PROGDOC - a Program Documentation System

Volker Simonis
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen, 72076 Tübingen, Germany
E-mail : simonis@informatik.uni-tuebingen.de

Version 1.14 (PROGDOC Rel. *Unreleased*) - March 4, 2003

Abstract

Though programming languages and programming styles evolve with remarkable speed today, there is no such evolution in the field of program documentation. And although there exist some popular approaches like Knuth's literate programming system WEB [Web] and nowadays JavaDoc [JDoc] or DOC++ [DOCpp], tools for managing software development and documentation are not as widespread as desirable. This paper introduces a small tool box of utilities which can be used to easily produce nicely formatted PostScript, PDF and HTML documentations for software projects with LaTeX. It is especially useful for mixed language projects and for documenting already finished programs and libraries. Due to it's sophisticated syntax highlighting capabilities (currently implemented for C/C++/Java, Scheme/Elisp and XML) it is also a good choice for writing articles or technical white papers which contain source code examples.

1 Some words on Literate Programming

This section will discuss some general aspects of literate programming and give a historical overview of the existing program documentation systems known to the author. Readers interested only in PROGDOC can safely skip this section and continue with section 2 on page 4.

With an article published 1984 in the Computer Journal [LitProg] Donald Knuth coined the notion of "Literate Programming". Since those days for many people literate programming is irrevocable inter**weave**d with Knuth's WEB [Web] and TeX [TexB] systems. And many people refuse literate programming solely because they refuse TeX or WEB.

Knuth justifies the term "literate programming" in [LitProg] with his belief that ".. the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature." To support this programming style, he introduced the WEB system which is in fact both a language and a suite of utilities. In WEB, the program source code and the documentation are written together into one source file, delimited by special control sequences. The program source can be split into arbitrary chunks which can be presented in arbitrary order. The `tangle` program extracts these code chunks from the WEB file and assembles them in the right order into a valid source file. Another program called `weave` combines the documentation parts of the WEB files with pretty printed versions of the code chunks into a file which thereupon can be processed by TeX.

This system has many advantages. First of all, it fulfills the "one source" property. Because source code and documentation reside in one file, they are always consistent with each other. Second, the programmer is free to present the code he writes in arbitrary order. Usually, he will present the code in a manner more suitable for a human reader to understand the program. This can be done by rearranging code chunks, but also by using macros inside the code chunks, which can be defined later on in the WEB file. This way a top-down development approach can be achieved, in which the the structure of a program as a whole is presented in the beginning and then subsequently refined. `tangle` will expand these macros at the right place when constructing the source file out of the WEB file.

Another feature of the WEB system is the automatic construction of exhaustive indexes and cross references by `weave`. Every code chunk is accompanied by references which link it to all other chunks which reference or use it. Also an index of keywords with respect to code chunks is created and the source code is pretty printed for the documentation part. The best way to convince yourself of WEB's capabilities is to have a look at Knuth's TEX implementation [Tex]. It was entirely written in WEB and is undoubtful a masterpiece of publishing and literate programming.

1.1 WEB and its descendants

Besides its many advantages, the WEB system also has a bunch of serious drawbacks. Many of them apply only to the original WEB implementation of Knuth and have been corrected or worked around in numerous WEB clones implemented thereafter. In this section I will present some of them[1] and discuss their enhancements.

One of the biggest disadvantages of WEB was the fact that it was so closely tied to TEX as typesetting system and to Pascal as implementation language. So one of the first flavors of WEB was CWEB [CWeb] which extended WEB to C as implementation language. It was implemented by Knuth himself together with Silvio Levy. CWEBx[CWebx] is a CWEB with some extensions by Marc van Leeuwen. CWEB suffers from the same problems like WEB, it is closely coupled to TEX and the C programming language.

To overcome this language dependencies, noWEB[noWeb] (which evolved from spiderWEB) and nuWEB[nuWeb] have been developed by Norman Ramsey and Preston Briggs respectively. They are both language independent concerning the programming language whereas they still use LATEX for typesetting. nuWEB is a rather minimalistic but fast WEB approach with only as few as four control sequences. Both noWEB and nuWEB offer no pretty printing by default but noWEB is based on a system of tools (called filters) which are connected through pipes and the current version comes with pretty printing filters for C and Java (see the actual documentation).

Another descendant of an early version of cWEB is fWEB [fWeb]. fWEB initially was an abbreviation for "Fortran WEB", but meanwhile fWEB supports not only Fortran, but C, C++, Ratfor and TEX as well. This languages can be intermixed in one project, while fWEB still supports pretty printing for the different languages. On the other hand, fWEB is a rather complex piece of software with a 140 page users manual.

Ross Williams' funnelWEB [funnelWeb] and Uwe Kreppels webWEB [webWeb] are not only independent of the programming language, but of the typesetting language

[1] I will mention here only systems which I know. If you want a more complete overview have a look at the Comprehensive TEXArchive Network (CTAN) under http://www.ctan.org/tex-archive/web or visit http://www.literateprogramming.org

as well. They define own format macros, which can be bound to arbitrary typesetting commands. As of now, they both come with HTML and LaTeX bindings respectively.

1.2 General drawbacks of WEB based literate programming tools

Though many of the initial problems of the WEB system have been solved in some of the clones, their sheer number indicates that none of them is perfect.

