Pretprin — a $\LaTeX\,2_\varepsilon$ package for pretty-printing texts in formal languages

Marcin Woliński

wolinski@melkor.mimuw.edu.pl

Abstract

A $\LaTeX\,2_\varepsilon$ package is presented which provides tools for building lexical and syntactical analyzers in TeX that can be used for pretty-printing. Examples of such analyzers for the programming languages Pascal and Prolog are shown as well as a small example of a new analyzer definition.

The problem

In most books on computer science published today, algorithms are presented in a way known to every TeXnician as *verbatim*. In some well printed books however programs in algorithmic language Pascal are formatted in a rather complicated way, with bold keywords, italic identifiers and appropriate indentation on every line:

```
begin q := 0;  r := x;
while r ≥ y do
  begin r := r − y;  q := q + 1
  end
end
```

Such a layout gives a graphical representation for logical structure of the program. Indentation shows control flow: one can easily see "which **end** matches which **begin**". Keywords are put in boldface while other identifiers are in italic, so even a beginner is able to recognize which elements are predefined parts of the language. Every occurrence of a given structure is formatted the same way, making it easier to "navigate" in the text.

This paper is dealing with the question how to generate such a layout with TeX.

A large amount of tedious work would be needed to add all typesetting commands by hand. Moreover it would be difficult to get a consistent result in, e.g., a 200-page book.

Fortunately someone already had this problem before. The person was Prof. Knuth, who wanted to publish a few large programs written in Pascal including TeX and METAFONT. Since he wanted to get good quality printouts in a finite length of time, he decided to teach the computer how to typeset Pascal. This way of generating pretty-printed text of the program became one of the WEB system functions.

The WEB system (or rather, WEAVE, which deals with the processing of the program's documentation) performs syntactical analysis to recognize language constructs such as **if — then — else**, **repeat — until**, etc. This provides a layout consistency that is hard to achieve with other techniques.

WEB is a really smart tool for generating technical documentation for programs. But using WEAVE to process a book on computer science, containing some random pieces of code that are not supposed to build a working program, is somewhat unnatural. In such a case it would be preferable to avoid using external tools. But that would mean teaching TeX itself how to pretty-print Pascal.

That is precisely what Pretprin was meant to do. The source for the example above in Pretprin's notation is:

```
\begin{Pascal*}
  begin q:=0;r:=x;while r>=
  y do begin r:=r-y;q:=q+1end end
\end{Pascal*}
```

One more insight was crucial for Pretprin's architecture: in WEAVE (being a Pascal program), the rules of parsing Pascal texts are expressed with a series of complicated **if — then — else** constructs. In TeX, on the other hand, it's much easier to write a general interpreter for such rules. But this means language-specific rules are separated from the rest of the program and it is relatively easy to substitute them with other sets of rules.

So in its present shape Pretprin is mainly a toolbox for building scanners and parsers in TeX. The tools can be used to analyze any sufficiently regular data, which probably includes any commonly used formal language.

Examples of Pretprin usage

Pretprin is loaded with `\usepackage` commands stating which language-specific modules should also be loaded. For example, to typeset a document containing pieces of Pascal and Prolog use:

```
\usepackage[pascal,prolog]{pretprin}
```

The Pascal module provides a LaTeX environment named `Pascal` that is used to typeset displayed pieces of Pascal programs:

```
\begin{Pascal}
  var i,L,w:integer; ch:char;Z:
  array [1..wmax]of char;
  begin L:=0;repeat w:=0;read(ch);
  while (ch<>' ') and (ch<>eol)
  do  ...
  until eof (input)
  end.
\end{Pascal}
```

