1400191301
(copia 1

Prettyprint: A simple tool for typesetting algorithms

Conrado Martínez

Report LSI–94–7–T

Prettyprint: A Simple Tool for Typesetting Algorithms

Conrado Martínez*

March 16, 1994

1 Introduction

This document presents a suite of tools for producing prettyprinted algorithms using LaTeX [2]. Writing an algorithm like the one in Figure 1 using standard LaTeX can be a cumbersome activity, since one must take care of a bunch of minor details: set proper fonts for keywords and identifiers, keep track of the indentation of each line of code, etc.

For this reason, many people use LaTeX's `verbatim` environment to include algorithms and programs in their documents. Everything written inside this environment is typeset in the output with `\tt` font. All characters go to the final output as they were typed in the input. The rule also applies to white space: spaces, tabs, newlines, etc. are retained. Therefore, it is really easy to write pieces of code, but you lose the ability to include special symbols, mathematical formulas, etc.

The main tool that we present is a small program, `prettyprint`, that replaces the keywords appearing in the input by corresponding macros (for example, `if` by `\IF`) and does its best at identifying some identifiers to typeset them in a different font. In particular, it recognizes type names and procedure names, and marks them in order to typeset them in ordinary roman font. The language that one should use to express the algorithms is a mixture of Dijkstra's guarded commands and PASCAL-like constructs. All the manipulations are done using pattern-matching and some reasonable heuristics. Therefore, the program is not as reliable as a true prettyprinter

*Departament de Llenguatges i Sistemes Informàtics.Universitat Politècnica de Catalunya.Pau Gargallo 5, 08028-Barcelona, Spain.Contact e-mail: `conrado@lsi.upc.es`

```
program hello_world
    var $i, n$ : integer
    end;

    Read($n$);
    for $i := 1$ to $n$ do
       Write("Hello world!")
    end

end.
```

Figure 1: A short prettyprinted program.

can be, but on the other hand it is very short and simple. It is not very difficult to include new features and, last but not least, it is amazingly easy to extend or change the "language" used to write the code. A more complete description of the program and a short explanation about how it works is given in Section 2.

The task done by `prettyprint` is completed by means of a new LaTeX environment called `pcode` and appropiate definitions for the macros introduced in the first step by `prettyprint`. The environment `pcode` is throughly explained in Section 3. Several other useful macros and variants of the basic `pcode` environment are described in Section 4.

Although the algorithms that `prettyprint` is meant to typeset are not written in a language that can be compiled and executed, I decided to make the tools compatible with `noweb`, a language-independent literate programming tool developed by Norman Ramsey [3]. The joint use of `noweb` and `prettyprint`, as well as some technical details on the way the compatibility is achieved, is described in Section 5.

2 prettyprint

The first step of the prettyprinting process is done by a simple UNIX filter called `prettyprint`[1]. To use it type:

[1]It was recently distributed as part of `noweb` release 2.5, under the name `d2tex`.

`prettyprint` [*options*] *filename1 filename2* ...

Each of the files specified in the command-line is proccessed in turn. If no file name is given then the input comes from the standard input stream. The output of the program is printed in the standard output stream.

To understand better what the program does, it is interesting to give some details of its implementation. The script `prettyprint` is written in AWK and relies heavily on pattern-matching; in fact, it does not know anything about the syntax of the programming language that it is supposed to prettyprint.

Only those lines of the input that go inside a LaTeX environment starting with `\begin{pcode}` and ending with `\end{pcode}` are processed; the rest is directly copied to the standard output. There are three basic things that the program does to the lines inside the `pcode` environment:

1. It spots the places where special patterns (the *keywords*) occur and replaces them by corresponding TeX macros. The keywords that the program recognizes are specified in the so-called *keywords* file and the corresponding macros are specified in the *macros* file. For instance, all isolated apparitions of the string `if` are replaced by `\IF`. Similarly, `while` is replaced by `\WHILE`, `array` by `\ARRAY`, etc.

