the reserved word prefixed with `p@`. If you want the Pascal identifier `foo` to be displayed in italics in your code, you can say

```
\def\p@foo{{\it foo}}
```

before your code and PPP will do the rest.

At a low level, you can go and change anything you want, providing you know what you are doing *and* you first save PPP under a different name.

Underlying principles

The PPP environment pretty-prints the code in one pass: it reads the tokens, recognizes reserved words and constructs, and typesets the code accordingly, indenting the commands according to depth of grouping. Specifically, PPP

- relies on tail-recursion to read a list of tokens: one main command reads one or several tokens, processes them, then calls itself again to read and process subsequent tokens until it encounters a stop token;
- decides what to do for each token using a modified version of Jonathan Fine's `\FIND` macro;
- recognizes words as reserved by checking for the existence of a TeX command with the corresponding name and acts upon reserved words by executing this command;

- typesets the code by building a nested group structure in TeX that matches the group structure in Pascal.

Tail-recursion. Jonathan Fine (1993) offers useful control macros for reading and modifying a string of tokens. Rewritten with a ';' instead of a '*' (to follow the Pascal syntax for a **case**), his example for marking up vowels in boldface, a problem introduced in *Einführung in TeX* by Norbert Schwarz (1987), becomes:

```
\def\markvowels #1
{
  \FIND #1
    \end:;
    aeiou AEIOU:{\bf#1}\markvowels;
    #1:{#1}\markvowels;
  \END
}
```

so that `\markvowels Audacious \end` produces "**A**u**d**a**c**i**o**u**s**". `\FIND` is a *variable delimiter macro* (as Fine puts it), defined as

```
\long\def\FIND #1{%
  \long\def\next##1#1##2:##3;##4\END{##3}%
  \next}
```

It extracts what is between the ':' and the ';' immediately following the first visible instance of #1 and discards whatever is before and whatever is after (up to the following `\END`). The same idea is used in the *Dirty Tricks* section of the *The TeXbook* (Knuth 1984, page 375). The generic use of `\FIND` is thus

```
\FIND ⟨search token⟩
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ...
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ⟨search token⟩:⟨default action⟩;
\END
```

PPP brings the following three basic changes to Fine's scheme:

- first, it uses a tail-recursion scheme that reads tokens two by two rather than one by one; this extension makes it easier to recognize and treat character pairs such as '>=', '..', and '(*'.
- next, it moves the tail-recursion command (the equivalent of `\markvowels` in the example above) to the end of the macro, to avoid having to repeat it for each entry in the `\FIND` list. This move also simplifies brace worries: whatever is specified between the ':' and the ';' in the above definition can now be enclosed in braces. These protect a potential #1 in the ⟨*action*⟩ (they make it invisible when `\next` scans its argument list), but do not produce an extra level of grouping (they are stripped off when `\next` reads its argument #3).
- finally, it replaces ':' and ';' — which need to be recognized explicitly when reading the Pascal code — respectively by '?' and '!' — which do not. (Other tricks are possible; see for example Sections 4 and 6 in Fine (1993).)

To consider all pairs of tokens, the new scheme spits out the second token before calling the recursive command again, so this second token is read as the first token of the new pair. While this double-token system has proved very convenient in many applications I developed, it has one inherent limitation: because the spit-out character has been into TeX's mouth, it has already been tokenized (assigned a character code). If the action corresponding to the first token read is to redefine character codes, then the second token will not reflect these new codes. When such a recoding is an issue, alternative constructs using `\futurelet` can be devised to consider pairs (i.e., to take the next token into account in deciding what to do), but such constructs are rather heavy.

With these changes, the tail-recursion core of the Pascal pretty-printer looks something like this:

```
\long\def\Find #1{
  \long\def\next##1#1##2?##3!##4\END{##3}
  \next}

\def\Pascal{\pascal \relax}
  % \relax is passed as first token
  % in case the code is empty
  % i.e., the next token is \endPascal

\def\pascal#1#2{\def\thepascal{\pascal}%
  \Find #1
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    ...
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    #1?{⟨default action⟩}!
  \END
  \ifx\endPascal#2
    \def\thepascal##1{\relax}\fi
  \thepascal#2}
```

with the typesetting taking the form

```
\Pascal ⟨Pascal code⟩ \endPascal
```

In this two-token scheme, the end-of-sequence test must now be done on the second token read, so the tail recursion does not read past the end-of-sequence token (`\endPascal`). The sequence is ended by redefining `\thepascal` to gobble the next token and do nothing else.