Note in the output shown below that spacing, indentation and line-breaking was done by Pretprin, completely ignoring the layout of the input (which, by the way, is definitely bad). Moreover, relational symbols such as =, < or ≠ (which is substituted for `<>`), and binary operators like +, .., and ∧ (which replaces **and** in the source), are put into TeX's math mode with correct spacing. The overall layout is very similar to that generated by WEAVE.

```
var i, L, w: integer; ch: char;                    (1)
      Z: array [1 .. wmax] of char;
  begin L := 0;
  repeat w := 0; read(ch);
    while (ch ≠ '␣') ∧ (ch ≠ eol) do
      begin w := w + 1; Z[w] := ch;
      read(ch)
      end;
    if w > 0 then
      begin if L + w < Lmax then
        begin write('␣'); L := L + 1
        end
      else begin write(eol); L := 0;
        end;
      for i := 1 to w do write(z[i]);
      L := L + w
      end
  until eof(input)
  end.
```

Inline pieces of code such as "Consider A being an **array** $[1\,..\,N]$ **of** *integer*…" can be written as

```
Consider \pascal{A} being an
\pascal{array [1..N] of integer}...
```

The Pretprin module for pretty-printing Prolog (in modern syntax) defines an environment `Prolog` and a command `\prolog` with one parameter. Here is an example of four Prolog clauses:

```
\begin{Prolog}
d(X,X,D):-atomic(X),!,D=1.
d(C,X,D):-atomic(C),!,D=0.
d(U+V,X,DU+DV):-d(U,X,DU),d(V ,X,DV).
d(U*V,X,DU*V+U*DV):-d(U,X,DU),d(V,X,DV).
\end{Prolog}
```

The output is simpler than in the Pascal case, since Prolog's syntax is more terse. However, typographical symbols are substituted for `:-` and reasonable spacing is added.

```
d(X, X, D) ← atomic(X), !, D = 1.
d(C, X, D) ← atomic(C), !, D = 0.
d(U + V, X, DU + DV) ←
      d(U, X, DU), d(V, X, DV).
d(U ∗ V, X, DU ∗ V + U ∗ DV) ←
      d(U, X, DU), d(V, X, DV).
```

Our last example shows a few rules of Stanisław Szpakowicz's formal grammar of a large subset of Polish. The notation used is a variation on the DCG (Definite Clause Grammar) theme.

Nonterminals in the grammar are put in bold, but conditions (marked with minus sign in front) are in normal weight. In arguments, variables are set in italic and constants in upright shape. The pretty-printer prefers to break lines between a nonterminal and a condition rather than between two conditions or two nonterminals, so conditions tend to group on separate lines. Pretprin also carefully takes into account the space needed by rule numbers on the right side of the column.

```
ZDANIETO
 = ZDANIEOGR(NR, R, L, O, war, prze, NEG)     (zt1)
 = SPOJNIK(TO)                                (zt2)
   ZDANIEOGR(nr, r, l, o, WAR, PRZE, neg) .
SZDRZ(p, r, MNO, o)
 = SPOJLEWY(nr) −ALT(nr, 1.2.3)             (szdrz1)
   FRZ(p, r1, l1, o1) PRZEC
   SPOJPRAWY(nr) FRZ(p, r2, l2, o2)
   −UZGR(r1, r2, r) −MIN(o1, o2, o)
 = FRZ(p, r1, l1, o1) PRZEC KSPOJ(wz)       (szdrz2)
   −ALT(wz, (A.TAKŻE).(JAK.RÓWNIEŻ)
     .(JAK.TEŻ)) FRZ(p, r2, l2, o2)
   −UZGR(r1, r2, r) −MIN(o1, o2, o) .
```

One more thing worth emphasizing here is that it is possible to use multiple Pretprin modules in a single document. For example, the current paper contains examples in three different programming languages and it was generated with a single LaTeX run.

How to build a pretty-printer

In this section we will try to build a pretty-printer for a very simple language of terms. Terms are abstract expressions which logicians and computer scientists just love to write. We will consider a basic notation for terms which allows atoms (names built from letters) and functors (having a name and, in parentheses, a list of arguments, each being a term). Here is a typical example of a term:

```
loves(John, and(Mary, and(Tom,Jerry)))
```

(No semantic interpretation please, terms are merely abstract structures.)