2. Since we want to typeset in roman font the identifiers of types, procedures, modules and functions, the script tries to identify these. It follows some heuristic rules to do this: if an identifier follows a semicolon ':' and another identifier precedes the semicolon, it is likely that the former is the name of a type in a variable declaration or parameter list; if an identifier precedes an equal sign '=' and then it follows the macro `\RECORD`[2] or the macro `\ARRAY` then it must be a type identifier in a type definition; if an identifier precedes an opening parentheses '(' then the identifier should be a procedure or function name in a definition or a call, etc. Whenever a type or procedure identifier is "recognized", `prettyprint` encloses it inside vertical bars; e.g. `integer` $\longrightarrow$ `|integer|` .

3. None of the rules above applies to text that is inside a comment. Everything between the symbols `/*` and `*/` is considered by `prettyprint` as a comment, and replaced by

[2]The first phase should replace the keyword record by the macro RECORD.

\COMMENT{*the original comment*}

Of course, these rules fail sometimes (specially the second one) and introduce some restrictions on the layout of the input: for instance, if one writes

```
type intvector =
         array [1..10] of integer
```

then `prettyprint` does not recognize `intvector` as the identifier of a type.

The next two excerpts show part of the input given to `prettyprint` and its corresponding output (these lines were the ones written to produce our first example in Figure 1).

If the input contains the following lines inside a `pcode` environment

```
program hello_world
var i, n : integer
endvar;

Read(n);
for i:= 1 to n do
  Write({\tt "Hello\ World"})
endfor

endprogram.
```

then the output should look like:

```
\PROGRAM |hello_world|
\VAR i, n : |integer|
\ENDVAR;

|Read|(n);
\FOR i:= 1 \TO n \DO
|Write|({\tt "Hello\ world!"})
\ENDFOR

\ENDPROGRAM.
```

The prettyprinter is really simple and easily modifiable. Also the fact that it works with two files makes it easy to use it for different "programming languages". The default *macros* and *keywords* files, to write algorithms in English, are the file NAMES and the file names, respectively.

But you can use other different files, other than the default ones. For instance, I use the file NAMES as the *macros* file and nombres as the *keywords* file, to write algorithms using keywords in Spanish. This behavior can be achieved using option -s of the prettyprinter.

I shall not describe the syntax of the "programming language" used to write the algorithms. Therefore, I strongly recommend to have a look at the *keywords* file(s) you plan to use, and the sample files in the distribution.

The macros defined in the standard file called NAMES are a kind of "intermediate" language that can be easily modified, extended, etc. The first rule that you must follow to build or modify a *macros-keywords* pair is that every macro and the keyword it translates should appear a matching positions in the corresponding file. Furthermore, keywords are not restricted to be words: for instance, the patterns [], !=, >= and many other non-alphabetic patterns are defined as keywords in the standard *keywords* files: names, nombres, noms. The second rule is that only letters can be used for the words that appear in the *macros* file since these words are names for TEX macros; in order to avoid confusions with other TEX macros, it is wise to use only uppercase letters for these macros.
Finally, the program has several available options:

-s to prettyprint code that is written using Spanish keywords

-c to prettyprint code that is written using Catalan keywords

-f *F*1 *F*2 to prettyprint code that is written using *keywords* file *F*1 and *macros* file *F*2.

3 The pcode environment

The prettyprinter translates the input to some suitable representation for the second step. Interspersed with text, figures, tables, etc. we should find some pieces that look like

```
\begin{pcode}{}
\IF
        a > b \THEN max:= a
\ELSE   a \LEQ b \THEN max:= b
\ENDIF
```

```
...
\end{pcode}
```

The definition of the macros `\IF`, `\THEN`, etc. and the `pcode` environment are the responsible for producing the following output, given the input above:

```
if
      $a > b$  $\longrightarrow$  $max := a$
    $[]$ $a \leq b$  $\longrightarrow$  $max := b$
fi
...
```

The `pcode` environment is the main definition in the style file called `algoritmos.sty`. This style file also contains several other useful definitions and variations of the `pcode` environment, that are explained in section 4.

I have been heavily influenced by Martin Ward's `program.sty` and to a less extent by George Ferguson's `code.sty`, when writing the definition of the `pcode` environment.

