Integrating Programs and Documentation

Ralf Hartmut Güting

Praktische Informatik IV, FernUniversität Hagen
D-58084 Hagen, Germany
gueting@fernuni-hagen.de

Abstract: A simple tool called *PD system* is described and offered for general use that allows one to integrate programs and their documentation in ASCII files. Such a *program with documentation file* (*PD file*) consists of alternating *documentation* and *program sections*. PD files can be compiled directly; for a compiler, documentation sections are just commentaries to be ignored. On the other hand, the PD system allows one to transform PD files into LaTeX source files and so to produce formatted documents. For the description of formatted material in the documentation sections, a simple markup language is offered. The main goal in the design of this language is readability of the source text. In other words, formatting specifications are kept as "implicit" and "invisible" as possible and much of the "formatting noise" occurring in LaTeX and other markup languages can be hidden. PD files should be readable for anyone without prior learning of the markup language.

Independent from the problem of documenting programs, PD files may be convenient for describing text documents in ASCII files (rather than using LaTeX directly). The PD System was written in C using the *lex* and *yacc* tools. It is free software available from the author and includes its own documentation as a PD file.

May 1995

1 Introduction

We would like to integrate program documentation with the program text. A single file should contain the program together with its documentation. It should be possible and easy to write documentation together and interleaved with programming. If documentation is written separately, we have the following problems:

- There is a strong tendency that documentation is not written at all.
- People studying the program text might not see the documentation or know about it; if documentation and program are kept in separate files, then organizational means are necessary to ensure that everybody interested in the program knows where to find the documentation.
- Programs are dynamic; if documentation is treated separately, then it is very likely that changes to a program will not be reflected directly in the documentation.

On the other hand, the standard facilities for program documentation are very poor. It is just possible to write ASCII text within “commentary” brackets within a program file. For writing good documentation one would like to have a full-fledged formatting system available, so that it is possible to produce section headings, numbered lists, use formulas, graphics, etc.

The most comfortable way to write documentation is certainly to use the text system one is familiar with; for most people this means a WYSIWYG editor such as FrameMaker, MS Word, etc. On the other hand, a program file is an ASCII file. A third aspect is that the program file with its integrated documentation should be distributable so that “everybody” can on the one hand compile it and on the other hand display or print it as a formatted document.

One approach to solve this problem is to use an ASCII file that allows LaTeX within commentary sections. LaTeX is widely available, well-documented, and quite powerful in its formatting capabilities. Such a file is directly compilable; a simple converter can transform a program file into a LaTeX source file. The converter only ensures that program sections are taken “verbatim” by LaTeX which means they are just reproduced the way they have been formatted in the source file. Unfortunately, there are also some problems with this approach:

- To people used to WYSIWYG editors, LaTeX source files look ugly. They contain too many commands interleaved with the text which disturbs readability. It is not nice to work on this representation of the text. In the environment described here one often has to look at this representation because in programming one works with the ASCII file.
- The LaTeX source files are hardly readable for people unfamiliar with LaTeX. It would be good if the program files were readable even before formatting.
- The treatment of drawings is clumsy. One needs a separate drawing tool anyway.

A second approach is to use an editor (e.g. FrameMaker) and to write program and documentation as a document of this editor. This is very nice for producing the documentation but has the following problems:

- One must find a way to produce from the document a source file for compilation. (This is probably not too difficult.)
- In program debugging, one has to go through many cycles of getting error messages, changing the program, recompiling. The compiler refers to line numbers of its source file. If one edits the program within the document, it (i) may be difficult to find the relevant position

in the document for changes, and (ii) it might take too long to save the document, transform to source file for compilation, etc.

- Relying on a specific text editor is much less portable than the first approach.
- If the text editor and the compiler are on different machines, this may also lead to problems.

To alleviate the problems of readability associated with the first approach, we suggest a technique that we call *implicit formatting*.

The basic approach is the same as in LaTeX or other formatting languages: to produce an ASCII file that contains text as well as formatting specifications. The difference is that we strive for readability of this file by giving formatting specifications as much as possible in an implicit way rather than by explicit formatting commands. Such a file can be compiled directly and by a preprocessor be transformed into a LaTeX source file. We call it a *program with documentation file* (*PD file*).

The paper is structured as follows: Sections 2 through 6 explain the structure of PD files and the markup language. Section 7 gives some basic information about LaTeX. Section 8 describes installation and use of the PD system, the preprocessor converting PD files to LaTeX.

2 Structure of PD Files

A PD file consists of alternating *program* and *documentation* sections. From the point of view of the compiler, documentation sections are just commentaries and are ignored. The separation between these sections is achieved by special lines in the source file.

```
(******************************************************************
This is a first documentation section.

Here is another paragraph within that section.

*******************************************************************)

(* Here is a program section *)

MODULE Example;

FROM … IMPORT

(*
Here a second documentation section.

*)

VAR x: integer;
…
```

A *documentation section* starts with a line containing the “commentary start” bracket of the respective programming language (which is “(*” in Modula-2, “/*” in C) followed by zero or more stars making up the rest of the line. Except for blanks, there may be no other symbols in such a line. Similarly, it ends with a line containing zero or more stars followed by the “commentary end” bracket of the programming language (“*)” in Modula-2, “*/” in C, for example). Inline comments within a program or commentaries bracketed in any other way will not be recognized as documentation sections; hence no formatting will be applied to them.

Within documentation sections, the text consists of a sequence of *paragraphs*. A paragraph is given in the PD file as a sequence of lines followed by an empty line. An empty line consists of an end-of-line character, possibly followed by blanks. Note in particular that an empty line is needed to finish a paragraph just at the end of a documentation section, hence the following will lead to a syntax error:

```
(*
My little documentation section            is wrong
*)
```

In the following sections we discuss specifications of paragraph formats, character formats, and special characters.

3 Paragraph Formats

The way a paragraph should be formatted is specified by a few initial characters. Several kinds of paragraphs are predefined, described in the following subsections. Finally, it is possible to define new paragraph formats (Section 3.7).

3.1 Standard

If a paragraph does not start with a blank, a digit, or a square bracket, it is a standard paragraph and will be formatted according to the style for normal paragraphs (usually left- and right-adjusted, Times Roman font, etc.). This would look as follows:

```
Here is a little paragraph to be formatted as a standard
paragraph.
Note that it does not matter, how this material
is put into lines. In particular, there may be blanks in between; sequences
of blanks are reduced to a single blank. However, we recommend not to put
any line breaks into a paragraph because then a paragraph stays formatted
also after changes.
```

>>>>>

Here is a little paragraph to be formatted as a standard paragraph. Note that it does not matter, how this material is put into lines. In particular, there may be blanks in between; sequences of blanks are reduced to a single blank. However, we recommend not to put any line breaks into a paragraph because then a paragraph stays formatted also after changes.

<<<<<

3.2 Headings

A first level heading (“Section”) starts with one or two digits, followed by a blank. Second or third level headings contain dots.

```
1 This specifies a first level heading

1.1 This is a second level heading

2.3.3 This is a third level heading
```

>>>>>

1 This specifies a first level heading

1.1 This is a second level heading

2.3.3 This is a third level heading

<<<<<

Note that the actual numbers are not necessarily (if the LaTeX preprocessor is used, then certainly not) taken into the formatted document. Also the actual size of headings depends on the document style.

3.3 Displayed Paragraphs

```
........A displayed paragraph starts with 8 blanks (or a tab symbol) in the
PD file (blanks are only in this document shown as dots). It will be
typeset indented from the left margin.
```

>>>>>

> A displayed paragraph starts with 8 blanks (or a tab symbol) in the PD file. It will be typeset indented from the left margin.