Lexical analysis (scanning). The first task in analyzing a string is to detect "words" in it. For the string

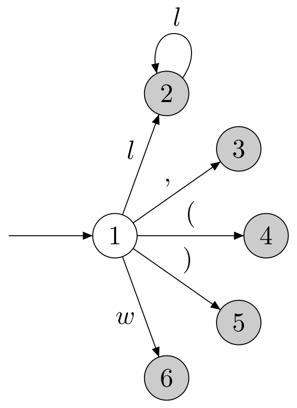
```
foo(␣bar,␣baz)
```

the "words" are

$\boxed{\texttt{foo}}\ \boxed{\texttt{(}}\ \boxed{\texttt{bar}}\ \boxed{\texttt{,}}\ \boxed{\texttt{baz}}\ \boxed{\texttt{)}}$

(Note that space characters were ignored.) We can distinguish here four "parts of speech": sequences of Latin letters (atoms), opening parentheses, closing parentheses, and commas.

For splitting strings of characters into words we use an apparatus known as a finite state automaton. The diagram below represents an automaton for the example language:



In the example language the function of any letter is the same: letters are building-blocks for atoms. For that reason the notion of character groups is used. The following declarations define in TeX parlance character groups l and w:

```
\DeclareGroup{l}
  {abcdefghijklmnopqrstuvwxyz}
\DeclareGroup{w}{ ^^I^^M}
\CompileGroups
```

Therefore the label l in the diagram denotes any letter, while the label w denotes any "white character".

Let us now trace what happens when the string

```
foo(␣bar,␣baz)
```

is being run through our automaton. The automaton starts in state 1 and encounters the letter `f` from the input string. Since this character belongs to the "letters" group, the automaton performs a transition to state 2 (remembering the `f`). Now, the letter `o` is input, and since there is a transition from 2 to 2, reading a letter, the automaton does just that. Then another `o` comes that is handled similarly. And now, still in state 2, the automaton sees a `(`. Since there is no arrow from 2 labeled with `(`, the automaton cannot consume it and stops. State 2 is marked with a gray circle meaning that it is an accepting state; stopping in this state means a word has just been read. In our case, the word is `foo`. The action associated with state 2 will pass this word to the next processing stages.

Now a search is started for another word, so the automaton returns to state 1. A transition labeled with `(` leads to state 4. The next character is a space and there is no transition from 4 labeled with a space. The automaton stops, and a single parenthesis is recognized as the second word.

The next "word" consists of a single space. It is accepted in state 6, which is somewhat special in that the action associated with it is empty. This way spaces get gobbled.

The process continues until the whole string is processed.

Before we actually describe this automaton in TeX we'll take a closer look at the actions associated with states. These actions prepare a list of "scraps" on which the syntax analysis will work. This list is constructed with `\AppendElem` procedure. Every scrap has a grammatical category and translation (the actual text). These two constitute arguments of `\AppendElem`. In our case, categories are just the parts of speech mentioned earlier: atom, open, close and comma.

The first state is not accepting, so there is no action associated with it. The state has five transitions:

```
\DeclareTransition 1-l->2.
\DeclareTransition 1-,->3.
\DeclareTransition 1-(->4.
\DeclareTransition 1-)->5.
\DeclareTransition 1-w->6.
```

State 2 on the other hand is accepting, words of the category atom are recognized in it:

```
\DeclareState{2}{\AppendElem{atom}{#1}}
\DeclareTransition 2-l->2.
```

(`#1` above is the string read as the automaton was going from the start state.) The rest of the states have no leaving transitions but are accepting. Commas and parentheses are recognized in them, and in state 6, blank characters are gobbled.

```
\DeclareState{3}{\AppendElem{comma}{#1}}
\DeclareState{4}{\AppendElem{open}{#1}}
\DeclareState{5}{\AppendElem{close}{#1}}
\DeclareState{6}{}
```

And the last declaration specifies the starting state:

```
\CompileScanner{1}
```

Syntax analysis (parsing). Now the input string has been read and converted to the form

atom	open	atom	comma	atom	close
`foo`	`(`	`bar`	`,`	`baz`	`)`

The next task is to recognize the syntactical structure of the text. We already know the parts of speech, but now higher level grammatical categories can emerge. This process is described with a set of simple grammatical rules.