Hmm ... it is a little more complicated than that. The `\pascal` macro (which is really called `\p@sc@l`) must be able to recognize and act upon

braces, used as comment delimiters in Pascal. These braces are recatcoded to the category `other` by saying `\catcode`\{=12 \catcode`\}=12` somewhere in `\Pascal`, so they lose their grouping power when TEX scans Pascal code. Because the `\FIND` macro identifies *tokens*, category codes must match. In other words, '{' and '}' must be of category 12 when `\p@sc@l` is defined, so we must use another pair of characters as group delimiters for defining `\p@sc@l`. I use the square brackets '[' and ']'.

Accumulating words. Identifiers in Pascal are composed of letters, digits, and the underscore character '_', but must start with a letter. Correspondingly, PPP identifies words in the following way. It uses an `\ifword` switch to indicate whether a word is currently constructed and an `\ifreserved` switch to indicate whether the accumulated word is a candidate reserved word. Starting on a situation in which `\ifword` is false, it does the following:

- if the token read is a letter, set `\ifword` and `\ifreserved` to true, empty the token register `\word`, and accumulate the letter in it.
- if the token read is a digit, look at `\ifword`. If true, accumulate the digit in the token register `\word` and set `\ifreserved` to false (reserved words contain no digit); if false, treat as a number.
- if the token read is an underscore, look at `\ifword`. If true, accumulate the underscore in the token register `\word` and set `\ifreserved` to false (reserved words contain no underscore); if false, treat as an underscore.
- if the token is not a letter, a digit, or an underscore, look at `\ifword`. If true, set to false and take care of the word so terminated. If false, pass token to other macro for further processing.

Recognizing reserved words. PPP recognizes reserved words by checking words composed of letters only against a list. This list is in reality a set of macros, the names of which are formed by prefixing Pascal reserved words with 'p@'. These macros have thus a double role:

- by their existence, they identify a word as reserved; for example, the existence of a macro named `p@begin` indicates that **begin** is a reserved word.
- by their definition, they tell what to do when the corresponding reserved word has been identified; for example, `\p@begin` takes care of what needs to be done when the reserved word **begin** is encountered.

The actions to perform when a reserved word has been identified depend of course on the word, but are within a small set, namely

- typesetting the word as a reserved word, possibly with space before or after;
- opening a group and increasing the indentation;
- closing a group, thus going back to the level of indentation present when that group was opened; or
- turning flags on or off.

Because many reserved words require the same action, the corresponding TEX macros can all be `\let` equal to the same generic macro. For example, `\r@serv` simply typesets the last reserved word accumulated (without extra space), so reserved words like **string** or **nil** can be taken care of simply by saying

```
\let\p@string=\r@serv
\let\p@nil=\r@serv
```

Grouping and indenting. PPP manages the levels of indentation by creating a nested group structure that matches the structure of the program. A **begin**, for example, opens a group and increments the indentation by one unit within the group; an **end** closes the group, thus returning to the level of indentation in effect before the group was opened.

Of course, grouping is not always that simple. All the declarations that follow a **var**, for example, should be within an indented group, but there is no reserved word to mark the end of the group. Such cases are treated by setting a flag to true, to indicate that a group without terminator is open. The next of a subset of reserved words can then close that group before performing its own task.

Other examples of "preprocessing"

A tail-recursion engine based on a `\FIND`-like macro does pretty much what one would expect a preprocessor to do: it gobbles the characters one by one and replaces them with other, possibly very different tokens. This similarity is what leads me to refer to such a scheme as "preprocessing within TEX" (though, strictly speaking, this is a contradiction in terms).

The one-token examples presented in Fine (1993) are the simplest case of this preprocessing: decisions are taken each time on the basis of a single token. Such a scheme is simple, straightforward, and sufficient in many applications. And when following tokens must be taken into account, it can be extended with `\futurelet` constructs, though these quickly become quite heavy. For

example, the \markvowels macro can be modified in the following way to mark, say, "i before e" combinations:

```
\def\spellcheck#1{%
  \FIND #1
    \end:;
    i:\ie;
    #1:{#1}\spellcheck;
  \END}

\def\ie{\futurelet\nextchar\iiee}
\def\iiee{\let\etemp=e%
  \ifx\etemp\nextchar
    {\bf ie}%
    \let\temp\gobblenexttoken
  \else
    i%
    \let\temp\spellcheck
  \fi
  \temp}

\def\gobblenexttoken#1{\spellcheck}
```

so that typing

```
{\obeyspaces
\spellcheck  I receive a piece of pie\end}
```

yields "I receive a p**ie**ce of p**ie**".

