pa al sc: formatting Pascal using TEX

Pedro Palao Gostanza and Manuel Núñez García

Departamento de Informática y Automática
Universidad Complutense de Madrid
gostanza@eucmax.sim.ucm.es, manuelnu@eucmvx.sim.ucm.es

Abstract

This paper is based on our ideas about how a system which formats programs written in a structured language must work. Particularly, tools which help in typesetting texts where algorithms are described. Most of our ideas have been put in practice in the pa al sc system, which automatize the elegant layout of Pascal programs. This system is programmed as a TEX macro package.

1 Introduction

Almost every programming language have a structured syntax, and usually, there are several standard ways for the layout of programs in these languages. Both facets accomplish the same goal: programs must be easily understood. But, while the first one is used in order to facilitate the task to the compiler, the second one is used exclusively[1] in order to facilitate the comprehension of programs to readers.

During the history of programming languages, two variants in the representation of programs have been developed. First, programmers in a given language usually organize their programs in a similar way. Then, it is easier to read programs written by other people, and this fact gives rise to the development of particular modes for text editors like Emacs, which partially formats programs while writing then. On the other hand, a typesetting tradition has been developed for presenting programs in books or journals, which usually must have a very important aesthetic component.

There have been programs which partially solve these problems. For example, most programming languages environments have a pretty-printer program. Some systems, like `WEB`, go one step beyond extending the programming language so that it is possible to

1. This is not true in some languages, like Occam or Miranda, in which written representation prevails over syntax.

mix texts and codes in the same program. Then, the compiler sorts codes and texts out, obtaining a correct program in the given language, and a file containing the documentation ready to be processed by TEX. In this documentation, codes appears very well formatted.

Our experience mixing TEX and programs (writing class exercises, our own papers, or reading papers written by other people) says that, most of the times, a *verbatim mode* is used. As a matter of fact, if the program is not very large, it is customary to format it *by hand* using different type styles. We disagree with both solutions. The main advantage of using a verbatim mode is the simplicity and clearness of the source code, and the similitude between the result and what it is given to the compiler. Nevertheless, visual quality of this result is very poor. On the other side, formatting programs by hand is very hard, error prone, and it is rather difficult to understand the final code. We think that so many programs are presented in these two forms because there do not exist tools which format programs as they usually appear in papers or in lecture notes. These programs have the following characteristics:

- Programs are split in several fragments, usually unordered.
- Each of these fragments is not necessarily complete.
- In these programs, notation which does not belong to the programming language appears (either mathematical one or natural language).
- Programs are very related to the text around them, and thus, it is not so convenient to *input* them, but it is preferably to *paste* programs in the text.

We planned to develop a tool which helps us to write Pascal programs with these characteristics. The last item gives rise to two different alternatives: a preprocessor, or a TEX macro package. The first option has been widely used in the TEX world as well as in the pretty-printers world. It is enough to write a program which leaves the text without any change, adding the adequate TEX indications for typesetting programs. The latter task is more difficult than usual, because of the first three previous items, but it is not substantially new.

While an adequate tool dealing with these items helps in the preparation of texts where programs are a fundamental part, it still has the same important problem of the usual solutions (either formatting by hand or using verbatim mode): a complete and explicit indication of all format aspects instead of *declaration*. This is a similar problem to that which appears when writing big documents using TEX without macros that organize the final result from a logical point of view. For example, a change in the indentation of a language component (e.g. the indentation of a **while** clause) would oblige to correct previous programs trying to fix it. From the reader's point of view, there exist another big problem: a well organized text can be easily understood, even if the final layout is not standard, but a reader would hardly understand a program formatted slightly away from his own style. This shows the importance of a declarative format, where final presentation of programs depends on some *mayor modes* and on a small set of explicit

```
\Program \Hello( \input, \output );
\Begin \WriteLn( 'Hello word!' ) \End.
```

---

program Hello (*input*, *output*);
begin
    WriteLn('Hello␣word!')
end. {*End of program*}

Figure 1: The first program scheme

indications. If one wants to read a text containing programs, it is enough to choose his favorite mode and to process it again.