Our first observation will be that when an atom is immediately followed by an open parenthesis it is the beginning of a term. We'll call such an entity termhd (a term head):

$$\textsf{atom open} \rightarrow \textsf{termhd}$$

In our example this rule allows us to derive that `foo(` is the beginning of a term.

If there is no parenthesis after an atom it surely is a term all by itself (this is the case with `bar` and `baz` in the example).

$$\textsf{atom} \rightarrow \textsf{term}$$

Terms can have arguments, so the next rule describes how a termhd can grow: adding a term and a comma to a termhd gives another well formed termhd.

$$\textsf{termhd term comma} \rightarrow \textsf{termhd}$$

And finally when after a termhd comes a term (the last argument) and a closing parenthesis the whole fabric can be stuffed into a new term:

$$\textsf{termhd term close} \rightarrow \textsf{term}$$

(Note that we do not accept `foo()` as a term.)

These rules again in the TeX notation are:

```
\DeclareProduction{atom,open}
  \ThisElem\TwoElems{termhd}
  {#2#1}\ThisElem
\DeclareProduction{atom}
  \ThisElem\OneElem{term}
  {\textit{#1}}\PrevElem
\DeclareProduction{termhd,term,comma}
  \ThisElem\ThreeElems{termhd}
  {#3\formatterm{#2}#1\ }\ThisElem
\DeclareProduction{termhd,term,close}
  \ThisElem\ThreeElems{term}
  {#3\formatterm{#2}#1}\PrevElem
```

This notation is somewhat more verbose and allows us to describe situations where not all elements of the left side of a rule are to be collapsed into a new scrap (context rules).

The first rule can be read as follows: if you are looking at the atom scrap followed by an open scrap, do as follows: starting from `ThisElement` (the open scrap), take `TwoElements` and replace them with a scrap of category termhd, the translation of which was formed from the translations of atom (`#2`) and open (`#1`) scraps. Then continue the process starting from `This` (the inserted) `Element`.

The pretty-printing depends on building appropriate translations of complex grammatical entities. We have ignored all spaces in the input, so now we are responsible for putting them back in a consistent manner. Moreover, in the example language when an atom is being recognized as a term without arguments, `\textit` is applied to render the atom's name in italics. And whenever a term is added to the list of arguments of a term-under-construction, the macro `\formatterm` is applied. In the definition of this macro we decide what it means to pretty-print a term. To keep things simple, we will just put a frame around each sub-term:

```
\newcommand\formatterm[1]{\fbox{#1}}
```