<<<<<

3.4 Item Lists and Enumeration Lists

There are lists and paragraphs within lists available at a first and a second level (of indentation).

```
At the first level, a paragraph starts with four designating characters.
The following cases occur:

..*.This is the start of a “bulleted” paragraph within an item list. The
    following lines may also start with four blanks (to achieve a nice
    appearance of the PD file itself) but this does not
really matter.

..*.A second bulleted paragraph may follow.
```

>>>>>

At the first level, a paragraph starts with four designating characters. The following cases occur:

- This is the start of a “bulleted” paragraph within an item list. The following lines may also start with four blanks (to achieve a nice appearance of the PD file itself) but this does not really matter.
- A second bulleted paragraph may follow.

<<<<<

```
    An enumeration list has numbers consisting of one or two digits in place of
    the star symbol of an item list.

    ..1.This is the first item. Again,
    it does not matter, how the following
    lines start.

    .15.This is the second item. The actual number used doesn't matter, items
    will be numbered in the order of appearance.

    ....This is a follow-up paragraph within an enumeration list or an item
    list. The four leading blanks indicate that it receives a first level
    indentation.
```

>>>>>

An enumeration list has numbers consisting of one or two digits in place of the star symbol of an item list.

1. This is the first item. Again, it does not matter, how the following lines start.
2. This is the second item. The actual number used doesn't matter, items will be numbered in the order of appearance.

   This is a follow-up paragraph within an enumeration list or an item list. The four leading blanks indicate that it receives a first level indentation.

<<<<<

```
    The same schema applies also to the next level. Bullet items, numbered
    items, or follow-up paragraphs are distinguished from the corresponding
    first level entities by two additional leading blanks. Hence, one can
    construct a two-level nested list as follows:
```

```
..*.This is again the first item of an item list. But now, it may contain a
second level enumeration list:

....1.This is the first item at the second level.

......It can also have follow-up paragraphs.

....2.This is the second item of the second level.

..*.And here is the next item of the first level item list. It can again
have a follow-up paragraph, indented to the first level:

....This is the follow-up. Note that the actual appearance of the
paragraphs, that is, the amount of indentation, and the amount of space
below paragraphs, depends on the style used.
```

>>>>>

The same schema applies also to the next level. Bullet items, numbered items, or follow-up paragraphs are distinguished from the corresponding first level entities by two additional leading blanks. Hence, one can construct a two-level nested list as follows:

- This is again the first item of an item list. But now, it may contain a second level enumeration list:

  1. This is the first item at the second level.

     It can also have follow-up paragraphs.

  2. This is the second item of the second level.

- And here is the next item of the first level item list. It can again have a follow-up paragraph, indented to the first level:

  This is the follow-up. Note that the actual appearance of the paragraphs, that is, the amount of indentation, and the amount of space below paragraphs, depends on the style used.

<<<<<

3.5 Figures

Another kind of paragraph is a figure, consisting of a figure caption and the actual drawing. It is described by 16 leading blanks (or some equivalent in tabs and blanks, e.g. 2 tabs). For example:

```
................Figure 1: Representation of a LISP
                          expression [mytext.Figure1.eps]
```

Here “mytext.Figure1.eps” must be the name of an encapsulated postscript file containing a drawing for Figure 1. The file must be element of a directory called “Figures” within the same directory as the file “mytext” itself. As a result, the figure will be embedded into the formatted document like this:

>>>>>

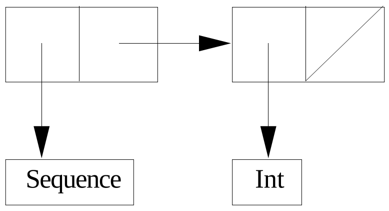

Figure 1: Representation of a LISP expression

<<<<<

LaTeX generates the string “Figure 1:” including the figure number itself, so the figure number used here does not really matter; figures are numbered sequentially (like sections). Hence a figure paragraph consists of the following parts: (i) the string after the leading blanks up to the colon – which is ignored, (ii) everything from there to the file name in square brackets, which must be the end of the paragraph, and (iii) the file name in square brackets.

3.6 Program Displays

Sometimes it is necessary to cite a piece of a program within a documentation section. This should be printed indented, as is, without being interpreted either by the PD preprocessor or by LaTeX or by the compiler. This can be achieved as follows:

```
----    PROCEDURE CreateSegment
          (VAR SegId    : CARDINAL       (* out *);
           VAR Error    : SMErrorType    (* out *));
----
```

>>>>>

```
        PROCEDURE CreateSegment
          (VAR SegId    : CARDINAL       (* out *);
           VAR Error    : SMErrorType    (* out *));
```

<<<<<

Such a program display can occur whenever a new paragraph is expected; it is recognized by the four leading hyphens “`----`”. From then on, everything is taken to be a program display until another sequence of four hyphens occurs at the beginning of a line.

3.7 Special Paragraphs

Apart from these implicitly specified paragraphs, one can define special paragraph formats. A definition for these must be given within a “header” as a part of a documentation section of the PD file, preferably of the *first* documentation section. A special paragraph format is recognized within the text by a number in square brackets at the beginning of the paragraph.

```
    [10] It is recommended to use two-digit numbers for paragraph formats. This
    is because we want to reserve single digits for special character formats
    (see below). With such a special paragraph, one can introduce all kinds of
    formatted material.
```

```
>>>>>
```

It is recommended to use two-digit numbers for
paragraph formats. This is because we want to
reserve single digits for special character formats (see
below). With such a special paragraph, one can
introduce all kinds of formatted material.

```
<<<<<
```

The number used can be resolved in one of two ways. The first is to offer a definition of a paragraph format with this number in a header of the PD file, for example:

```
    //paragraph [10] CenteredQuotation: [\begin{center}]   [\end{center}]
```

This definition consists of four pieces of text after the keyword "paragraph" indicating a special paragraph format definition. Namely, "10" is the format number, "CenteredQuotation" a format name. Then, in the first pair of square brackets, there is some text that should be prefixed to the paragraph, and in the second pair some text that should be appended. Normally this is LaTeX code achieving the desired effect (but nobody prevents one from prepending and appending any kind of text).

The second way (in principle) is to give a definition for the number at the end of this paragraph, for example as follows:

```
    [10] It is recommended to use two-digit numbers for paragraph formats. This
    is because we want to reserve single digits for special character formats
    (see below). With such a special paragraph, one can introduce all kinds of
    formatted material.
                                           //[10:CenteredQuotation]
```

In this case, the number 10 refers only to the format specification at the end of the paragraph. Hence, "CenteredQuotation" may have been defined under any number within the header of the PD file.

An advantage of the first technique is that one doesn´t have to clutter the text with the explicit name of the paragraph format, as with the second technique. The disadvantage is that one has to remember the number associated with this paragraph format. The second technique obviously has the opposite advantages and problems.

The last line of the paragraph in the last example contains a *paragraph annotation*. In general, such a line consists of any number of blanks (including zero) followed by two slashes after which the annotation information follows. There may be any number of annotation lines at the end of a paragraph.

Other kinds of annotation information are explained below. Note however, that this second technique (using annotation lines) has not yet been implemented in the current version of the *PD System*.

After a special paragraph format has been used once, one can write further paragraphs in this format by starting a paragraph just with empty square brackets.

```
[] This paragraph will also get the ``CenteredQuotation´´ format which has
been introduced above.
```

>>>>>

This paragraph will also get the “CenteredQuotation”
format which has been introduced above.