One of the most controversial topics in the field of literate programming is pretty printing. There are two questions here to consider. Do we want pretty printing at all, and if yes, how should the pretty printed code look like. While for the first question a rational answer can be found, the second is kind of a *np*-hard problem of computer science.

From a practical point of view it must be stated that doing pretty printing is possible for Pascal, although a look at the WEB sources will tell you that it is not an easy task to do. Doing it for C is even harder[2]. Taking into account the fact that `weave` usually processes only a small piece of code, which itself even mustn't be syntactically correct, it should be clear that pretty printing this code in a complex language like for example C++ will be impossible.

To overcome this problems, special tags have been introduced by the various systems to support the pretty printing routines. But this clutters the program code in the WEB file and even increases the problem of the documentation looking completely different than the source. This can be annoying in a develop/run/debug cycle. As a consequence, the use of pretty printing is discouraged. The only feasible solution could be a simple syntax highlighting instead of pretty printing, as it is done by many editors nowadays.

Even without pretty printing and additional tags inserted into the program source, the fact that the source code usually appears rearranged in the WEB file with respect to the generated source file makes it very hard to extend such a program. A few lines of code laying closely together in the source file may be split up to completely different places in the WEB file. Because every WEB system needs at least some control characters, they must be quoted if used inside the program code. Moreover navigating through a web file with respect to a given program structure is a hard task because of the splitting and rearrangement of functions and declarations. But changes definitively must be applied to the web file, since it is the master copy of all source files. Finally, debugging a program created from a web file resembles debugging a program compiled without debugging symbols.

Another serious problem common to WEB systems is their "one source" policy. While this may help to hold source code and documentation consistent, it breaks many other development tools like revision control systems and make utilities. Moreover, it is nearly impossible for a programmer not familiar with a special WEB system to maintain or extend code devolved with that WEB.

Even the possibility of giving away only the tangled output of a WEB is not attractive. First of all, it is usually unreadable for humans[3] and second this would break the "one source" philosophy. It seems that most of the literate programming projects realized until now have been one man projects. There is only one paper from Ramsey and Marceau[RamMarc] which documents the use of literate programming tools

[2] The biggest part of CWEB consists of the pretty printing module. Recognition of keywords, identifiers, comments, etc. is done by a hard coded shift/reduce bottom up parser

[3] nuWEB is an exception here, since it forwards source code into the tangled output without changing its format

in a team project. Additionally, some references can be found about the use of literate programming for educational purpose (see [Childs] and [ShumCook]).

The general impression confirms Van Wyk's observation[VanWyk] ".. that one must write one's own system before one can write a literate program, and that makes [him] wonder how widespread literate programming is or will ever become." The question he leaves to the reader is whether programmers are in general to individual to use somebody else's tools or if only individual programmers develop and use (their own) literate programming systems. I think the question is somewhere in between. Programmers are usually very individual and conservative concerning their programming environment. There must be superior tools available to let them switch to a new environment.

On the other hand, integrated development environments (IDEs) evolved strongly during the last years and they now offer sophisticated navigation, syntax highlighting and online help capabilities for free, thus making many of the features of a WEB system, like indexing, cross referencing and pretty printing become obsolete. Last but not least the will to write documentation in a formatting language like TEX or LATEX using a simple text editor is constantly decreasing in the presence of WYSIWYG word processors.

1.3 New program documentation system

With the widespread use of Java a new program documentation system called JavaDoc was introduced. JavaDoc [JDoc] comes with the Java development kit and is thus available for free to every Java programmer. The idea behind JavaDoc is quit different from that of WEB, though it is based on the "one source" paradigm as well. JavaDoc is a tool which extracts documentation from Java source files and produces nicely formatted HTML output. Consequently, JavaDoc is tied to Java as programming and HTML as typesetting language. By default JavaDoc parses Java source files and generates a document which contains the signatures of all public and protected classes, interfaces, methods and fields. This documentation can be further extended through specially formatted comments which may contain HTML tags.

Because JavaDoc is available only for Java, Roland Wunderling and Malte Zöckler created DOC++ [DOCpp], a tool similar to JavaDoc but for C++ as programming language. Additionally to HTML, DOC++ can create LATEX formatted documentation as well. Doxygen by Dimitri van Heesch [Doxygen], which was initially inspired by DOC++, is currently the most ambitious tool of this type which can also produce output in RTF, PDF and Unix man-page format. Both, DOC++ and Doxygen can create a variety of dependency-, call-, inclusion- and inheritance graphs, which may be included into the documentation.

These new documentation tools are mainly useful to create hierarchical, browsable HTML documentations of class libraries and APIs. They are intended for interface description rather than the description of algorithms or implementation details. Although some of them support LATEX, RTF or PDF output, they are not best suited for generating printed documentation.

2 Overview of the ProgDOC system

With this historical background in mind, PROGDOC takes a completely different approach. It releases the "one source" policy, which was so crucial for all WEB systems,

thus giving the programmer maximum freedom to arrange his source files in any desirable way. On the other hand, the consistency between source code and documentation is preserved by special handles, which are present in the source files as ordinary comments[4] and which can be referenced in the documentation. `pdweave`, PROGDOC's weave utility incorporates the desired code parts into the documentation.