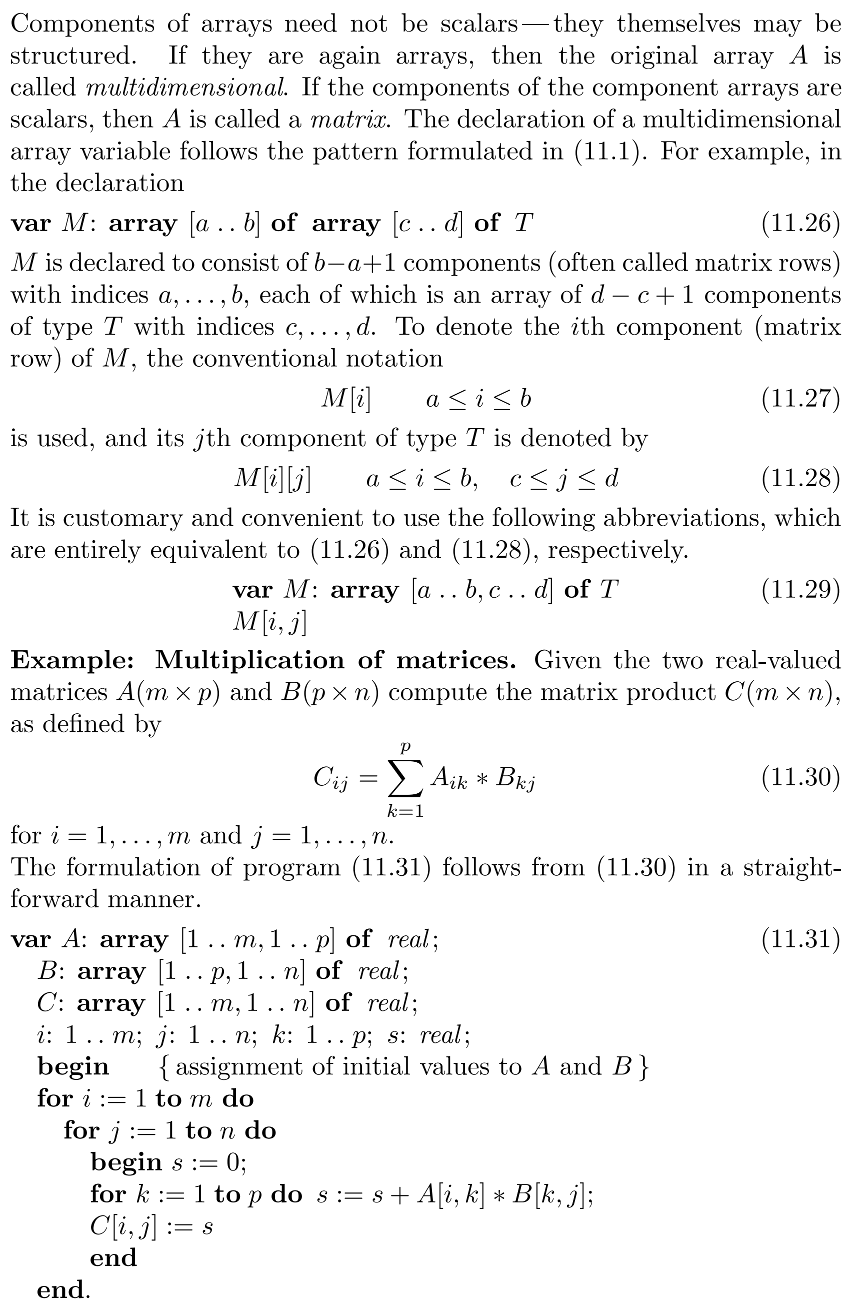
With these definitions

```
\begin{Terms}
loves(John, and(Mary, and(Tom,Jerry)))
\end{Terms}
```

yields

$$\text{loves}(\boxed{\textit{John}},\ \boxed{\text{and}(\boxed{\textit{Mary}},\ \boxed{\text{and}(\boxed{\textit{Tom}},\ \boxed{\textit{Jerry}})})})$$

A Appendix



Components of arrays need not be scalars — they themselves may be structured. If they are again arrays, then the original array A is called *multidimensional.* If the components of the component arrays are scalars, then A is called a *matrix.* The declaration of a multidimensional array variable follows the pattern formulated in (11.1). For example, in the declaration

$$\textbf{var } M\text{: } \textbf{array } [a\,..\,b] \textbf{ of } \textbf{array } [c\,..\,d] \textbf{ of } T \qquad (11.26)$$

M is declared to consist of $b-a+1$ components (often called matrix rows) with indices $a, \ldots, b$, each of which is an array of $d-c+1$ components of type T with indices $c, \ldots, d$. To denote the ith component (matrix row) of M, the conventional notation

$$M[i] \qquad a \le i \le b \qquad (11.27)$$

is used, and its jth component of type T is denoted by

$$M[i][j] \qquad a \le i \le b, \quad c \le j \le d \qquad (11.28)$$

It is customary and convenient to use the following abbreviations, which are entirely equivalent to (11.26) and (11.28), respectively.