<<<<<

In other words, a special paragraph format remains in effect to simplify the repeated use of one special format. Note that other (non-special) paragraph formats may be used in between.

4 Character Formats

A character format consists of a *font*, a *font size*, a *weight* (e.g. bold face or not), an *angle* (italics or not), a *baseline specification* (e.g. lowered or raised by some points with respect to the paragraph base line), and for some text editors even more than that.

Each paragraph has a default character format (e.g. Times Roman, 10pt, not bold, no italics, 0 points). Without special treatment in the PD file, all characters in a paragraph will receive the default character format. We will introduce facilities to specify italics and bold face directly, because these are the most frequently used among the non-default character formats. In addition, it is possible to define special character formats and to refer to them in the text.

4.1 Italics

Italics are very often needed to describe *emphasis*. Italics are specified by enclosing a string of characters within a paragraph by the “~” symbol.

```
A character format consists of a ~font~, a ~font size~, a ~weight~ (e.g.
bold face or not), an ~angle~ (italics or not), a ~baseline specification~
(e.g. lowered or raised by some points with respect to the paragraph base
line), and for some text editors even more than that.
```

>>>>>

A character format consists of a *font*, a *font size*, a *weight* (e.g. bold face or not), an *angle* (italics or not), a *baseline specification* (e.g. lowered or raised by some points with respect to the paragraph base line), and for some text editors even more than that.

<<<<<

4.2 Bold Face

```
*Bold face* is specified by enclosing a string of characters by star
symbols. It is also possible to use *~italics~ and bold face* nested.
However, due to the behaviour of LaTeX, for each character only the
innermost specification containing it will have an effect.
```

>>>>>

Bold face is specified by enclosing a string of characters by star symbols. It is also possible to use *italics* **and bold face** nested. However, due to the behaviour of LaTeX, for each character only the innermost specification containing it will have an effect.

<<<<<

4.3 Special Character Formats

Like special paragraph formats, these are defined in the header of the PD file, for example (here in LaTeX):

```
//characters [1] Type: [\underline{\it ]   [}]
```

This definition works exactly in the same way as that of special paragraph formats: material in the first pair of square brackets (after the colon) is prepended to a string of characters; material in the second pair of brackets is put behind it.

One refers to these special formats by enclosing a string of characters within a paragraph in double quotes and an attached character format number in square brackets. Note that double quotes are normally not used in LaTeX text (see Section 6); hence they can be used for this purpose.

```
For example, one can refer to data types "points"[1], "lines", and
"regions" in this way. A character format once introduced remains in effect
for subsequent special format character strings. Hence we can now refer to
"points", "lines", and "regions" again.
```

>>>>>

For example, one can refer to data types *points*, *lines*, and *regions* in this way. A character format once introduced remains in effect for subsequent special format character strings. Hence we can now refer to *points*, *lines*, and *regions* again.

<<<<<

The two reference techniques introduced for special paragraph formats are valid here as well (in principle, but remember that annotation lines are not yet implemented in the PD System). Special paragraph and character formats may also be used in conjunction:

```
[20] Here is again a paragraph in the ~CenteredQuote~ format which contains
special character formats as well. For example, assume we have in the
header defined a font ~Underlined~ and another font ~Program~ (to describe,
for example key words of a programming language). Then we can still refer
to types "points[1]", "lines", and "regions", but also use "underlined
text"[2] as well as key words like "begin"[3], "if", "then", "else", and so
forth.
```

>>>>>

Here is again a paragraph in the *CenteredQuote* format which contains special character formats as well. For example, assume we have in the header defined a font *Underlined* and another font *Program* (to describe, for example key words of a programming language). Then we can refer still to types *points*, *lines*, and *regions*, but also use underlined text as well as key words like `begin`, `if`, `then`, `else`, and so forth.

<<<<<

5 Special Characters

Often one needs characters that are not present in ASCII files. We describe such characters by sequences of non-digit characters enclosed in square brackets. The meaning of such sequences is again defined in the header of a PD file. For example, in the header we might have the following definitions:

```
    //[ae] [\"{a}]
    //[oe] [\"{o}]
    //[ue] [\"{u}]
    //[ss] [{\ss}]
    //[x]  [$\times $]
    //[->] [$\rightarrow $]
    //[forall] [\forall]
```

Then we can write, for example:

```
    Ralf Hartmut G[ue]ting begr[ue][ss]t es, da[ss] er Professor an der
    FernUniversit[ae]t Hagen ist.

    ........[forall] geo in GEO. equal: geo [x] geo [->] "bool"[1]
```

>>>>>

Ralf Hartmut Güting begrüßt es, daß er Professor an der FernUniversität Hagen ist.

$\forall$ geo in GEO. equal: geo $\times$ geo $\rightarrow$ *bool*

<<<<<

6 Predefined Special Characters and “Escaping”

Whenever text is intermixed with formatting specifications one has the problem that special symbols or character strings used for the specifications might occur in the normal text as well. One needs a mechanism to prevent their interpretation as formatting specification.

6.1 Escaping Paragraph Formats

A typeset paragraph does not start with blanks. Hence there should be no problem with the predefined formats for *displays*, *lists*, and *figures*. A paragraph may start with a number as used to specify *headings*. One can escape all headings by adding one leading blank, except for a two digit integer which would then be interpreted as an item in an enumeration list. In this case one can escape by using two leading blanks. Another possibility is to use 7 leading blanks which works for all numbers starting a paragraph.

There is no reason for a paragraph to start with four hyphens (*program display*). A paragraph starting with a one- or two-digit number will be interpreted as a special paragraph (if the number is defined). To escape, add a leading blank.

To summarize: *One can escape all special paragraph formats by either one or two leading blanks. Seven leading blanks work in all cases.*

6.2 Escaping Predefined Characters

The following characters have a special meaning: ~, *, ", [, and]. The first three can be printed as they are by enclosing them in square brackets. In the output of the preprocessor, these brackets are stripped off. The three characters are also not interpreted if they are enclosed by blanks. However, the blanks will also appear in the output. Note that the " symbol is not needed in normal text because in LaTeX one writes opening double quotes as `‘‘` and closing double quotes as `’’`, using the two available types of apostrophs.

7 Some Hints about LaTeX

PD files, as described in the previous sections, are transformed by the *PD System* into LaTeX source files. On the one hand, this provides more or less unlimited “formatting power” since it is possible to write directly LaTeX specifications into the documentation sections of PD files. On the other hand, it leads to certain restrictions because some characters have a special meaning in LaTeX and cannot be used just as text characters. One should therefore know a little bit about LaTeX. Some basic information is given in this section. For more details, see a LaTeX book (e.g. [La86]).

The following characters can be used without problems in LaTeX source files (and hence, in text within PD file documentation sections):

```
A..Z, a..z, 0..9
. : ; , ? ! ` ´ ( ) [ ] - / * @ + =
```

However, the ten characters

```
# $ % & ~ _ ^ \ { }
```

are used only in LaTeX commands and may therefore not occur in ordinary text. Some of these can be produced in ordinary text by using a preceding backslash character:

```
\# \$ \% \& \_ \{ \}
```

The other three characters (~, ^, and \) can be produced only in “verbatim” environments in LaTeX. In running text, these are created by enclosing the character by “`\verb+`” and “`+`”. Hence the three characters could be written as

```
\verb+~+, \verb+^+, and \verb+\+
```

as far as LaTeX is concerned (but the “~” character would be interpreted by the PD System and lead to a syntax error there; this can be circumvented as shown below). Finally, the three characters

```
< > |
```

may occur in LaTeX only within mathematical formulas (formula environments); these are explained below. Note that within PD files, all program sections as well as all program displays (Section 3.6) are translated into “verbatim sections” of the LaTeX source files; hence in these places one can use any character whatsoever without problems, it is interpreted neither by the PD system, nor by LaTeX. Apart from that, one may of course use the text substitution capabilities of the PD system to produce these characters, for example by defining:

```
//[tilde] [\verb+~+]
```

Another possibility is to define a special character format to produce the “`\verb+`” and “+” brackets.