But let's first of all start with an example. Suppose we have a C++ header file called `ClassDefs.h` which contains some class declarations. Subsequent you can see a verbatim copy of the file :

```
class Example1 {
private :
  int x;
public :
  explicit Example1(int i) : x(i) {}
};

class Example2 {
private :
  double y;
public :
  explicit Example2(double d) : y(d) {}
  explicit Example2(int i) : y(i) {}
  explicit Example2(long i) : y(l) {}
  explicit Example2(char c) : y((unsigned int)c) {}
};
```

It is common practice until now, especially among programmers not familiar with any literate programming tools, that system documentations contain such verbatim parts of the source code they want to explain. The problem with this approach is the code duplication which results from copying the code from the source files and pasting it into the text processing system. From now on every change in the source files has to be repeated in the documentation. This is reasonable of course, but the practice tells us that the discipline among programmers to keep their documentation and their source code up to date is not very high.

At this point, the PROGDOC system enters the scene. It allows us to write `ClassDefs.h` as follows :

```
// BEGIN Example1
class Example1 {
private :
  int x;                              // Integer variable
public :
  explicit Example1(int i) : x(i) {} // The constructor
};
// END Example1

// BEGIN Example2
class Example2 {
// ...
private :
  double y;
```

[4] As far as I know, any computer language offers comments, so this seems to be no real limitation.

```
// ...
public :
  explicit Example1(double d) : y(d) {}
  explicit Example2(int i) : y(i) {}
  explicit Example2(long i) : y(l) {}
  explicit Example2(char c) : y((unsigned int)c) {}
};
// END Example2
```

The only changes introduced so far are the comments at the beginning and at the end of each class declaration. These comments, which of course are non-effective for the source code, enable us to use the new `\sourceinput[`*options*`]{`*filename*`}{`*tagname*`}` command in the LaTeX documentation. This will results in the inclusion and syntax highlighting of the source code lines which are enclosed by the "`// BEGIN` *tagname*" and "`// END` *tagname*" lines respectively. Consequently the following LaTeX code:

```
''.. next we present the declaration of the class {\mytt Example1}:

\sourceinput[fontname=blg, fontsize=8, listing, linenr,
             label=Example1]{ClassDefs.h}{Example1}

as you can see, there is no magic at all using the {\mytt \symbol{92}sourceinput}
command ..''
```

will result in the following output:

".. next we present the declaration of the class `Example1`:

Listing 1: ClassDefs.h [Line 2 to 7]

```
class Example1 {
private :
  int x;                              // Integer variable
public :
  explicit Example1(int i) : x(i) {} // The constructor
};
```

as you can see, there is no magic at all using the `\sourceinput` command .."

First of all, we observe that the source code appears nicely highlighted, while its indentation is preserved. Second, the source code is preceded by a caption line similar to the one known from figures and tables. In addition to a running number, the caption also contains the file name and the line numbers of the included code. Furthermore this code sequence can be referenced everywhere in the text through a usual `\ref` command (like for example here: see Listing 1). Notice however that the boxes shown here are used for demonstrational purpose only and are not produced by the PROG$\mathcal{DOC}$ system.

After we got an impression of how PROG$\mathcal{DOC}$'s output looks like, it's time to explain the way how it is produced. First of all the style file '`progdoc.sty`' has to be included into the latex source file. Among some definitions and default settings (see section 9) '`progdoc.sty`' contains an empty definition of `\sourceinput`. If LaTeX will process any file with this command, it will only print out the following warning:

WARNING !!! Run pdweave on this file before processing it with LaTeX. Then you will see the sourcecode example labeled Example1 from the file ClassDefs.h instead of this message.

There are two main reasons for this behavior. The first and main one is that I'm an awful LaTeX "speaker" and thus unable to implement all this functionality in pure TeX/LaTeX. The second reason is that there already are a lot of useful tools around there in the Web, so why not combine and use them as shown in Figure 1.

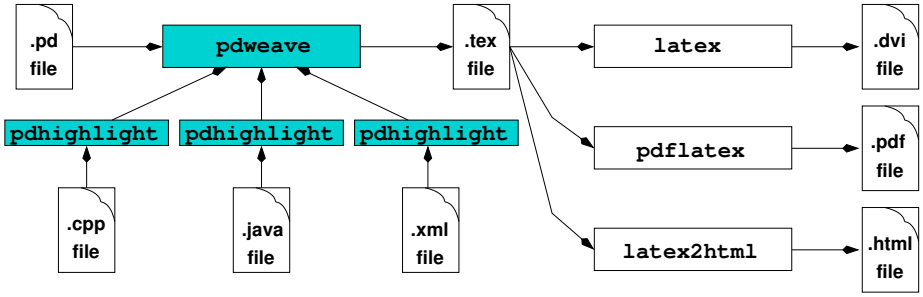

Figure 1: Overview of the ProgDOC system.

In fact the ProgDOC system consists of two parts: pdweave and pdhighlight where pdweave is an AWK script while pdhighlight is a heavily modified, extended and renamed version of Norbert Kiesel's c++2latex filter. The production of HTML is done by Nikos Drakos' and Ross Moore's latex2html [La2HT] utility.

The main idea behind ProgDOC is to write the documentation into so called '.pd' files which contain pure LaTeX code and, as an extension to ordinary LaTeX, some additional commands like the above mentioned \sourceinput. These '.pd' files are processed by pdweave which extracts the desired parts out of the source files, highlights them and finally merges them with the ordinary parts of the documentation. The file generated this way is an usual LaTeX source file which in turn can be passed to the LaTeX text processor.