$$\begin{array}{l} \textbf{var } M\text{: } \textbf{array } [a\,..\,b, c\,..\,d] \textbf{ of } T \\ M[i,j] \end{array} \qquad (11.29)$$

Example: Multiplication of matrices. Given the two real-valued matrices $A(m \times p)$ and $B(p \times n)$ compute the matrix product $C(m \times n)$, as defined by

$$C_{ij} = \sum_{k=1}^{p} A_{ik} * B_{kj} \qquad (11.30)$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.
The formulation of program (11.31) follows from (11.30) in a straightforward manner.

```
var A: array [1 .. m, 1 .. p] of real;                    (11.31)
  B: array [1 .. p, 1 .. n] of real;
  C: array [1 .. m, 1 .. n] of real;
  i: 1 .. m; j: 1 .. n; k: 1 .. p; s: real;
  begin     { assignment of initial values to A and B }
  for i := 1 to m do
    for j := 1 to n do
      begin s := 0;
      for k := 1 to p do  s := s + A[i, k] * B[k, j];
      C[i, j] := s
      end
  end.
```

Figure 1: A page from chapter 11 of Niklaus Wirth's *Systematic Programming: An Introduction*

```
\documentclass{book}
\usepackage{pascal}

\begin{document}
...

Components of arrays need not be scalars---they themselves may be structured.  If they are
again arrays, then the original array \pascal{A} is called \emph{multidimensional}.  If
the components of the component arrays are scalars, then \pascal{A} is called a
\emph{matrix}.  The declaration of a multidimensional array variable follows the pattern
formulated in (\ref{arrtype}).  For example, in the declaration
\begin{Pascal}
  var M: array[a..b]of array[c..d]of T
\end{Pascal}\pplabel{abcdarray}
\pascal{M} is declared to consist of $b-a+1$ components (often called matrix rows) with
indices $a,\ldots,b$, each of which is an array of $d-c+1$ components of type \pascal{T}
with indices $c,\ldots,d$.  To denote the $i$th component (matrix row) of \pascal{M}, the
conventional notation
\begin{equation}
  \pascal{M[i]}\qquad a\leq i \leq b
\end{equation}
is used, and its $j$th component of type \pascal{T} is denoted by
\begin{equation}\label{ijthelem}
  \pascal{M[i][j]}\qquad a \leq i \leq b, \quad c \leq j \leq d
\end{equation}

It is customary and convenient to use the following abbreviations, which are entirely
equivalent to (\ref{abcdarray}) and (\ref{ijthelem}), respectively.
\begin{equation}
  \begin{tabular}[t]{l}
    \pascal{var M: array [a..b,c..d] of T}\\
    \pascal{M[i,j]}
  \end{tabular}
\end{equation}

\subsubsection{Example: Multiplication of matrices.}
Given the two real-valued matrices $A(m\times p)$ and $B(p\times n)$ compute the matrix
product $C(m\times n)$, as defined by
\begin{equation}\label{mmultdef}
  C_{ij} = \sum_{k=1}^p A_{ik}*B_{kj}
\end{equation}
for $i=1,\ldots,m$ and $j=1,\ldots,n$.

The formulation of program (\ref{mmult}) follows from (\ref{mmultdef}) in a
straightforward manner.
\begin{Pascal}
var A:array[1..m,1..p]of real; B:array[1..p,1..n] of real; C:array[1..m,1..n] of real;
  i:1..m; j:1..n; k:1..p; s:real;
begin (*assignment of initial values to \pascal{A} and \pascal{B} *)
for i:=1to m do for j:=1 to n do
begin s:=0; for k:=1 to p do s:=s+A[i,k]*B[k,j]; C[i,j]:=s
end
end.
\end{Pascal}\pplabel{mmult}

\end{document}
```

Figure 2: Source code for the previous example