The two-token example presented in this paper is a convenient extension of the scheme. True, it has as inherent limitation that the second parameter is tokenized (assigned a character code) one step earlier than it would in the one-token case. On the other hand, the corresponding code is particularly readable (thus easy to program and easy to maintain). The above example becomes, with a two-token model,

```
\def\check#1#2{\def\nextcheck{\check}%
  \FIND #1
    i:{\FIND #2
         e:{{\bf ie}\gobbleone};
         #2:{i};
       \END};
    #1:{#1};
  \END
  \ifx\end#2
    \def\nextcheck##1{\relax}\fi
  \nextcheck#2}

\def\gobbleone{\def\nextcheck##1%
  {\check \relax}}
\def\spellcheck{\check \relax}
```

where \gobbleone gobbles the next token and replaces it with \relax. The nested \FIND structure makes it easy to see the underlying idea of "once you know the first letter is an i, see whether the second is an e". Clearly, the mechanism can be extended to take into account three, four, or even more tokens at the same time, with limitations and advantages similar to those in the two-token case.

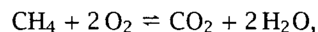
Two-token tail-recursion can also be achieved with other constructs, for example Kees van der Laan's \fifo macro. In van der Laan (1993) he underlines the importance of the *separation of concerns*: going through the list is separated from processing each element of the list. This elegant programming principle is sometimes hard to achieve in practice: in the case of string literals, for example, \Pascal reacts to a single quote by interrupting token-by-token progression and reading all tokens to the next single quote — progressing and processing are thus closely linked. For the "i before e" example, the separation is clearer and the use of the \FIND structure for processing the elements is largely unchanged:

```
\def\fifo#1#2{\check#1#2%
  \ifx\ofif#2\ofif\fi\fifo#2}
\def\ofif#1\ofif{\fi}

\newif\ifgobbleone

\def\check#1#2{\ifgobbleone
  \gobbleonefalse
  \else
  \FIND #1
    i:{\FIND #2
         e:{{\bf ie}\gobbleonetrue};
         #2:{i};
       \END};
    #1:{#1};
  \END
  \fi}
```

I have used the two-token scheme successfully in a variety of situations. For the same engineering textbook format, I devised an elementary chemistry mode, so that

```
\chem CH4+2O2<>CO2+2H2O \endchem
```

yields



$$\mathrm{CH_4} + 2\,\mathrm{O_2} \rightleftharpoons \mathrm{CO_2} + 2\,\mathrm{H_2O},$$

and a unit mode, so that

```
\unit 6.672,59e-11 m3.kg-1\endunit
```

yields the ISO representation

$$6.672\,59 \times 10^{-11}\,\mathrm{m^3{\cdot}kg^{-1}}.$$

Actually, mentioning that the \FIND-like tail-recursion applies to tokens is not entirely correct. Because it reads arguments, it will also gobble up as one object a group delimited by braces (or by the current TeX delimiters), not a single token. This case cannot happen with the \Pascal macro, for there are no current group delimiters during tail-recursion ('{' and '}' are given category code 12), but it can happen in other situations. When a group is read as argument #1 by \check, the first level of grouping is removed, so the \FIND selection is actually performed on the first token (or group) within the original group. Whether this characteristic is a

feature or a bug depends on your application. Sometimes, it is quite useful: for the chemistry mode above, it enables `\chem C{60}H{60}\endchem` to give the correct output $C_{60}H_{60}$, with the `\FIND` recognizing the '6', but acting on the group 60; by contrast, `\chem C60H60\endchem` yields the incorrect $C_{6}{}_{0}H_{6}{}_{0}$, with the '0' being a subscript to an empty subformula and hence being too far away from the '6'. Sometimes, however, you may prefer strict token-per-token processing; in the FIFO paper mentioned above, Kees van der Laan shows a way of acting on each token by assigning it to a temporary variable instead of reading it as an argument.

Conclusion

Preprocessing within TEX—reading a list of tokens (or brace-delimited groups) and replacing them with others for TEX to process further—has unlimited applications for TEX users and macro-writers. A processing based on a `\FIND` macro (Fine, 1993) is powerful, especially when nested and applied on two tokens. The progression along the list can be built in the same macro or can be separated, for example using the `\fifo` macro (van der Laan 1993). The approach is powerful enough to handle such tasks as pretty-printing of Pascal code fragments.