Usually, all this steps are simplified by the use of a special Makefile which also keeps track of dependencies between source files and the documentation itself (see section 10 for an example).

In the next sections a brief description of the different commands available in '.pd' files will be given. The format of the handles required in the source files will be explained and finally an example Makefile which automates the generation of the program documentation will be presented.

3 The \sourceinput command

Now that we have an idea of the general mechanism of the ProgDOC system, let's have a closer look on the \sourceinput command. Its syntax is similar to that of other LaTeX commands though, as we know by now, it will be normally processed by pdweave and not by LaTeX. The general form of the command is:

\sourceinput[*options*]{*filename*}{*tagname*}

Like in LaTeX, arguments in {}-brackets are required whereas the ones in []-brackets are optional.

\sourceinput Arguments	
filename	Absolute or relative pathname of the source file. This may be internally preceded by a base path if the command \sourceinputbase{*filename*} (see section 7) has been used.
tagname	This is an arbitrary string which uniquely identifies a part of source code in the file specified by *filename*. A special *tagname* "ALL" is available, which includes a whole file. (See section 8 for a detailed description of the comment format in the source files).
\sourceinput Options	
label=name	An ordinary LaTeX label name which will be declared inside of the produced source listing and which can be used subsequently as parameter for the \ref command.
fontname=name	The name of the base font used for highlighting the source listing. It is desirable here to specify a mono spaced font of which italic, bold and bold italic versions exist, since they are used to emphasize keywords, comments, string constants and so on[5]. (The default is the initial setting for \ttdefault, usually cmtt)
fontenc=encoding	The encoding of the font chosen with the *fontname* option above. (The default is OT1.)
fontsize=pt	The fontsize in point used for highlighting the listings. Since mono spaced fonts are usually some wider compared to proportional counterparts, a somewhat smaller size is recommended here. (The default is 8pt.)
linesep=length	The line separation used for the source listings. (The default is 2.5ex.)
type=language	This option controls the type of language assumed for the source file. The *language* argument will be handed over to the actual highlighter (see the option *highlighter*). Currently the default highlighter pdhighlight supports the values *c*, *cpp*, *java*, *xml*, *scm*, *el* or *text*. If not set, the default language is *cpp*. If type is set to *text* no syntax highlighting will be done at all. Notice that this option also affects the way in which comments are recognized in the source files (see also the option *comment* and chapter 8 about the source file format 8 on page 12).
comment='string'	If you use one of the supported languages listed in the table on page 12, the tag names will be recognized automatically. If you however include parts of a file in an unsupported language, it may be necessary to set the string which denotes the beginning a comment in that language with this option.
tab=value	The value of *tab* indicates the number of space characters used to replace a tab character ('\t'). The dafault is 8.

[5] For more information on choosing the right base font see section B.7 on page 23

\sourceinput Options	
listing[=noUnderline]	If the *listing* option is present, a heading will be printed above the listing, which contains at least the running number of the listing and the name of the file it was extracted from. By default, this heading will be underlined. You can change this behavior by using the optional noUnderline argument
linenr	If the *linenr* option is set, the heading additionally will contain the line numbers of the code fragment in its source file. The special *tagname* "ALL" always turns line numbers off.
center	With this option set, the listing will appear centered, without it will be left justified.
underline	If this option is set, pdhighlight will underline keywords instead of setting them in bold face. This is useful for fonts for which there exists no bold version (e.g. cmtt).
caption='captiontext'	If this option is set, then the caption produced by the *listing* option will contain *captiontext* instead of the file name and possibly the line numbers. Notice that *captiontext* must be enclosed between apostrophe signs " ' ".
wrap=column	With this option, you can instruct pdweave to wrap the lines of the source code you include at the specified *column*. pdweave uses a heuristics in order to find a "good" break position, so the column argument supplied with *column* is just a maximum value which will be not exceeded. Lines broken by pdweave, will be marked by an arrow ("↩") at the breaking point. This option is especially useful in two-column mode. For en example see the Listings 10 to 14 on page 24.
highlighter=program	This option controls which program is used to highlight the source code. The default highlighter is pdhighlight. Currently the only additional highlighter is pdlsthighlight. Refer to section 5 for further information.
useLongtable DEPRECATED	This is a compatibility option which forces the default highlighter pdhighlight to arrange the source listings in a longtable environment. Because of layout problems which resulted from the interaction of longtables with other float objects, the use of the longtable environment has been abandoned. This option is only for people who want to typeset a document in exactly the same way it was done with older versions of PROGDOC.

Apart from whitespace, the \sourceinput command must be the first to appear in a line and it must end in a line of its own. However the command itself can be split over up to five different lines. (This number can be adjusted by setting the variable DELTA in the script pdweave.awk.). It may also be necessary to quote some option arguments between apostrophe signs " ' ", if they contain white space or special characters like angle or curly brackets.

Some of this options like *fontname* or *fontsize* can be redefined globally in the '.pd' file. See section 9 on page 15 for more information.

4 Using ProgDOC in two-column mode

Starting with version 1.3, PROGDOC can be used in the LaTeX two-column or multicolumn mode. However some restrictions apply in these modes which will be discussed here. We will switch now to two-column mode by using the multicols environment with the command `\begin{multicols}{2}`:

First of all, there is no two-column support when using the deprecated *useLongtable* option, because the longtable environment doesn't work in the two-column mode.