Some useful special characters in German are ä, ö, ü, ß; they are produced in LaTeX by

```
\"{a}, \"{o}, \"{u}, {\ss}
```

Mathematical symbols can be used only in formula environments. Such environments are created, for example, by enclosing material in “$” symbols. Hence

```
$ x \in (A \union B) $
```

would be formatted as (in fact, somewhat prettier in LaTeX):

$x \in (A \cup B)$

In PD files one can, for example, define formula environments as a special character format:

```
//characters [1] formula: [$] [$]
```

After a first use of this format (e.g. `"  "[1]`) it can be invoked simply by quote symbols, for example:

```
Let "n" be "O(m^{2})"
```

would result in: Let n be $O(m^2)$. This shows also how superscripts are made, subscripts are written “`m_{i-1}`” to produce $m_{i\text{-}1}$. Lots of LaTeX codes are available to produce special characters or mathematical symbols, see e.g. [La86, Sections 3.2 and 3.3.2].

8 Using the PD System

8.1 Installation, System Environment

The system is available as a file “pdsystem.gz”. Apply the sequence of commands:

```
gunzip pdsystem.gz
tar xf pdsystem
```

The second command creates a directory “PDSystemPublic”. Descend into this directory and issue a “make” command. All components of the system should then be available.

The PD system needs the following other programs:

```
latex                    produces a formatted document
xdvi                     let´s you view this document (except figures)
dvips                    converts to postscript and prints
pageview                 postscript viewer (shows also figures)
```

Additionally, one needs a tool to produce drawings in encapsulated postscript format, e.g. “xfig”. The programs *latex* and *dvips* belong to the standard TeX distribution, *xdvi* and *xfig* to X11R5, *pageview* is part of *openwin*. On our UNIX system, environment variables have to be set as follows:

```
setenv TEXINPUTS '.:/usr/local/TeX/lib/inputs:/usr/local/TeX/lib/inputs209'
setenv TEXFONTS
'/usr/local/TeX/lib/fonts:/usr/local/TeX/lib/fonts/tfm:/usr/local/lib/tex/d
vitoolfonts'
set path = ($path /usr/local/TeX/bin /usr/local/X11R5/bin)
```

If these programs are installed correctly, then the PD system should now be usable. For example, issue a command

```
pdview docu
```

to convert the “docu” file (a PD file) to LaTeX and to look at it under the “xdvi” viewer. This is the documentation of the PD system itself. An initial part of this PD file is given in Appendix A of this paper; the corresponding formatted document is shown in Appendix B.

8.2 Commands

The following commands are offered by the PD System:

- `pdview` *`pdfile`*.

  Show the PD file *pdfile* as a formatted document in the xdvi viewer. Does not show figures (but leaves space for them).

- `pdshow` *`pdfile`*.

  Show the PD file *pdfile* in the postscript viewer. Figures do appear, but the quality is not as good as with xdvi.

- `pdprint` *`pdfile`*.

  Prints the PD file *pdfile* . The file must have been viewed under *pdview* before (because this command procedure starts from *pdfile*.dvi created there).

- `pd2tex` *`pdfile`*.

  Just create the Latex source file *pdfile*.tex from the PD file *pdfile*.

- `pdshowtex` *`pdfile`*.

  Show the PD file *pdfile* in the postscript viewer, starting from the LaTeX version *pdfile*.tex (created by *pd2tex*).

- `print` *`pdfile`*.

  Print the PD file *pdfile* directly (unformatted) in ASCII. This introduces line breaks within paragraphs before printing. It is recommended not to use line breaks within paragraphs in the source file because in this way a paragraph can be modified and remain reasonably formatted there.

- `linebreaks <` *`pdfile`* `>` *`pdfile`*`.lb`

  Produce a version *pdfile*.lb from pdfile which has line breaks within paragraphs (for example, to send this file by email).

8.3 Errors

Unfortunately, the PD system has not yet been equipped with intelligent error messages. Whenever it discovers an error in the source file *pdfile*, it displays a message “syntax error” and stops writing further paragraphs to the file *pdfile*.tex that it is producing. If this occurs in a command procedure such as *pdview*, LaTeX will take over directly and after a while complain about its incomplete source file. Hence, watch out for a message “syntax error” before LaTeX starts “This is TeX, ...”. To find the error, look at the file *pdfile*.tex to determine the last paragraph written; the error must be in the next paragraph.

Common errors are unmatched “parentheses” of some kind, for example, a “~” or “*” symbol interpreted by PD system as an opening bracket for italic or bold face text without a matching closing bracket. Other errors are paragraph formats within lists which are not allowed there, or a missing empty line to finish a paragraph at the end of a documentation section. In spite of the lack of error messages, it is usually easy to spot the error.

8.4 Documenting Program Systems

Only small programs reside in a single file; usually one deals with program systems consisting of several, sometimes of many files. One can produce a coherent documentation for a program system by concatenating all relevant files into a “documentation file”; the command producing the concatenation should be included in the makefile for the program. In this way, every change in any of the component programs is directly reflected in the documentation. The resulting document may, for example, have an introductory section describing the overall structure of the program system, and then deal with the various modules in appropriate sections and subsections. This technique has been employed in the documentation file “docu” of the PD system itself which corresponds to reference [Gü95] (the makefile is also contained in Section 9). We recommend [Gü95] as an example of documenting whole program systems as well as of the use of markup techniques in PD files. As mentioned before, an initial part of it is shown in the Appendices.

References

[Gü95] Güting, R.H., The PD System: Integrating Programs and Documentation. Praktische Informatik IV, FernUniversität Hagen, Software-Report 2, May 1995.

[La86] Lamport, L., LaTeX: A Document Preparation System. User's Guide & Reference Manual. Addison-Wesley, 1986.