Maybe the main advantage of these preprocessing schemes is that they are fast and easy to implement. They are not reserved to large-scope application, but can be used for one-off, *ad hoc* macros as well. I once had to typeset phone numbers on the basis of the following syntax: the code

```
\phone{725.83.64}
```

should yield 725 83 64, that is, periods must be replaced by thin spaces and pairs of digits must be slightly kerned (it looked better for the particular font at that particular size). The corresponding tail-recursion scheme is easy to implement:

```
\catcode'\@=11

\def\k@rn#1#2{\let\thek@rn=\k@rn
  \FIND #1
    0123456789:{#1%
      \FIND #2
        0123456789:{\kern-0.0833em};
        #2:{\relax};
      \END};
    .:{\thinspace};
    #1:{#1};
  \END
  \ifx\end#2\def\thek@rn##1{\relax}\fi
  \thek@rn#2}

\def\phone#1{\k@rn#1\end}

\catcode'\@=12
```

It may not be the fastest piece of TEX code in the world (and some would doubtlessly qualify it as "syntactic sugar"), but it made optimal use of my time, by allowing me to get the job done fast and well.

Bibliography

Fine, Jonathan, "The \CASE and \FIND macros." *TUGboat* **14** (1), pages 35–39, 1993.

Knuth, Donald E., *The TEXbook*. Reading, Mass.: Addison-Wesley, 1984.

Laan, C. G. van der, "\FIFO and \LIFO sing the BLUes." *TUGboat* **14** (1), pages 54–60, 1993.

Laan, C. G. van der, "Syntactic sugar." *TUGboat* **14** (3), pages 310–318, 1993.

Oostrum, Piet van, "Program text generation with TEX/LATEX." MAPS91.1, pages 99–105, 1991.

Schwarz, Norbert, *Einführung in TEX*. Addison-Wesley, Europe, 1987. Also available as *Introduction to TEX*. Reading, Mass.: Addison-Wesley, 1989.

Appendix: example of use

(The following program may not be particularly representative of code fragments inserted in a textbook with the PPP package, but it has been designed to illustrate as many features of the \Pascal environment as possible.)

```
% Filename="ppptest"

\input ppp

\everycomment={\sl}

\Pascal

Program demo;
const pi=3.141592;
type date=record year: integer;
month:1..12; day:1..31;end;
flags=packed array[0..7] of boolean;
var MyDate:date; MyFlags:flags;
i1,i2:integer;
last_words:string[31];

function factorial (n:integer):integer;
begin
if n<=1 then factorial:=1
else factorial:=n*factorial(n-1);
end;
{\invisible\vadjust[\medskip[\it
$\langle$more code here$\rangle$]\medskip]}

function Days_in_month(theDate:date);
begin
case theDate.month of 1:Days_in_Month:=31;
2:with theDate
do {check if leap year} begin
 if (0=(year mod 4))
then Days_in_Month:=29 else
Days_in_Month:=28;end;
3:Days_in_Month:=31;{\invisible
\vadjust[\hbox[\hskip8em$\vdots$]]}
12:Days_in_Month:=31;
end;

begin
last_words:='That''s all, folks';
end.{Et voil\'a\thinspace!}

\endPascal
```

program demo;
const
  pi = 3.141592;
type
  date = **record**
      year: integer;
      month: 1..12;
      day: 1..31;
    end;
  flags = **packed array**[0..7] **of** boolean;
var
  MyDate: date;
  MyFlags: flags;
  i1, i2: integer;
  last_words: **string**[31];

function factorial(n: integer): integer;
begin
  if n <= 1 **then**
    factorial := 1
  else
    factorial := n ∗ factorial(n − 1);
end;

⟨*more code here*⟩

function Days_in_month(theDate: date);
begin
  case theDate.month **of**
    1:
      Days_in_Month := 31;
    2:
      with theDate **do**{*check if leap year*}
        begin
          if (0 = (year **mod** 4)) **then**
            Days_in_Month := 29
          else
            Days_in_Month := 28;
        end;
    3:
      Days_in_Month := 31;
              ⋮
    12:
      Days_in_Month := 31;
  end;

  begin
    last_words := 'That''s all, folks';
  end.{*Et voilà !*}