Otherwise, the two-column mode set with the `twocolumn` option of the `documentclass` command or inside the document with the `\twocolumn` command is supported as well as the two- or multicolumn mode of the `multicols` environment (see [multicol]), however with some minor differences.

Listing 2: A short Python example

```
#
# QuickSort and Greatest Common Divisor
# Author: Michael Neumann
#

<see Listing 3 on page 10>

<see Listing 4 on page 11>

print "Hello World"
print quicksort([5,99,2,45,12,234,29,0])
```

Because of incompatibilities between the `multicols` environment and the `afterpage` package, the caption "**Listing** x: ... (continued)" on subsequent columns or pages is not supported for listings inside the `multicols` environment (as can be seen in Listing 2 to 4 which are printed inside a `multicols` environment). If in `twocolumn` mode, columns are treated like pages for the caption mechanism of PROGDOC (see section C for an example printed in `twocolumn` mode). Therefore the "**Listing** x: ... (continued)" captions are repeated on the top of each new column the listings spans on, just as if it was a new page.

5 Using the alternative highlighter `pdlsthighlight`

In addition to the default highlighter `pdhighlight` PROGDOC comes now with an additional highlighter called `pdlsthighlight` which is in fact a wrapper for the `listings` environment of Carsten Heinz (see [listings]).

Listing 3: test.py [Line 8 to 12]
(Referenced in Listing 2 on page 10)

```
def ggd(a, b):
  if a < b: a,b = b,a
  while a%b != 0:
    a,b = b,a%b
  return b
```

To use this highlighter the `listings.sty` package has to be installed and manually loaded into the document with `\usepackage{listings}`. The Listings 2 to 4 are typeset using `pdlsthighlight` with the following options: `[linenr, listing, wrap=40, fontname=blg, highlighter='pdlsthighlight', type=Python]`.

`pdlsthighlight` also works in both, single and two-column mode, however it doesn't support the "**Listing** x: ... (continued)" captions at all. The benefits of the new highlighter are the many supported language for which the `listings` package performs syntax highlighting. One of the main drawbacks is the fact that you can not produce an HTML version of the document because LaTeX2HTML doesn't support the package.

Notice furthermore that you have to set the *type* option of the \sourceinput command to a value recognized by the listings environment if you use pdlsthighlight as highlighter (e.g. *type=C++* instead of *type=cpp*). Refer to [listings] for a complete list of supported languages.

Listing 4: test.py [Line 16 to 21]
(Referenced in Listing 2 on page 10)

```
def quicksort(arr):
  if len(arr) <= 1: return arr
  m = arr[0]
  return quicksort(filter(lambda i,j=m: i<j, arr)) + \
         filter(lambda i,j=m: i==j, arr) + \
         quicksort(filter(lambda i,j=m: i>j, arr))
```

In this context it may also be necessary to use the *comment* option to specify the comment characters of a language not known to pdweave.

6 The \sourcebegin and \sourceend commands

Beneath the \sourceinput command there exists another pair of commands, which can be used to highlight source code written directly into the '.pd' file. Of course they are pseudo LaTeX commands as well and will be processed by the pdweave utility rather than by LaTeX. Their syntax is as follows:

\sourcebegin[*options*]{ *header*}
source code
\sourceend

The \sourcebegin command has the same options like the \sourceinput command, but no *filename* and *tagname* options, since the source code begins in the line that follows the command. For compatibility reasons with older ProgDOC versions there is an optional *header* argument. It will be printed instead of the filename in the header of the listing if the option *listing* is set. The recommendation for new users however is to use the *caption* option instead. Notice that in contrast to the usual LaTeX conventions, this is an optional argument. The source code will be terminated by a line which solely contains the \sourceend command.

This commands are useful if some code must be presented in the documentation which is not intended to appear in the real source code. Consider for example the following code:

```
.. we don't use void pointers and ellipsis for our function {\mytt func}

\sourcebegin[fontname=blg, fontsize=8, listing, center]{Just an example ..}
void func(void *p, ...) {
  cout << "A function with an arbitrary number of arguments\n";
  ..
}
\sourceend

since they are bad programming style and can lead to unpredictable errors ..
```

which will result in the following output:

".. we don't use void pointers and ellipsis for our function `func`

Listing 5: Just an example ..

```
void func(void *p, ...) {
  cout << "A function with an arbitrary number of arguments\n";
  ..
}
```

since they are bad programming style and can lead to unpredictable errors .."

The same restrictions that apply for the `\sourceinput` command hold good for `\sourcebegin` and `\sourceend` as well. Additionally, if present, the opening brace of the optional *header* argument must start in the same line like the closing bracket of the *options* argument.

7 The \sourceinputbase command

If you want to present to the reader a certain view of the source code, relative and absolute path names may be not enough for the `\sourceinput` command. In this case you can use the command:

\sourceinputbase{*pathname*}

It defines a global path prefix for all `\sourceinput` commands which follow in the same file. You can reset this path prefix by calling `\sourceinputbase{}` with a zero length argument. Like the `\sourceinput` command, the `\sourceinputbase` command must be in its own line and may be preceded only by whitespace. This command has file scope.