Appendix A: Documentation of PD System (Initial Part) as a PD file

```
/*
//paragraph [1] Title: [{\Large \bf] [}]
//[ue] [\"{u}]


[1] The PD System: Integrating Programs and Documentation

Ralf Hartmut G[ue]ting

May 1995



1 Overview

The purpose of ~PDSystem~ is to allow a programmer to write ASCII program files
with embedded documentation (~PD~ stands for ~Programs with Documentation~).
Such files are called ~PD files~. Essentially a PD file consists of alternating
~documentation sections~ and ~program sections~. Within documentation sections,
one can describe a number of paragraph formats (such as headings, displayed
material, etc.), character formats (e.g. italics), and special characters (e.g.
``[ue]''). How this is done, is described in the document ``Integrating Programs
and Documentation'' [G[ue]95].

The main component of PDSystem is an executable program ~maketex~ which converts
a PD file into a LaTeX file. More precisely, given a file ~pdfile~, a LaTeX file
~pdfile.tex~ is created as follows: First, a standard header for LaTeX is copied
from a file ~pd.header~ into ~pdfile.tex~.Then, ~pdfile~ is first run through a
filter program called ~tabs~ which converts tabulator symbols into corresponding
sequences of blanks. The output of this filter is fed into ~maketex~ (which can
therefore be sure not to encounter any tab symbols) which converts material in
documentation sections into LaTeX code and encloses program sections by
``verbatim'' commands to force TeX to typeset them exactly as they have been
written.

A PD file may contain very long lines, because CR (end of line) symbols need
only be present to delimit paragraphs. To make the output file ~pdfile.tex~
easily printable, the output of ~maketex~ is run through a further filter called
~linebreaks~ which introduces CR symbols such that no lines with more than 80
characters are in the output. In total, we have the processing steps illustrated
in Figure 1.

                Figure 1: Construction of a TeX file from a PD file
                                [pddocu.Figure1.eps]

The file ~pd.header~ is shown in Section 6.1. ~Cat~ is a UNIX system command.
Executables ~tabs~ and ~linebreaks~ are created from the corresponding C
programs ~tabs.c~ and ~linebreaks.c~ shown in Section 7. The programs leading to
~maketex~ are described below. The complete process shown in Figure 1 is
executed by a command procedure called ~pd2tex~ given in Section 8.4.

There are further command procedures:

  * ~pdview~ allows one to view a PD file under the ~xdvi~ viewer (after it has
been processed by LaTeX. This viewer has a good resolution, but does not
show embedded postscript figures.

  * ~pdshow~ shows a PD file using the postscript viewer ~pageview~. This shows
embedded figures. However, the quality of the display is not as good as with
~xdvi~.

  * ~pdprint~ prints a PD file. The file must have been viewed under  ~pdview~
before, because this procedure starts from ~pdfile~.dvi.

  * ~pdshowtex~ shows a PD file starting from its TeX version ~pdfile~.tex
(created by ~pd2tex~).

As an example, the processing steps used in ~pdshow~ are illustrated in Figure
2.

                Figure 2: Steps of ~pdshow~ [pddocu.Figure2.eps]
```

We now consider the construction of ~maketex~, the central component of the PD system. ~Maketex~ depends on the components shown in Figure 3.

Figure 3: Components for building ~maketex~ [pddocu.Figure3.eps]

These components play the following roles:

* ~Lex.l~ is a ~lex~ specification of a lexical analyser (which is transformed by the UNIX tool ~lex~ into a C program ~lex.yy.c~). The lexical analyser reads the input PD file and produces a stream of tokens for the parser.

* ~Parser.y~ is a ~yacc~ specification of a parser (transformed by the UNIX tool ~yacc~ into a C program ~y.tab.c~). The parser consumes the tokens produced by the lexical analyser. On recognizing parts of the structure of a PD file, it writes corresponding LaTeX code to the output file.

* ~NestedText.c~ is a ``module'' in C providing a data structure for ``nested text'' together with a number of operations. This is needed because text for the output file cannot always be created sequentially. Sometimes it is neccessary, for example, to collect a piece of text from the source file into a data structure and then to create enclosing pieces of LaTeX code before and after it. The ~NestedText~ data structure corresponds to a LISP ``list expression'' and is a binary tree with character strings in its leaves. There are operations available to create a leaf from a string, to concatenate two trees, or to write the contents of a tree in tree order to the output.

* ~ParserDS.c~ contains a number of data structures needed by the parser. These data structures are used to keep definitions of special paragraph formats, special character formats, and special characters, which can be defined in header documentation sections of PD files.

* ~Maketex.c~ is the main program. It does almost nothing. It just calls the parser; after completion of parsing (which includes translation to Latex) a final piece of Latex code is written to the output.

Figure 3 describes the dependencies among these components at a logical level. An edge describes the ``uses'' relationship. For example, the ~NestedText~ module is used in the lexical analyser, the parser and in the parser data structure component. Figure 4 shows these dependencies at a more technical level, as they are reflected in the ~makefile~ (see Section 9).

Figure 4: Technical dependencies in the construction of the executable ~maketex~ [pddocu.Figure4.eps]

Here each box corresponds to a file. An unlabeled arrow means the ``include'' relationship (for example, ~NestedText.h~ is included into ~Lex.l~, ~Parser.y~, and ~ParserDS.c~. Edges labeled with ~lex~, ~yacc~, and ~cc~ mean that the tools ~lex~ and ~yacc~ or the C compiler, respectively, produce the result files. Fat edges connecting files indicate that these files are compiled together.

The rest of this document is structured as follows: Section 2 describes the ~NestedText~ module (files ~NestedText.h~ and ~NestedText.c~), Section 3 the lexical analyser (~Lex.l~), Section 4 ~ParserDS.c~, Section 5 the parser itself (~Parser.y~). Section 6 shows the header file for Latex and the rather trivial program ~Maketex.c~. Section 7 contains the auxiliary functions ~tabs.c~ and ~linebreaks.c~. Section 8 gives the command procedures ~pdview~, ~pdshow~, etc. Finally, Section 9 contains the ~makefile~.

```
*/
/***************************************************
//[x] [$\times $]
//[->] [$\rightarrow $]
//paragraph [2] verse:  [\begin{verse}] [\end{verse}]
```

2 The Module NestedText

2.1 Definition Part

(File ~PDNestedText.h~)

This module allows one to create nested text structures and to write them to standard output. It provides in principle a data type ~listexpr~ (representing such structures) and operations:

```
[2]     atom: string [x] int [->] listexpr              \\
        atomc: string [->] listexpr                     \\
        concat: listexpr [x] listexpr [->] listexpr     \\
        print: listexpr [->] e                          \\
        copyout: listexpr [->] string [x] int           \\
        release-storage                                 \\
```

However, for use by ~lex~ and ~yacc~ generated lexical analysers and parsers which only allow to associate integer values with grammar symbols, we represent a ~listexpr~ by an integer (which is, in fact, an index into an array for nodes). Hence we have a signature:

```
[2]     atom: string [x] int [->] int                   \\
        atomc: string [->] int                          \\
        concat: int [x] int [->] int                    \\
        print: int [->] e                               \\
        copyout: int [->] string [x] int                \\
        release-storage                                 \\
```

The module uses two storage areas. The first is a buffer for text characters, it can take up to STRINGMAX characters, currently set to 30000. The second provides nodes for the nested list structure; currently up to NODESMAX = 30000 nodes can be created.

The operations are defined as follows:

```
----    int atom(string, length)
        char *string;
        int length;
----
```

List expressions, that is, values of type ~listexpr~ are either atoms or lists. The function ~atom~ creates from a character string ~string~ of length ~length~ a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
----    int atomc(string)
        char *string;
----
```

The function ~atomc~ works like ~atom~ except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
----    int concat(list1, list2)
        int list1, list2;
----
```

Concats the two lists; returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
----    print(list)
        int list;
----
```

Writes the character strings from all atoms in ~list~ in the right order to standard output.

```
----    copyout(list, target, lengthlimit)
        int list;
        char *target;
        int lengthlimit;
----
```

Copies the character strings from all atoms in ~list~ in the right order into a string variable ~target~. Parameter ~lengthlimit~ ensures that the maximal available space in ~target~ is respected; an error occurs if the list expression ~list~ contains too many characters.

```
----    release_storage()
----

Destroys the contents of the text and node buffers. Should be used only when a
complete piece of text has been recognized and written to the output. Warning:
Must not be used after pieces of text have been recognized for which the parser
depends on reading a look-ahead token! This token will be in the text and node
buffers already and be lost. Currently this applies to lists.