In this paper we present a system which automatically formats Pascal programs, fulfilling the previous requirements. This system is called pa/sc[2] and it is entirely programmed in TEX. Developing pa/sc, we have experimented most of our previous ideas. In fact, we built prototypes for Pascal and for Modula-2. At last, we decide to implement a complete version for Pascal because it always stated more difficult problems, and there exist several format styles for Pascal which are quite different.

pa/sc is fully declarative. It has three explicit format *hints*, which change the default option of the system. Although we only have implemented one format mayor mode, we will show how other modes can be simulated using these explicit hints. This shows us that these format hints are enough expressive and that more modes may be easily added to the system, just by simulating them internally.

2 Basic usage

TEX recognizes the beginning of a piece of program by the control word `\beginPascal`, while the program must be finished with the control word `\endPascal`. In LATEX version, there exist an environment called pascal. We will call the piece of program that appears between these two control words a *program scheme*, because it does not need to be neither a complete program nor an acceptable program by a Pascal compiler.

In the following, we will show program schemes together with its the final layout (as they are formatted by pa/sc) using the following convention: program schemes in `\tt` font and bellow, separated by a rule, the result of processing this program scheme with pa/sc. See figure 1 for an example. Some important characteristics of pa/sc appear in the simple program of this figure. A correct Pascal program is almost a correct program

2. The name is chosen in order to remark the formatting aspect of the system parameterized by the used programming language.

scheme, but it is necessary to add the character \ before each symbol (reserved words or identifiers) converting it in a control sequence. In order to understand why this addition is mandatory, it is enough to know how pa|sc internally works. There is no parser which formats a program scheme, but each of the elements of the program performs certain local actions, contributing to the final result. Reserved words usually carry out decisions, like changing indentation or breaking a line, while identifiers usually just write themselves with the adequate font. For this reason they need to be control sequences, in order to associate them a TEX macro.

Another significant detail of pa|sc is that reserved words are capitalized. In order to avoid possible interferences with TEX internal macros (e.g. `\if` or `\else`). This is not a constraint because, as we show bellow, pa|sc has an option that chooses the final result (upper case, lower case, etc). But the problem still remains for identifiers. In the previous program, we had a variable called *output*, recognized inside the program by the control sequence `\output`. When pa|sc finds this sequence, it is redefined as a macro that expands to "*output*." But `\output` is a fundamental register in the pages generation mechanism of TEX. If it is activated when this variable is redefined, a very strange error is produced. In order to solve this problem, we allow identifiers to have a format such that even a user who is not a TEXnician will be sure that there are not interferences with TEX. We decide to provide an special character to begin identifier names. This special character cannot appear neither in correct Pascal identifier names nor in TEX control words. Due to the first characteristic, we can detect and delete it from the final result, while due to the second one we can be sure that there are no interferences with macros. The special character is `!`[3] and it must be used exclusively as the first letter of identifiers, because this is the only place where it is deleted. Then, in the previous program we would write `\!input` and `\!output` instead of `\input` and `\output`. Note that `!` is not needed in `\Hello`, because pa|sc does not define it as a macro. Also, it cannot be used in `\WriteLn`, because this is an identifier introduced automatically with this capitalization by pa|sc.

Let us remark that while Pascal is case insensitive, TEX is case sensitive, and thus some coherence must be kept when writing identifiers along a program.

3 Piecemeal programs and options

Let us remember that a program scheme is a *self-contained* piece of a program in the following sense: it can format itself. For example, a simple sentence

```
\WriteLn( 'End of file' )
```

WriteLn('End␣of␣file')

3. The character `@` may seem more suitable because it is used when defining private macros, but precisely for this reason it is not guaranteed absence of interferences.

a declarations sequence

```
\Var \!x: \Integer;
\Const \!c = 100;
```

---

var x: Integer;
const $c = 100$;

or structured sentences,

```
\Var \!c: \Char;
\Repeat
  \WriteLn( 'Do you want to continue? ' );
  \ReadLn( \!c );
\Until (\!c = 'y') \lor (\!c = 'n');
```

---

var c: Char;
repeat
    WriteLn('Do␣you␣want␣to␣continue?␣');
    ReadLn(c);
until $(c = \texttt{'y'}) \vee (c = \texttt{'n'})$;

where all the identifiers are explicitly declared. In the previous examples, we have seen that all of the identifiers are explicitly declared inside the program scheme. This is because pa sc al encloses a program scheme in a group, so that all the declarations appearing in this scheme remain until the end of this group.