Notice that automatic references between nested code sequences (see section 8.2) will work only if the code sequences have been included with the same path prefix. This is because of the algorithm which automatically generates the labels for nested code sequences. It uses the pathname of the file from which a code sequence has been included as a part of the generated label name.

8 The source file format

As shown in the first section, arbitrary parts of a source file can be made available to PROG$\mathcal{DOC}$ by enclosing them with comment lines of the form '// `BEGIN` *tagname*' and '// `END` *tagname*' respectively where in this and the following examples we will use the C++ comment syntax. However PROG$\mathcal{DOC}$ also supports a number of other languages.

When speaking about supported languages, one has to distinguish between highlighting support for a language which comes from `pdhighlight` and the support to extract code snippets out of files of a given language, which is provided by `pdweave`. The following table lists the supported languages with respect to both these tools. In general, any file may be used as input source, even if not listed here, by specifying "text" as *type* argument and the corresponding comment character(s) as *comment* argument to the `\sourceinput` command (see table on page 8).

type	Language	Comment character(s)	`pdweave`	`pdhighlight`
c	C	// , /∗	√	√

type	Language	Comment character(s)	pdweave	pdhighlight
cpp	C++	// , /∗	√	√
java	Java	// , /∗	√	√
xml	XML	<! − −	√	√
scm	Scheme	; , ;; , ;;; , ;;;;	√	√
el	ELisp	; , ;; , ;;; , ;;;;	√	√
vb	VisualBasic	’	√	−
py	Python	#	√	−
text	Text	# , // , -	√	−

8.1 Hiding code parts

An arbitrary even number of ’// ...’ comments may appear inside a ’BEGIN/END’ code block. All the code between two of these comment lines will be skipped in the output and replaced by a single “dotted line” (...). This is useful for example, if you want to show the source code of a class, but don’t want to bother the reader with all the private class stuff.

Recall the header file from section 2, which will be reprinted here for convenience, by using the following command: “\sourceinput[fontname=blg, fontsize=8, listing]{ClassDefs.h}{ALL} ”. Notice the use of the special tag name “ALL”, which includes a source file as a whole.

Listing 6: ClassDefs.h

```
// BEGIN Example1
class Example1 {
private :
  int x;                              // Integer variable
public :
  explicit Example1(int i) : x(i) {} // The constructor
};
// END Example1

// BEGIN Example2
class Example2 {
// ...
private :
  double y;
// ...
public :
  // BEGIN Constructors
  explicit Example2(double d) : y(d) {}
  explicit Example2(int i) : y(i) {}
  explicit Example2(long l) : y(l) {}
  explicit Example2(char c) : y((unsigned int)c) {}
  // END Constructors
  void doSomething(); // do something
};
// END Example2
```

In the way described until now we can include the class definition of the class "Example2" by issuing the command: "\sourceinput[fontname=ul9, fontenc=T1, fontsize=7, listing, linenr, label=Example2]{ClassDefs.h}{Example2}".

Listing 7: ClassDefs.h [Line 11 to 24]

```
class Example2 {
...
public :
  <see Listing 8 on page 14>
  void doSomething(); // do something
};
```

As you can see however, the private part of the class definition is replaced by the mentioned "dotted line" which stands for as much as "there is some hidden code at this position in the file, but this code is not important in the actual context".

8.2 Displaying nested code sequences

Another possibility of hiding code at a specific level, is to nest several "BEGIN/END" blocks. If a "BEGIN/END" block appears inside another block, then he will be replaced by a single line of the form "*<see Listing xxx on page yyy>*". *xxx* denotes the listing number in which the code of the nested block actually appears and *yyy* the page number on which that listing begins. Of course this is only possible, if the mentioned nested block will be or already has been included by a \sourceinput command.

In turn, if a nested block will be included through a \sourceinput command, his heading line will additionally contain the listing and page number of his enclosing block. You can see this behavior in the following example where we show the constructors of the class Example2 by issuing the following command: "\sourceinput[fontname=ul9, fontenc=T1, fontsize=7, listing, linenr, label=Constructors]{ClassDefs.h}{Constructors}".

Listing 8: ClassDefs.h [Line 18 to 21] (Referenced in Listing 7 on page 14)

```
explicit Example2(double d) : y(d) {}
explicit Example2(int i) : y(i) {}
explicit Example2(long l) : y(l) {}
explicit Example2(char c) : y((unsigned int)c) {}
```

So lets finally state more precisely the difference between hiding code through '// ...' comment lines and the nesting of code blocks. While '// ...' comments always match the following '// ...' line, a nested 'BEGIN *tagname*' always matches its correspondent 'END *tagname*' and can potentially contain many '// ...' lines or even other nested chunks. Another difference is the fact that nested chunks can be presented later on in the documentation and will be linked together by references in that case , while parts masked out by '// ...' lines will simply be ignored. Nevertheless, '// ...' lines can be useful for example if a part of a source file contains many lines of comments which aren't intended to be shown in the ProgDOC documentation. If you want to use nested "BEGIN/END" chunks together with the \sourceinputbase command, be sure to read the comments on this topic in section 7.

One last word on the format of the comments processed by the ProgDOC system. They must be in a line on their own. The comment token, `BEGIN/END` and the tagname must be separated by and only by whitespace. The comment token must not necessarily begin in the first column of the line as long as it is preceded only by whitespace. The tagname should consist only of characters which are valid in a LaTeX `\label` statement.

