The PD System: Integrating Programs and Documentation

Ralf Hartmut Güting

May 1995

2002-2004, Markus Spiekermann. Changes of makefiles and shell scripts.

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 *pdtabs* 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.

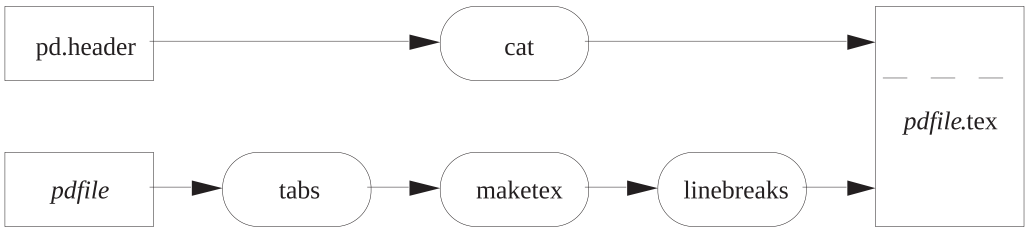

Figure 1: Construction of a TeX file from a PD file

The file *pd.header* is shown in Section 6.1. *Cat* is a UNIX system command. Executables *pdtabs* and *linebreaks* are created from the corresponding C programs *pdtabs.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 unix using the *xdvi* viewer (after it has been processed by LaTeX). On MS-Windows the previewer is called *yap*. The viewer is defined in the environment variable *PD_DVI_VIEWER*.

- *pdshow* shows a PD file using a postscript viewer defined in the environment variable *PD_PS_VIEWER*. This shows embedded figures. However, the quality of the display is not as good as with *xdvi*.
- *pd2pdf* converts a PD file into the portable document format using the program *dvipdfm*.

At a lower level there are three scripts which convert PD to other formats:

- *pd2tex* is called by every script to convert PD into a LaTeX file.
- *pd2dvi* is called by *pdview* to create a DVI file.
- *pd2ps* is called by *pdshow* and creates a postscript file.

These scripts may be useful to recreate a DVI file while previewing an older version. The preview will detect automatically that there exists a newer DVI file and reloads it. Another application may be to use Adobe Distiller in order to create a PDF starting from a postscript file.

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

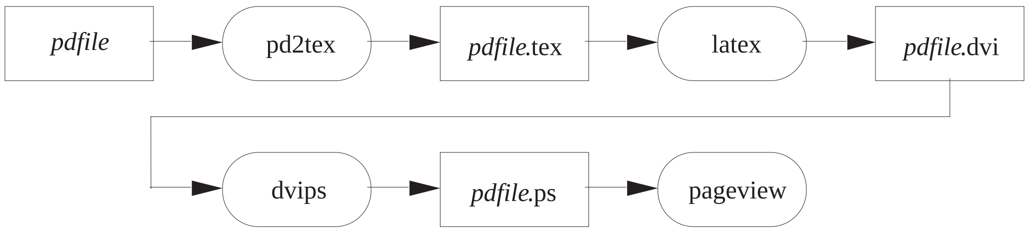

Figure 2: Steps of *pdshow*

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

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 necessary, 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.

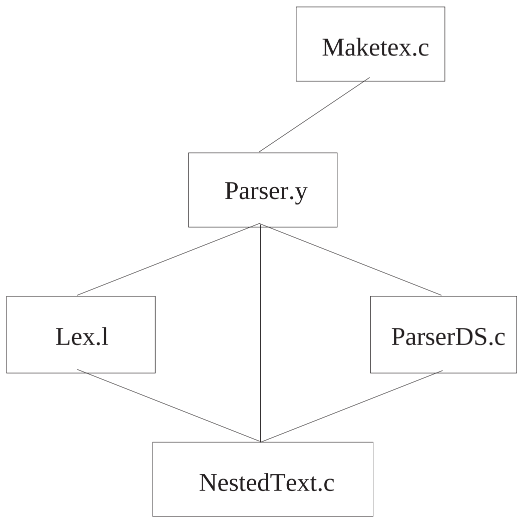

Figure 3: Components for building *maketex*

- *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 *pdtabs.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*)

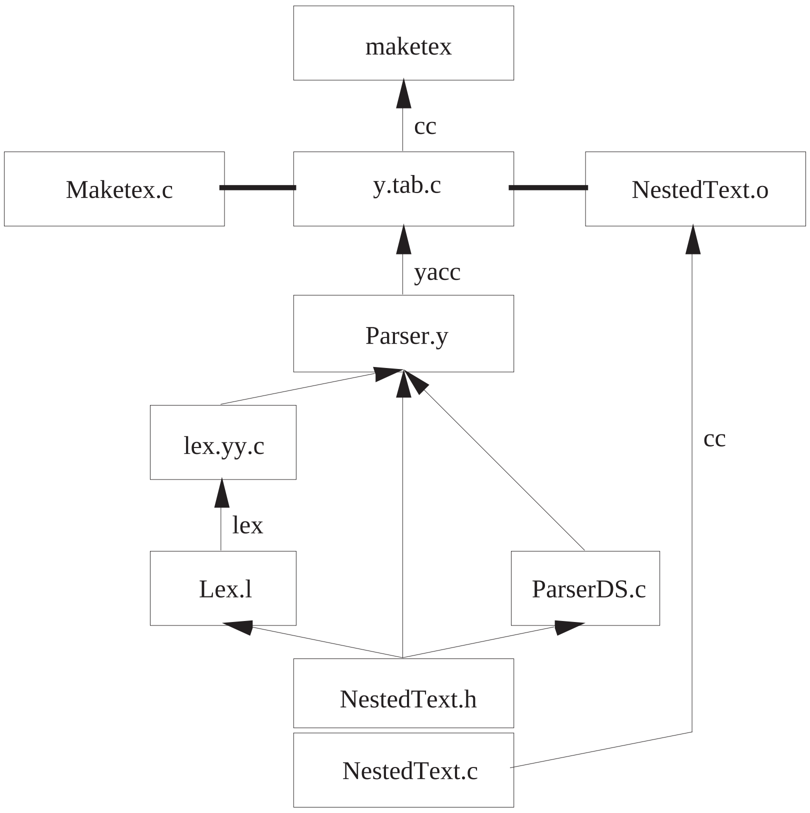

Figure 4: Technical dependencies in the construction of the executable *maketex*

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 STRING-MAX 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(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(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(int list1, int list2)
```

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

```
print(int list)
```

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

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

2.2 Implementation Part

(File *PDNestedText.c*)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "PDNestedText.h"

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

#define STRINGMAX 300000
```

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

    /* put string into text buffer: */

        if (first_free_char + length > STRINGMAX)
            {fprintf(stderr, "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)
            {fprintf(stderr, "Error: too many nodes!!!!.\n");
             show_storage(); 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(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(int list1, int list2)
{
    int newnode;

    newnode = first_free_node++;
```

```
        if (first_free_node > NODESMAX)
            {fprintf(stderr, "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(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(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(int list, char *target, int lengthlimit)
    {   int i;

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

    int copylist(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
                {fprintf(stderr, "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;
        fprintf(stderr,"first_free_char %d\n",first_free_char);
        for (i = 0; i < first_free_char; i++) putchar(text[i]);

        fprintf(stderr,"first_free_node %d\n",first_free_node);

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