pa sc al provides two methods for writing programs which depend on identifiers that we do not want to introduce explicitly. Both methods are part of the *options* mechanism. Options, enclosed by brackets ([ ]), can appear prefixing a program scheme. Particularly, there exists a family of options to declare identifiers which are used in the subsequent program scheme. In the program

```
[\var\!power\!x\!y;]
\!power := 1;
\While \!y \not= 0 \Do
  \Begin
    \!power := \!power * \!x; \!y := \!y - 1
  \End
```

$power := 1$;
while $y \neq 0$ **do begin**
    $power := power * x$;
    $y := y - 1$
end

we have *declared* the variables `\!power`, `\!x` and `\!y`. The list of identifiers declaration options is: `\var`, `\type`, `\const`, `\proc`, `\func`, `\pseudoVar` and `\field`. This options are used as `\var` in the example: prefixing some control sequences without separation between them, finishing with a semicolon. The first five options correspond to the usual Pascal identifiers. Option `\pseudoVar` introduces a identifier name representing a function name, and it is necessary for representing an isolated function body. Option `\field` introduces record field names.

These options are only useful if the different pieces of code are not related among them. But usually, the same identifier is used in different pieces, and in a grouped form (e.g. a data structure with types and operations). pasc introduces the concept of *declarations set*. A declarations set is an object that records the declarations of a program scheme, allowing that these declarations may be used in another program scheme. For example, let us suppose that one wants to write a function which calculates the number of nodes of a binary tree. First, the type must be introduced

```
[\newDecls{treedec}\memoDecls{treedec}\type\!Element\!TreeNode;]
\Type \!Tree = ^\!TreeNode;
      \!TreeNode = \Record
                     \!elem: \!Element;
                     \!left, \!right: \!Tree
                   \End;
```

---

type Tree = ↑TreeNode;
TreeNode = **record**
elem: Element;
left, *right*: Tree
end;

The first option, `\newDecls`, creates a new declarations set called `treedec`. The second one indicates that we want to record in `treedec` all the declarations appearing from this point until the end of the program scheme. Particularly, `treedec` records declarations given by the third option which introduces types "Element" and "TreeNode" which is used before it is declared.[4] Briefly, the second line of options indicates that reserved words are presented with capital letters and that lines are numbered starting with 1. Following with our example, the function "nodes" is

```
[\useDecls{treedec}\memoDecls{treedec}]
\Function \!nodes( \!t: \!Tree ): \Integer;
```

---

function nodes(*t*: Tree): Integer;

4. pasc does not deal with recursion in pointer types, but it is not very complicated to fix it.

\useDecls allows to use in this program scheme the identifiers recorded in treedec. The second option adds to treedec the declarations appearing in this program scheme. Then, the function body is

```
[\useDecls{treedec}\var\!t;\pseudoVar\!nodes;]
\If \!t = \Nil \Then \!nodes := 0\>
\Else \!nodes := \!nodes( \!t^.\!left ) + \!nodes( \!t^.\!right );
```

if $t = Nil$ **then** $\mathrm{nodes} := 0$
else $\mathrm{nodes} := \mathrm{nodes}(t\uparrow.left) + \mathrm{nodes}(t\uparrow.right)$;

which presents an example where the option \pseudoVar is necessary. Finally, an example using the function "nodes" is

```
[\useDecls{treedec}\var\!t;
 \noMarkStringSpaces]
\WriteLn( 'Number of nodes in tree :', \!nodes(\!t) );
```

WriteLn('Number of nodes in tree :', nodes(t));

As it is shown in this example, it is usual to use \newDecls or \useDecls preceding \memoDecls. The option \decls produces one of these two sequences depending if the declarations set already exists. In fact, \decls{name} is equivalent to \newDecls{name}\memoDecls{name} if name has not yet been declared, and otherwise it is equivalent to \useDecls{name}\memoDecls{name}.

Using declarations set, it is very easy to change the capitalization of the predefined identifiers:

```
\beginPascal[
  \decls{predefined}
  \const\!nil\!true\!false;
  \type\!integer\!boolean\!real\!char\!text;
  \proc\!write\!writeln\!read\!readln\!new\!dispose;
  \func\!succ\!pred\!sqr\!sqrt;
]\endPascal
```

4 Layout hints

Previous examples show the pa/sc default formatting mode. Possibly, this is not a very standard style and, as we suggest in the introduction, it is difficult to understand programs when the reader is not used to this style. But this is not a problem for the philosophy behind pa/sc: the reader can choose another mode and recompile the file. Unfortunately, by now, this is the only implemented mode in pa/sc.