The specific characteristics of the environment are:

1. It has an additional parameter for cross-referencing (see Section 5).
2. There is a separation between the code and the surrounding text that can be controlled using the length parameter `\algosep`.
3. A rule is drawn at the top and bottom of the code. The width of the rules is defined by the length parameter `\algoruleheight`. Its default value is `0pt`.
4. The font size of the output is controlled by the macro `\algofontsize`. The default vaule is `\footnotesize`.
5. All lines are typeset in math mode; the exception to this rule is the text written inside a comment, which is typeset in paragraph mode.
6. The vertical bars are active characters. The sequence |*something*| prints *something* in the font specified by `\identinfont` (`\rm` is the default), if it appears anywhere inside the `pcode` environment. On the other hand, |*something*| prints *something* in the font specified by `\identoutfont` (`\sl` is the default), if it appears outside a `pcode` environment.

7. Underscore characters issued inside a pcode environment produce '_' in the output; inside any other math environment they are used for subscripts, as usual, e.g. x_2 produces x_2. Finally, they also produce an underscore if they appear outside of the pcode or math environments.

8. Newlines are obeyed; if you want to break a line in the input, but not in the output, put a character % at the end of the first line of the input.

A few comments on the specific characteristics listed above are in order. First, if you are not using these tools with the literate-programming tool called noweb, the cross-referencing parameter of the environment is useless. Nevertheless, you must write it, even if it is empty. Second, the vertical bars can be used outside the pcode environment, as a quoting mechanism. If you need the vertical bars to appear in your document, you can use one of the following alternatives: use the macro `\vert`, use the macro `\origbar` or temporarily disable the vertical bar as an active character. Third, if you want to have subscripts anywhere inside a pcode environment you must use the macro `\sb` instead of the underscore. Fourth, you can use standard LaTeX commands to produce mathematical symbols, such as `\alpha` or `\sum` inside a pcode; but do not forget, that comments are processed in paragraph mode!

It is clear that the special macros, the ones likely produced by prettyprint, appearing inside a pcode environment are defined in such a way that: 1) they produce a symbol or keyword in the appropriate font; 2) they introduce or remove some amount of indentation.

For instance, `\THEN` prints in the current definition a long right-pointing arrow ($\longrightarrow$) and sets a tab. The following lines will be indented and aligned with the right end of the arrow, unless one of them starts a new guarded command, i.e. begins with `\ELSE`. This last macro prints a small box ($[\!]$) and removes the indentation introduced by the last `\THEN` command.

The exception to the rules 1 and 2 above is the macro `\COMMENT`, that has an argument and processes it in TeX's paragraph mode. Hence, if you need to write a formula or mathematical symbol inside a comment, you must change to math mode, i.e. enclose it within $'s.

If you want to change the behavior of some macros you can redefine them at the beginning of the document. Also, you can produce a totally new set of macro definitions and forget about the standard ones.

The three-part design of the "system" (*keyword* $\rightarrow$ *macro* $\rightarrow$ *TeX definition*) allows to generate prettyprinted output that seems out of reach for a simple tool like `prettyprint`. For instance, assume that we want to use brackets to delimit conditional constructs, that is, pure Dijsktra's notation. The prettyprinter would have to know about the syntax of the language if we were using `[` and `]` in the input file, because there are several other meanings for these symbols.

But we want to keep our prettyprinter really simple; hence, we give it clues using `if` and `endif` in the input. The pattern-matching engine substitutes them by `\IF` and `\ENDIF`. Finally, a proper definition of these macros produces the desired output[3]:

```
[
  $a > b$  $\longrightarrow$  $max := a$
[] $a \leq b$  $\longrightarrow$  $max := b$
]
...
```

There are two additional LaTeX commands that are useful to specify indentation. Their names are `\tab` and `\untab`. The first one sets a tab and pushes it, so the left margin of the following lines is aligned with the tab. The other one pops and removes a tab previously set with `\tab`. Other tabbing commands can be used inside the `pcode` environment (for instance, `\>`, `\<`, etc.) but they are seldom needed. Almost all macros, like `\THEN`, `\WHILE`, `\ENDIF`, etc. are defined using `\tab` and `\untab`. `\tab` and `\untab` can also be used to override the default indentation. For example, the `\PROCEDURE` macro prints the keyword **procedure** in boldface and sets a tab between **pro** and **cedure**. The next lines in the input will be indented slightly to the right, so they are aligned with the **c** of **procedure**. Suppose that you have a very long parameter list; you want to break it into several lines and that each of these lines is aligned with the opening parentheses of the parameter list.