The following is what is technically exported from this file:

***************************************/

int atom(), atomc(), concat();
print(), copyout(), release_storage(),
show_storage();                 /* show_storage used only for testing */
/*****************************************************************************

2.2 Implementation Part

(File ~PDNestedText.c~)

*****************************************************************************/

#include <strings.h>
#include "PDNestedText.h"

#define AND &&
#define TRUE 1
#define FALSE 0
#define NULL -1

#define STRINGMAX 30000
/*
Maximal number of characters in buffer ~text~.

*/

#define NODESMAX 30000
/*
Maximal number of nodes available from ~nodespace~.

*/

struct listexpr {
    int left;
    int right;
    int atomstring;             /* index into array text */
    int length;                 /* no of chars in atomstring*/
};

/*
If ~left~ is NULL then this represents an atom, otherwise it is a list in which
case ~atomstring~ must be NULL.

*/
struct listexpr nodespace[NODESMAX];
int first_free_node = 0;

char text[STRINGMAX];
int first_free_char = 0;

/***************************************

The function ~atom~ creates from a character string ~string~ of length ~length~
a list expression which is an atom containing this string. Possible errors: The
text buffer or storage space for nodes may overflow.

***************************************/

int atom(string, length)
```

```
char *string;
int length;
{
    int newnode;
    int i;

    /* put string into text buffer: */

        if (first_free_char + length > STRINGMAX)
            {printf("Error: too many characters.\n"); exit(1);}

        for (i = 0; i< length; i++)
            text[first_free_char + i] = string[i];

    /* create new node */

        newnode = first_free_node++;
        if (first_free_node > NODESMAX)
            {printf("Error: too many nodes.\n"); exit(1);}

        nodespace[newnode].left = NULL;
        nodespace[newnode].right = NULL;
        nodespace[newnode].atomstring = first_free_char;
            first_free_char = first_free_char + length;
        nodespace[newnode].length = length;
        return(newnode);
}

/****************************************

The function ~atomc~ works like ~atom~ except that the parameter should be a
null-terminated string. It determines the length itself. To be used in
particular for string constants written directly into the function call.

******************************************/

int atomc(string)
char *string;
{   int length;

    length = strlen(string);
    return(atom(string, length));
}

/***************************************

The function ~concat~ concats two lists; it returns a list expression
representing the concatenation. Possible error: the storage space for nodes may
be exceeded.

****************************************/

int concat(list1, list2)
int list1, list2;
{
    int newnode;

    newnode = first_free_node++;

    if (first_free_node > NODESMAX)
        {printf("Error: too many nodes.\n"); exit(1);}

    nodespace[newnode].left = list1;
    nodespace[newnode].right = list2;
    nodespace[newnode].atomstring = NULL;
    nodespace[newnode].length = 0;
    return(newnode);
}

/****************************************
```

```
Function ~print~ writes the character strings from all atoms in ~list~ in the
right order to standard output.

*****************************************/

print(list)
int list;
{
    int i;

    if (isatom(list))
        for (i = 0; i < nodespace[list].length; i++)
            putchar(text[nodespace[list].atomstring + i]);
    else
        {print(nodespace[list].left); print(nodespace[list].right);};
}

/******************************************

Function ~isatom~ obviously checks whether a list expression ~list~ is an atom.

*******************************************/

int isatom(list)
int list;
{
    if (nodespace[list].left == NULL) return(TRUE);
    else return(FALSE);
}

/*******************************************

The function ~copyout~ copies the character strings from all atoms in ~list~ in
the right order into a string variable ~target~. Parameter ~lengthlimit~ ensures
that the maximal available space in ~target~ is respected; an error occurs if
the list expression ~list~ contains too many characters. ~Copyout~ just calls an
auxiliary recursive procedure ~copylist~ which does the job.

*******************************************/

copyout(list, target, lengthlimit)
int list;
char *target;
int lengthlimit;
{   int i;

    i = copylist(list, target, lengthlimit);
    target[i] = '\0';
}

int copylist(list, target, lengthlimit)
int list;
char *target;
int lengthlimit;
{   int i, j;

    if (isatom(list))
        if (nodespace[list].length <= lengthlimit - 1)
            {for (i = 0; i < nodespace[list].length; i++)
                target[i] = text[nodespace[list].atomstring + i];
            return nodespace[list].length;
            }
        else
            {printf("Error in copylist: too long text.\n"); print(list);
            exit(1);
            }
    else
        {i = copylist(nodespace[list].left, target, lengthlimit);
        j = copylist(nodespace[list].right, &target[i], lengthlimit - i);
        return (i+j);
        }
}
```

```
/****************************************

Function ~release-storage~ destroys the contents of the text and node buffers.
Should be used only when a complete piece of text has been recognized and
written to the output. Do not use it for text pieces whose recognition needs
look-ahead!

****************************************/

release_storage()
{   first_free_char = 0;
    first_free_node = 0;
}

/****************************************

Function ~show-storage~ writes the contents of the text and node buffers to
standard output; only used for testing.

****************************************/

show_storage()
{   int i;
    for (i = 0; i < first_free_char; i++) putchar(text[i]);