In this section we present the *layout hints*. These elements allow to locally change defaults options but they must be used exclusively in those places where the understanding of the code would improve if it is not presented in the default mode. Anyway, their use must be limited because it is against the declarative form behind pasc.

pasc only has three layout hints. Each of them specifies a kind of operation which is normally used to write Pascal programs: to break a line, to join two lines, and to align to a point. Respectively, they are activated by the three control symbols \>, \< and \!. For example:

```
[\decls{listdec}\type\!Element\!Node;]
\Type\! \!List = ^\!Node;
        \!Node =\! \Record\<
                      \!value: \!Element;
                      \!next: \!List;
                   \End;
```

```
type List = ↑Node;
     Node = record   value: Element;
                     next: List;
            end;
```

Layout hints work in a coherent form: if a line is broken, then the rest of the text is indented using a value (that depends on the context); if two lines are joined, a small separation is inserted; an alignment only remains in the corresponding context. In short, the layout hints *know* the mechanism of the structured construction of Pascal programs, and thus they are much more *abstract* than formatting by hand.

Using layout hints in a systematic way, other format styles can be obtained. For instance, the previous example presents a very frequent style where type declarations are aligned. Another style appears when these declarations are split:

```
[\useDecls{listdec}]
\Type\> \!List = ^\!Node;
        \!Node =\> \Record\<
                      \!value: \!Element;
                      \!next: \!List;
                   \End;
```

```
type
    List = ↑Node;
    Node =
        record   value: Element;
                 next: List;
        end;
```

This shows the expressiveness of the chosen layout hints and why it is so easy to add new format styles to pasc.

5 Other options

In addition to the definition of identifiers, the pasc options system allows to indicate many aspects of the final result. Below, we summarize some of the most significative options

`\lineNumbers,\noLineNumbers` This option allows (or does not) the numeration of lines.

`\firstLine` The count of lines is made globally. This option has an argument which changes the default value (i.e. 1).

`\cap,\Cap,\CAP` Alternative options indicating the capitalization of reserved words.

`\autoEnd,\noAutoEnd` This option indicate that a message corresponding to the end of functions, procedures or programs appears (or does not).

`\markStringSpaces,\noMarkStringSpaces` These options indicate that blank spaces must be substituted by ␣ or just a space is left.

`\abstractAssign/\textualAssign` Complementary options indicating if assignation is represented either by ":=" or by "←."

An example showing these features follows:

```
[\useDecls{listdec}
 \global\abstractAssign
 \noAutoEnd\Cap
 \lineNumbers\firstLine{1} ]
\Function \!exists( \!e:\!Element; \!l:\!List ): \Boolean;
  \Var\! \!find: \Boolean; \!aux: \!List;
\Begin
  \!find := \False;\< \!aux := \!l;
  \While \Not\!find \And (\!aux \not=\Nil) \Do\> \Begin
    \!find := \!aux^.\!value = \!e;
    \!aux := \!aux^.\!next
  \End;
  \!exists := \!find
\End;
```

```
1 Function exists( e: Element; l: List ): Boolean;
2     Var find: Boolean;
3         aux: List;
4 Begin
5     find ← False;   aux ← l;
6     While Not find And (aux ≠ Nil) Do
```

```
7           Begin
8               find ← aux↑.value = e;
9               aux ← aux↑.next
10          End;
11      exists ← find
12  End;
```

If these options are used together with `\global`, their effects remain in subsequent program schemes. For instance, in the previous example we have declared `\global\abstractAssign`, and thus, all the assignations appearing in the rest of the paper will be denoted by ←.