9 LaTeX customization of ProgDOC

Some of the options available for the '`\sourcebegin`' and the '`\sourceinput`' command (see section 3 on page 7) can be set globally by redefining LaTeX commands. Additional commands can be used to adjust the appearance of the generated output even further. Following a list of the available commands:

`\pdFontSize`	The font size used for printing source listings. The default is 8pt. This command is the global counterpart of the *fontsize* option of '`\sourcebegin`' and '`\sourceinput`'.
`\pdLineSep`	The line separation used for printing source listings. The default is 2.5ex. This command is the global counterpart of the *linesep* option of '`\sourcebegin`' and '`\sourceinput`'.
`\pdBaseFont`	The font family which is used to print source listings. The default is '`\ttdefault`'.This command is the global counterpart of the *fontname* option of '`\sourcebegin`' and '`\sourceinput`'.
`\pdFontEnc`	The encoding of the font family chosen with `\pdBaseFont` or with the *fontname* option of the '`\sourcebegin`' or '`\sourceinput`' commands. The default is OT1. This command is the global counterpart of the *fontenc* option of '`\sourcebegin`' and '`\sourceinput`'.
`\pdCommentFont`	The font shape used for highlighting comments in the source listing. The default setting is '`\itshape`'.
`\pdKeywordFont`	The font shape used to highlight the key words of a programming language. The default is '`\bfseries`'.
`\pdPreprFont`	The font shape used to highlight preprocessor commands in C or C++. The default is '`\bfseries\itshape`'.
`\pdStringFont`	The font used to highlight string constants in source listings. The default setting is '`\slshape`'.
`\ProgDoc`	Command to print the ProgDOC logo.
`\pdULdepth`	This is a length command which controls the depth of the line under a listing caption. ProgDOC uses the `ulem.sty` package for underlining which does a pretty good job in guessing a reasonable value for this purpose. However it may sometimes be necessary to manually fine tune it, depending on the used font. The length may be set with the `\setlength` command. Resetting `\pdULdepth` to 0pt reactivates the initial `ulem.sty` algorithm. (This tutorial for example uses `\setlength{\pdULdepth}{2.5pt}`.)

\pdPre[6] DEPRECATED	This and the following three length commands correspond to the longtable commands \LTpre, \LTpost, \LTleft and \LTright respectively. For more information see the documentation of the longtable package [longtable]. \pdPre sets the amount of space before a listing. The default is \bigskipamount.
\pdPost[6] DEPRECATED	\pdPost sets the amount of space after a listing. The default is 0cm.
\pdRight[6] DEPRECATED	The margin at the right side of the listing. The default is \fill.
\pdLeft[6] DEPRECATED	\pdLeft sets the amount of space at the left side of a listing. Usually the listing is left justified or centered (see also section 3, The \sourceinput command). But because listings are typeset inside a longtable environment, they aren't indented for example inside list environments. In that case it can be useful to set \pdLeft to \leftmargin. If the listing will be insight a nested list environment, you can use \renewcommand{\pdLeft}{x\leftmargin} where x is the nesting level. The default is 0cm.

All these commands can be redefined. If you want to typeset string constants in italic, you could insert the following line in the preamble of your '.pd' file: '\renewcommand{\pdStringFont}{\slshape}'.

The words used to built up the header of each listing can be set by the user according to his preferences (though this is intended mainly to permit a certain kind of localization). They are defined in 'progdoc.sty' as follows:

\ListingName	The name used to name listings. The default is "Listing".
\LineName	The name of a line. The default setting is "Line".
\toName	The word for "to" in "Line xxx to yyy". Defaults to "to".
\ReferenceName	The sentence "Referenced in".
\PageName	The words "on page".
\ListingContinue	A word to indicate that the current listing is a continuation from a previous page. Defaults to "continued".
\NextPage[6] DEPRECATED	This should be a small symbol to indicate that a listing is not finished, but will be continued on the next page. The default setting is '\ding{229}' which is the '➥' symbol.

You could customize these entries for the german language by inserting the following lines into the preamble of your '.pd' file:

```
\def\LineName{Zeile}
\def\toName{bis}
\def\ReferenceName{Referenziert in}
\def\PageName{auf Seite}
\def\ListingContinue{Fortsetzung}
```

[6] Because ProgDOC internally used the longtable environment in older versions to render the program listing, some of the longtable options have been made available to ProgDOC users. As new versions of ProgDOC don't use longtable anymore, this options have no effect. (See the *useLongtable* option of the \sourceinput command on page 1 for a compatibility option to enable the old style mode which uses the longtable environment).