You can do this as follows:

```
procedure my_proc(\tab inp a : T1; inp b : T2; inp c : T3;
```

[3]For similar reasons to the one discussed here, I think is useful to have a full parenthetic "language": **endprocedure**, **endwhile**, **endfor**, ..., although in the standard definition all these keywords produce the same: remove a tab and print **end** in boldface.

```
                       in/out d : T4; in/out e : T5;
                       out f  : T6; out g : T7) \untab
var x : integer;
```

The filter `prettyprint` writes the indentation commands in the output without modifying them. The final result, after you process the original input with both `prettyprint` and LaTeX, is:

```
procedure my_proc(in $a$ : T1; in $b$ : T2; in $c$ : T3;
                  in/out $d$ : T4; in/out $e$ : T5;
                  out $f$ : T6; out $g$ : T7)
   var $x$ : integer;
```

To use the `pcode` environment you must include the option `algoritmos` in the `\documentstyle` declaration of your document. The definitions of the macros are in a different file. The standard file for keywords in English is called `keywords.tex`. To simplify the process of building or updating a set of macro definitions, there is a simple script called `genkeyw` to merge the names of macros given in a *macros* file with its definitions. The script is given two files, one with the names of the macros (for instance, `NAMES`) ending with the special name `@@@`, and the other with the matching TeX definitions. The result is a file that contains lines of the form

`\def\`<*macro name*>`{`<*definition*>`}`.

The directory `genkey` of the distribution package contains several *definitions* files. For instance, the file `keywords.tex` is the result of merging the *macros* file `NAMES` and the *definitions* file called `dkeywords`.

4 Other features of `algoritmos.sty`

The style file `algoritmos.sty` defines two variations of the basic `pcode` environment: the `*`-form and `cntpcode`.

The `pcode*` environment is much like the `pcode` environment, but it numbers sequentially each line of code. Comments that span several lines count just as a single one. The label in front of each line can be customized by redefining the macros `\codelinelabel` and `\thecodeline`. The name of the counter is `codeline`.

Another difference between the pcode and pcode* environments is that the last does not have the additional parameter for cross-referencing.

The cntpcode environment defines a minipage, centers it and puts inside a pcode environment its own body. It is useful for small chunks of code, specifications, short declarations, ... Larger pieces of code can get into trouble, since no page breaks are allowed inside a cntpcode.

The prettyprinter recognizes both the pcode* and cntpcode environments as variants of the pcode environment and applies the same manipulations to the text that appears inside them.

Another feature of the style file is the algorithm environment. This environment works as the figure and table environments, enclosing a float object that gets its own numbering. The string "Algorithm <number>:" precedes the captions and the corresponding entry is written in a .lop file. To get an index or list of the algorithms, a \listofalgorithms should be issued. Both the title of the list of algorithms and the labels of captions can be changed by redefining the macros \listofalgorithmsname and \algorithmname. It also exists an algorithm* environment for double column algorithms.

Finally, there exists a declaration \noprettyprint that makes the pcode environment and its variations behave as a verbatim environment. You can use it to avoid proccessing the file through prettyprint until the final stages. The prettyprint filter is slow (especially for long pieces of code), so it is wise to use this strategy.

5 Using noweb and prettyprint

I assume that the reader is already familiar with the concept of literate programming (LP) and the tool called noweb. If this is not the case, the seminal paper by Don Knuth [1] can be a good starting point for those that are interested in the topic. The standard reference for noweb is [3]; the LaTeX source of this last reference is available as part of the noweb distribution.

Norman Ramsey's noweb is a LP tool that does not prettyprint the chunks of code nor does it maintain an index of identifiers, etc. Therefore, it is not dependent of the programming language and is extremely easy to use it. The noweb system is an integrated set of UNIX filters that can be composed to do many different tasks. New filters can be written to provide new features, connecting them with the existing ones. Examples of extensions that several people have contributed (including noweb's author)

```
The main loop generates a random number in each iteration.
If it is different from all previously generated numbers,
we add it to the set where we keep different values, whereas
it is rejected if we had already generated it at some iteration.
The loop ends when the set contains $n$ different values.

<<generate $n$ distinct random numbers>>=

while size(C) != n do
  <<generate a random number $x$>>
  <<if $x$ is not in $C$ put it into $C$>>
endwhile
@

....
```