6 The fields problem

Nowadays, almost every Pascal compiler allows several record declaration sharing field names. They also allow that field identifiers can be used to denote other objects. pasc also allows this. For example, we can define a function computing the left subtree of a tree:

```
[\decls{treedec}]
\Function \!left( \!t: \!Tree ): \!Tree;
\Begin
  \!left := \!t^.\!left;
\End;
```

```
function left( t: Tree ): Tree;
begin
    left ← t↑.left;
end; {End of left function}
```

In the left hand side of the assignation, "left" denotes a function, while in the right hand side, "*left*" denotes the field of the record implementing the type "Tree."

pasc can distinguish from the context among the different uses of an identifier which is simultaneously used as a record field and as another object. Nevertheless, pasc is more limited than a Pascal compiler, because it does not keep type informations. This problem appears when using the **with** sentence:

```
[\useDecls{treedec}\useDecls{listdec}\var\!l\!t;]
\With \!l \Do
  \!value := \!value + \!nodes(\!left(\!t));
```

```
with l do   value ← value + nodes(left(t));
```

In the previous example, pa/sc has misunderstood the call to the function "left" for a use of the field *"left."* In general, whenever pa/sc analyzes a **with** command, it does not use the type information of the expression associated with the **with**. All the identifiers associated with field declarations will expand as a field, without taking care of the possible association with another object. In order to indicate pa/sc that some symbol does not represent a field, one must prefix it with `\)`, which indicates a *local closure* of a **with**. For example, the previous program would be

```
[\useDecls{treedec}\useDecls{listdec}\var\!l\!t;]
\With \!l \Do
  \!value := \!value + \!nodes(\)\!left(\!t));
```

---

with *l* **do** $value \leftarrow value + \mathrm{nodes}(\mathrm{left}(t))$;

In addition to `\)`, the prefix `\(` *locally opens* a **with**. For example, the previous program, assuming an external **with**, would be

```
[\useDecls{treedec}\useDecls{listdec}\var\!l\!t;]
\(\!value := \(\!value + \!nodes(\!left(\!t));
```

$value \leftarrow value + \mathrm{nodes}(\mathrm{left}(t))$;

7 Differences with respect to Pascal and TEX

There exist two aspects in Pascal syntax which pa/sc implements in a slightly different way: comments and subrange types.

Comments are written using the control sequence `\Comment`, which has two arguments: the first one is an optional dimension (enclosed between squared brackets) and the second is the text.

If the first one is omitted, the text is written using a horizontal box. Otherwise, the optional parameter indicates the horizontal size of a vertical box. For example

```
\Const \!max = \cdots; \Comment{Maximum size of the stack}
\Type \!stack = \!\Record
                  \!top: [0..\!max];
                  \!data: \Array [1..\!max] \Of \Integer
                \End;\< \Comment[7cm]{Invariant: the
                          {\it top} field contains the
                          index of the last pushed element.}
```

---

const $max = \cdots$; {*Maximum size of the stack*}
type stack = **record**

```
        top: 0 .. max;
        data: array [1 .. max] of Integer
    end;      {Invariant: the top field contains the index
               of the last pushed element.                }
```

In this example, we show another difference with respect to Pascal: subrange types must be enclosed between square brackets, as done in Modula. With this, we allow arbitrary expressions appearing in both range limits:

```
[\const\!a\!b\!c\!d;]
\Type\! \!range = [(\!a+\!b)*(\!c+\!d)..100];
      \!colors = (\!red,\!green,\!blue);
```

---

type range $=$ $(a + b) * (c + d) \,..\, 100$;
colors $=$ (*red*, *green*, *blue*);

Without using "[]", an expression beginning with "(" will be taken as an enumeration.

As we have shown along the paper, any *math mode* control sequence can be used inside Pascal expressions. Nevertheless, ^ and ' have been redefined in order to respectively represent an indirection and the beginning of a string. Superscripts appears in Pascal expressions using `\^`. If one wants to write *primes*, the whole definition must be used, that is `\^\prime`.

8 Conclusions and future work

In this paper we have presented the system pa/sc, which has been developed for helping in the typesetting of texts where Pascal programs appear. pa/sc is very operative, and excepting the absence of several modes, fulfills all of our proposed objectives. We have used it to write exercises and it has proved to be very useful when presenting either bottom-up decompositions or top-down ones.

We think that the most important future task is to implement some major modes including the most usual ones. This is relatively easy because of the organization of pa/sc: most of the code is independent of the format style and the three major Pascal syntactic groups (types, sentences and declarations) are distributed in different files. Then, modes can be chosen by syntactic groups.

Another task is to translate the ideas behind pa/sc to other programming languages such as Modula-2, Ada or ML. Most of the code related with formatting and the declarations sets is independent of the language, and thus it can be shared.