    for (i = 0; i < first_free_node; i++)
         printf("node: %d, left: %d, right: %d, atomstring: %d, length: %d\n",
                i, nodespace[i].left, nodespace[i].right,
                nodespace[i].atomstring, nodespace[i].length);
}
```

Appendix B: Documentation of PD System (Initial Part) as a Formatted Document

The PD System: Integrating Programs and Documentation

Ralf Hartmut Güting

May 1995

1 Overview

The purpose of *PDSystem* is to allow a programmer to write ASCII program files with embedded documentation (*PD* stands for *Programs with Documentation*). Such files are called *PD files*. Essentially a PD file consists of alternating *documentation sections* and *program sections*. Within documentation sections, one can describe a number of paragraph formats (such as headings, displayed material, etc.), character formats (e.g. italics), and special characters (e.g. "ü"). How this is done, is described in the document "Integrating Programs and Documentation" [Gü95].

The main component of PDSystem is an executable program *maketex* which converts a PD file into a LaTeX file. More precisely, given a file *pdfile*, a LaTeX file *pdfile.tex* is created as follows: First, a standard header for LaTeX is copied from a file *pd.header* into *pdfile.tex*.Then, *pdfile* is first run through a filter program called *tabs* which converts tabulator symbols into corresponding sequences of blanks. The output of this filter is fed into *maketex* (which can therefore be sure not to encounter any tab symbols) which converts material in documentation sections into LaTeX code and encloses program sections by "verbatim" commands to force TeX to typeset them exactly as they have been written.

A PD file may contain very long lines, because CR (end of line) symbols need only be present to delimit paragraphs. To make the output file *pdfile.tex* easily printable, the output of *maketex* is run through a further filter called *linebreaks* which introduces CR symbols such that no lines with more than 80 characters are in the output. In total, we have the processing steps illustrated in Figure 1.

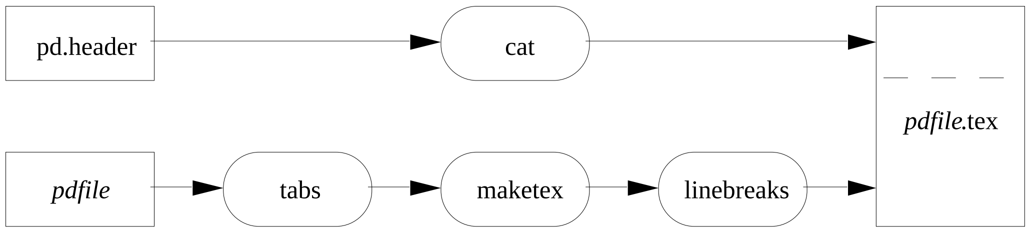

Figure 1: Construction of a TeX file from a PD file

The file *pd.header* is shown in Section 6.1. *Cat* is a UNIX system command. Executables *tabs* and *linebreaks* are created from the corresponding C programs *tabs.c* and *linebreaks.c* shown in Section 7. The programs leading to *maketex* are described below. The complete process shown in Figure 1 is executed by a command procedure called *pd2tex* given in Section 8.4.

There are further command procedures:

- *pdview* allows one to view a PD file under the *xdvi* viewer (after it has been processed by

LaTeX. This viewer has a good resolution, but does not show embedded postscript figures.

- *pdshow* shows a PD file using the postscript viewer *pageview*. This shows embedded figures. However, the quality of the display is not as good as with *xdvi*.
- *pdprint* prints a PD file. The file must have been viewed under *pdview* before, because this procedure starts from *pdfile*.dvi.
- *pdshowtex* shows a PD file starting from its TeX version *pdfile*.tex (created by *pd2tex*).

As an example, the processing steps used in *pdshow* are illustrated in Figure 2.

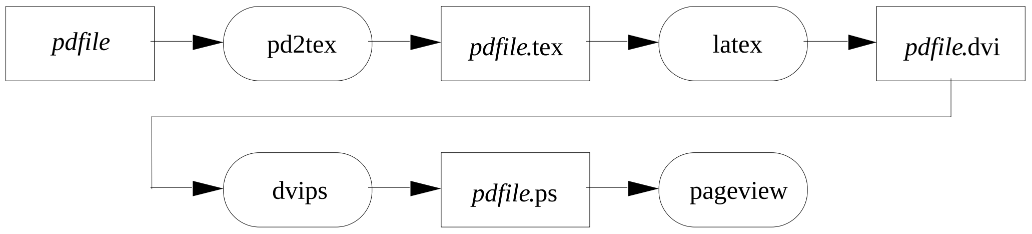

Figure 2: Steps of *pdshow*

We now consider the construction of *maketex*, the central component of the PD system. *Maketex* depends on the components shown in Figure 3.

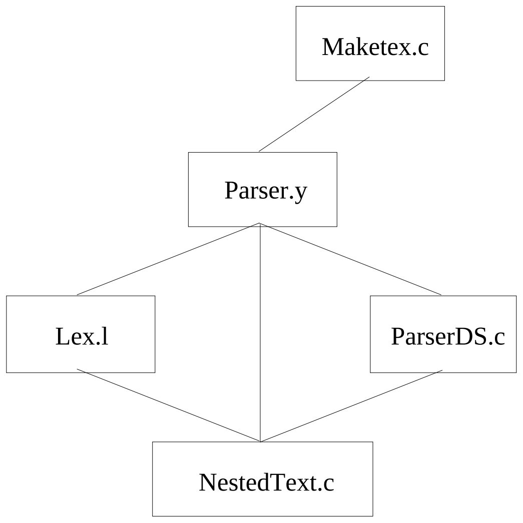

Figure 3: Components for building *maketex*

These components play the following roles:

- *Lex.l* is a *lex* specification of a lexical analyser (which is transformed by the UNIX tool *lex* into a C program *lex.yy.c*). The lexical analyser reads the input PD file and produces a stream of tokens for the parser.
- *Parser.y* is a *yacc* specification of a parser (transformed by the UNIX tool *yacc* into a C program *y.tab.c*). The parser consumes the tokens produced by the lexical analyser. On recognizing parts of the structure of a PD file, it writes corresponding LaTeX code to the output file.
- *NestedText.c* is a "module" in C providing a data structure for "nested text" together with a number of operations. This is needed because text for the output file cannot always be created sequentially. Sometimes it is neccessary, for example, to collect a piece of text from the source file into a data structure and then to create enclosing pieces of LaTeX code before and after it. The *NestedText* data structure corresponds to a LISP "list expression" and is a binary tree with character strings in its leaves. There are operations available to create a leaf from a string, to concatenate two trees, or to write the contents of a tree in tree order to the output.
- *ParserDS.c* contains a number of data structures needed by the parser. These data structures are used to keep definitions of special paragraph formats, special character formats, and special characters, which can be defined in header documentation sections of PD files.
- *Maketex.c* is the main program. It does almost nothing. It just calls the parser; after completion of parsing (which includes translation to Latex) a final piece of Latex code is written to the output.

Figure 3 describes the dependencies among these components at a logical level. An edge describes the "uses" relationship. For example, the *NestedText* module is used in the lexical analyser, the parser and in the parser data structure component. Figure 4 shows these dependencies at a more technical level, as they are reflected in the *makefile* (see Section 9).

Here each box corresponds to a file. An unlabeled arrow means the "include" relationship (for example, *NestedText.h* is included into *Lex.l*, *Parser.y*, and *ParserDS.c*. Edges labeled with *lex*, *yacc*, and *cc* mean that the tools *lex* and *yacc* or the C compiler, respectively, produce the result files. Fat edges connecting files indicate that these files are compiled together.

The rest of this document is structured as follows: Section 2 describes the *NestedText* module (files *NestedText.h* and *NestedText.c*), Section 3 the lexical analyser (*Lex.l*), Section 4 *ParserDS.c*, Section 5 the parser itself (*Parser.y*). Section 6 shows the header file for Latex and the rather trivial program *Maketex.c*. Section 7 contains the auxiliary functions *tabs.c* and *linebreaks.c*. Section 8 gives the command procedures *pdview*, *pdshow*, etc. Finally, Section 9 contains the *makefile*.

2 The Module NestedText

2.1 Definition Part

(File *PDNestedText.h*)

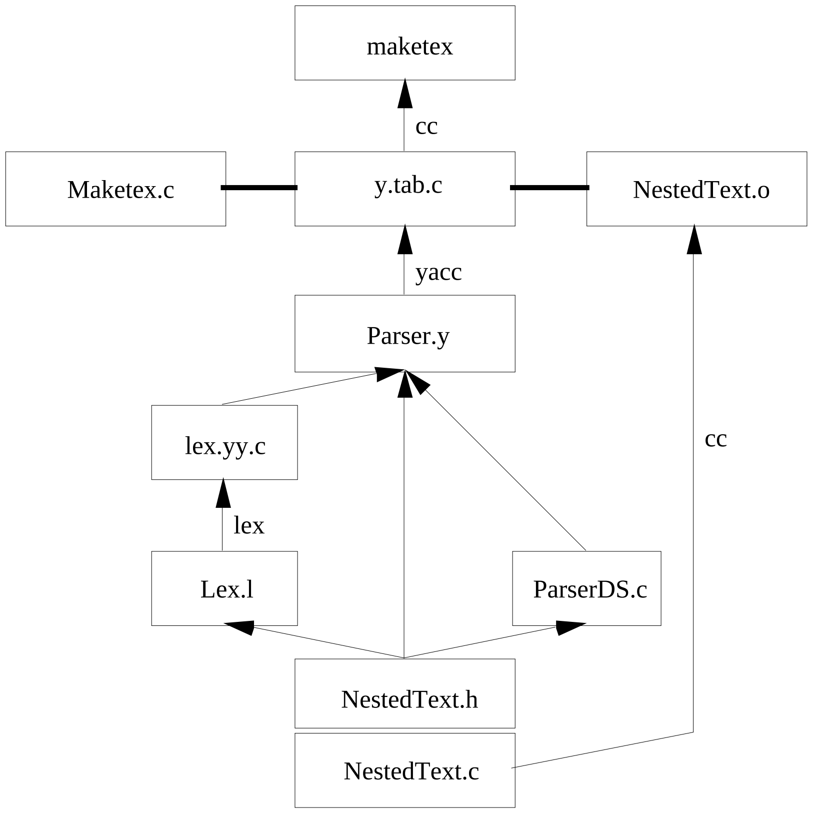

Figure 4: Technical dependencies in the construction of the executable *maketex*

This module allows one to create nested text structures and to write them to standard output. It provides in principle a data type *listexpr* (representing such structures) and operations:

> atom: string $\times$ int $\rightarrow$ listexpr
> atomc: string $\rightarrow$ listexpr
> concat: listexpr $\times$ listexpr $\rightarrow$ listexpr
> print: listexpr $\rightarrow$ e
> copyout: listexpr $\rightarrow$ string $\times$ int
> release-storage

However, for use by *lex* and *yacc* generated lexical analysers and parsers which only allow to associate integer values with grammar symbols, we represent a *listexpr* by an integer (which is, in fact, an index into an array for nodes). Hence we have a signature:

> atom: string $\times$ int $\rightarrow$ int
> atomc: string $\rightarrow$ int
> concat: int $\times$ int $\rightarrow$ int
> print: int $\rightarrow$ e

copyout: int $\rightarrow$ string $\times$ int
release-storage

The module uses two storage areas. The first is a buffer for text characters, it can take up to STRINGMAX characters, currently set to 30000. The second provides nodes for the nested list structure; currently up to NODESMAX = 30000 nodes can be created.

The operations are defined as follows:

```
int atom(string, length)
char *string;
int length;
```

List expressions, that is, values of type *listexpr* are either atoms or lists. The function *atom* creates from a character string *string* of length *length* a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
int atomc(string)
char *string;
```

The function *atomc* works like *atom* except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
int concat(list1, list2)
int list1, list2;
```

Concats the two lists; returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
print(list)
int list;
```

Writes the character strings from all atoms in *list* in the right order to standard output.

```
copyout(list, target, lengthlimit)
int list;
char *target;
int lengthlimit;
```

Copies the character strings from all atoms in *list* in the right order into a string variable *target*. Parameter *lengthlimit* ensures that the maximal available space in *target* is respected; an error occurs if the list expression *list* contains too many characters.

```
release_storage()
```

Destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Warning: Must not be used after pieces of text have been recognized for which the parser depends on reading a look-ahead token! This token will be in the text and node buffers already and be lost. Currently this applies to lists.

The following is what is technically exported from this file:

```
int atom(), atomc(), concat();
print(), copyout(), release_storage(),
show_storage();                 /* show_storage used only for testing */
```

2.2 Implementation Part

(File *PDNestedText.c*)

```
#include <strings.h>
#include "PDNestedText.h"

#define AND &&
#define TRUE 1
#define FALSE 0
#define NULL -1

#define STRINGMAX 30000
```

Maximal number of characters in buffer *text*.

```
#define NODESMAX 30000
```

Maximal number of nodes available from *nodespace*.

```
struct listexpr {
    int left;
    int right;
    int atomstring;             /* index into array text */
    int length;                 /* no of chars in atomstring*/
};
```

If *left* is NULL then this represents an atom, otherwise it is a list in which case *atomstring* must be NULL.

```
struct listexpr nodespace[NODESMAX];
int first_free_node = 0;

char text[STRINGMAX];
int first_free_char = 0;
```

The function *atom* creates from a character string *string* of length *length* a list expression which is an atom containing this string. Possible errors: The text buffer or storage space for nodes may overflow.

```
int atom(string, length)
char *string;
int length;
{
    int newnode;
    int i;

    /* put string into text buffer: */

        if (first_free_char + length > STRINGMAX)
            {printf("Error: too many characters.\n"); exit(1);}

        for (i = 0; i< length; i++)
            text[first_free_char + i] = string[i];

    /* create new node */

        newnode = first_free_node++;
        if (first_free_node > NODESMAX)
            {printf("Error: too many nodes.\n"); exit(1);}

        nodespace[newnode].left = NULL;
        nodespace[newnode].right = NULL;
        nodespace[newnode].atomstring = first_free_char;
            first_free_char = first_free_char + length;
        nodespace[newnode].length = length;
        return(newnode);
}
```

The function *atomc* works like *atom* except that the parameter should be a null-terminated string. It determines the length itself. To be used in particular for string constants written directly into the function call.

```
int atomc(string)
char *string;
{   int length;

    length = strlen(string);
    return(atom(string, length));
}
```

The function *concat* concats two lists; it returns a list expression representing the concatenation. Possible error: the storage space for nodes may be exceeded.

```
int concat(list1, list2)
int list1, list2;
{
    int newnode;

    newnode = first_free_node++;
    if (first_free_node > NODESMAX)
        {printf("Error: too many nodes.\n"); exit(1);}

    nodespace[newnode].left = list1;
    nodespace[newnode].right = list2;
    nodespace[newnode].atomstring = NULL;
    nodespace[newnode].length = 0;
    return(newnode);
}
```

Function *print* writes the character strings from all atoms in *list* in the right order to standard output.

```
print(list)
int list;
{
    int i;

    if (isatom(list))
        for (i = 0; i < nodespace[list].length; i++)
            putchar(text[nodespace[list].atomstring + i]);
    else
        {print(nodespace[list].left); print(nodespace[list].right);};
}
```

Function *isatom* obviously checks whether a list expression *list* is an atom.

```
int isatom(list)
int list;
{
    if (nodespace[list].left == NULL) return(TRUE);
    else return(FALSE);
}
```

The function *copyout* copies the character strings from all atoms in *list* in the right order into a string variable *target*. Parameter *lengthlimit* ensures that the maximal available space in *target* is respected; an error occurs if the list expression *list* contains too many characters. *Copyout* just calls an auxiliary recursive procedure *copylist* which does the job.

```
copyout(list, target, lengthlimit)
int list;
char *target;
int lengthlimit;
{   int i;

    i = copylist(list, target, lengthlimit);
    target[i] = '\0';
}
```

```
int copylist(list, target, lengthlimit)
int list;
char *target;
int lengthlimit;
{   int i, j;

    if (isatom(list))
        if (nodespace[list].length <= lengthlimit - 1)
            {for (i = 0; i < nodespace[list].length; i++)
                target[i] = text[nodespace[list].atomstring + i];
            return nodespace[list].length;
            }
        else
            {printf("Error in copylist: too long text.\n"); print(list);
            exit(1);
            }
    else
        {i = copylist(nodespace[list].left, target, lengthlimit);
        j = copylist(nodespace[list].right, &target[i], lengthlimit - i);
        return (i+j);
        }
}
```

Function *release-storage* destroys the contents of the text and node buffers. Should be used only when a complete piece of text has been recognized and written to the output. Do not use it for text pieces whose recognition needs look-ahead!

```
release_storage()
{   first_free_char = 0;
    first_free_node = 0;
}
```

Function *show-storage* writes the contents of the text and node buffers to standard output; only used for testing.

```
show_storage()
{   int i;
    for (i = 0; i < first_free_char; i++) putchar(text[i]);

    for (i = 0; i < first_free_node; i++)
         printf("node: %d, left: %d, right: %d, atomstring: %d, length: %d\n",
                i, nodespace[i].left, nodespace[i].right,
                nodespace[i].atomstring, nodespace[i].length);
}
```