Figure 2: An example of input for noweave.

include tools for automatic indexing of identifiers, cross-referencing of code chunks or modules by number (not by page), prettyprinting, etc. In particular, I have tried to make prettyprint and noweb work together for two reasons: 1) To provide another working example of the advantages of Ramsey's approach to LP; 2) To be able to use several nice features of noweb, as named chunks and module cross-referencing. Moreover, prettyprint is intended to enhance the exposition of algorithms. Since the fundamental paradigm of literate programming is "write your program as an exposition for human readers", making both noweb and prettyprint compatible seems to be natural.

To use both systems in a combined way, you first write your source files using noweb's conventions and the "language" recognized by prettyprint. Figure 2 gives an example of input for noweave.

The following step is to process the source file using noweave's option -delay, since you should make sure the style file algoritmos.sty and the macros file, say keywords.tex have been included. That is, your file should start with something like:

```
\documentstyle[...,noweb,algoritmos]{...}
\input{keywords}
```

The main loop generates a random number in each iteration. If it is different from all previously generated numbers, we add it to the set where we keep different values, whereas it is rejected if we had already generated it at some iteration. The loop ends when the set contains n different values.

```
⟨ generate n distinct random numbers⟩≡
  while size($C$) $\neq n$ do
      ⟨ generate a random number $x$⟩
      ⟨ if $x$ is not in $C$ put it into $C$⟩
  end
```

Figure 3: Output of the example of Figure 2.

...

and the command to use is:

```
% noweave -delay [other options] files > tmp
```

The resulting file `tmp` can be supplied to LaTeX (or TeX) as usual, and the code gets printed in the standard way, i.e. in `\tt` font.

But you can give `tmp` as the input to `prettyprint` and get the code prettyprinted. Of course, you can avoid the temporary file and connect both programs through a pipe:

```
% noweave -delay [other options] files | prettyprint > doc.tex
```

Figure 3 shows the result of applying `noweave` and `prettyprint` to our example in Figure 2.

You can use all features of `algoritmos.sty` in your source files without any trouble, and intermix the facilities provided by the two systems: for example, you can use `noweb`'s quoting mechanism (`[[...]]`) or `prettyprint`'s quoting bars `|...|`; you can put a chunk `<<...>>` inside an `algorithm` environment, mathematical formulas can appear inside the code, and so forth. Also, the cross-referencing parameter of the `pcode` environment allows using `noweave`'s -x option to get chunks/module cross-referencing.

The last paragraphs are intended for those readers that have some knowledge of `noweb` internals and/or would like to introduce changes to the prettyprinter or in `algoritmos.sty`.

The prettyprint script has some built-in rules to process the output of noweave: it changes every begincode and endcode by a corresponding pair begin{pcode}-end{pcode}, and does not apply any modification to the text appearing inside chunk name delimiters (\LA...\RA, moddef..endmoddef), just acting as if they were comments.

Moreover, some macros of the noweb.sty file are redefined in algoritmos.sty, because they won't work otherwise inside a pcode environment. For instance, the macro \LA has to work in math mode if it is inside the pcode environment, whereas it has to work in a verbatim-like mode if it is inside noweb's standard \begincode...\endcode, and therefore, I had to slightly change its definition.

I shall remark that prettyprint would be much easier than it actually is, should it always be used in combination with noweb. If that were the case, prettyprint could be written as a preprocessor and called using noweave's -filter option. The prettyprint filter would then receive as input a sequence of lines produced by markup and both the number and the complexity of pattern-matching rules in prettyprint would be reduced.

Acknowledgements

I thank Norman Ramsey for encouranging me to put prettyprint and the other tools in the public domain, and Borja Valles for his useful suggestions from a user's point of view.

References

1. D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

2. L. Lamport. LaTeX*: A Document Preparation System.* Addison-Wesley, Reading, MA, 1986.

3. N. Ramsey. Literate-programming can be simple and extensible, 1993. Submitted to IEEE Software. Available as part of the documentation of the noweb package (release 2.5).

• 1400 191299
copia 1

Representació realista d'entorns amb medis participatius. Estat de l'art

Frederic Pérez
Xavier Pueyo

Report LSI–94–8–T