10 An example Makefile

In this chapter a makefile will be presented which simplifies the task of calling all the scripts in the right order and keeps track of dependencies between source and documentation files. For the sake of simplicity, the makefile used to build this documentation will be shown:

Listing 9: Makefile

```
dvi   : tutorial.dvi
ps    : tutorial.ps
pdf   : tutorial.pdf
html  : tutorial/tutorial.html
out   : example
clean :
        rm -rf *.dvi *.ps *.pdf *.log *.aux *.idx *~ part1.tex tutorial.tex \
               *pk *.out _pdweave.tmp _pd_html.html tutorial

tutorial.dvi : tutorial.tex part1.tex

tutorial.pdf : tutorial.tex part1.tex progdoc.pdf

progdoc.pdf : progdoc.eps
        epstopdf progdoc.eps

part1.tex    :  ClassDefs.h test.xml test.py version.el

example : example.cpp ClassDefs.h
        g++ -o example example.cpp

tutorial/tutorial.html: tutorial.dvi
        latex2html -html_version 4.0 -show_section_numbers -image_type gif \
                -up_title "ProgDoc Home Page" -up_url "../progdoc.htm"  \
                -no_footnode -local_icons -numbered_footnotes tutorial.tex

# We generate ps from pdf now in order to depend only on pdfLaTeX!
# %.ps  : %.dvi
#        dvips -D 600 -o $@ $<

%.ps  : %.pdf
        acroread -toPostScript -binary $<

%.dvi : %.tex
        latex $< && latex $<

%.pdf : %.tex
        rm -f $*.aux && pdflatex $< && pdflatex $<

%.tex : %.pd
```

Listing 9: Makefile (continued)

```
	pdweave $<
```

Of course this file can be included with the `\sourceinput` command as well. Because syntax highlighting for makefiles is not supported yet, the file was included by using the *type* option set to *text*. But even in this case, there are still benefits in using the `\sourceinput` command. First of all, the documentation will always contain the actual makefile. Second, this makefile can be referenced throughout the documentation like every other source file (see Listing 9). And last but not least, ProgDOC may be extended in the future to highlight various other file formats, so you may improve your documentation by simply rebuilding it with a new version of ProgDOC.

Now lets have a closer look on the makefile. The first five lines define shortcuts for the different targets, namely the `dvi`, `ps`, `pdf` and `html` versions of the documentation and the `example` executable. `clean`, the last target removes all files created during a build process. Notice that '`_pdweave.tmp`' and '`_pd_html.html` are temporary files created by `pdweave`.

In the next lines, the dependencies are defined. The `dvi` output depends on the `tex` files of the documentation which in turn depend on the source code of the files they document. Therefore the documentation will be rebuild not only if the documentation source files will change, but also if the source code files change.

The next two rules tell make utility how to build the `example` executable and the html version of the documentation. The latter will be created by LaTeX2HTMLin its own subdirectory.

The last four parts of the makefile contain generic actions which tell the make utility how to generate '`.ps`' files out of '`.dvi`' files, '`.dvi`' files out of '`.tex`', '`.pdf`' files out of '`.tex`' files and finally '`.tex`' files out of `.pd`-files. As you can see, for the last step the `pdweave` utility will be used.

Using this example as skeleton, it should be straightforward how to write makefiles for your own projects.

11 Acknowledgements

I want to thank all the users who used ProgDOC and supplied feedback information to me. Among others these are Martin Gasbichler, Blair Hall and Roland Weiss. Finally I'm truly indebted to Holger Gast, who always answered patiently all my questions and solved most of my problems concerning TeX which beneath all its strengths is a terrible programming language.

A Installing ProgDOC

To install ProgDOC get the latest version. Currently there are binary versions available for Linux (ProgDoc_x.y_linux.tgz) Windows (ProgDoc_x.y_win32.tgz) and various other Unix versions (have a look at the ProgDOC home page at [progDoc]). If you use another system you must get the source distribution ProgDoc_x.y_src.tgz where (x and y denote the major and minor version number) and compile it yourself.

A.1 Requirements

In order to use the ProgDOC system you need a running awk interpreter. If you want to produce HTML output, you need LaTeX2HTML. Additionally, a make utility is recommended but not needed. If you want to compile the source code, you need flex, the GNU version of the lexical analyzer generator lex, an ANSI C compiler and make.

The binary version for Windows as well as the source distribution contain win32 executables of awk, flex and make, whereas Unix systems have them by default. The latest version of LaTeX2HTML can be obtained from [La2HT]. ProgDOC 1.0 was tested with LaTeX2HTML 99.2beta5 and beta6. Older versions may have some problems processing ProgDOC's output. Therefore using the LaTeX2HTML version which can be found on the ProgDOC home page is recommended.

A.2 Compiling ProgDOC

If you have a binary distribution, you can skip this section. Unpack the source distribution. If you work on a Windows systems be sure to copy the files located in the 'win32bin/' directory of the distribution to a location covered by your PATH environment variable. Then enter the 'src/' directory and type make. This should call flex which creates pdhighlight.c out of pdhighlight.l. Then the C/C++ compiler(s) will compile all the C/C++-files and produce the executables pdhighlight and pdlsthighlight in the 'bin/' directory. On Windows platforms you must execute 'make -f Makefile.win' in the 'src/' directory which will result in the creation of pdhighlight.exe and pdlsthighlight.exe in the 'bin/' directory.

Notice: pdhighlight uses the GNU getopt library to parse its command line arguments. Because this library is not standard on all Unix systems, ProgDOC comes with its own getopt files. However, in rare circumstances (apparently under some versions of AIX or BSD), these files can interfere with already installed getopt files. In this case, you can either delete the getopt files which come with ProgDOC and use the ones supplied by the system, or if this doesn't work, install a new version of getopt.

A.3 Installing ProgDOC

Copy pdhighlight, pdlsthighlight and pdweave (pdhighlight.exe, pdlsthighlight.exe, pdweave and pdweave.bat on Windows systems) from the 'bin/' directory to a directory which is in the executable path of your system (for example '/usr/local/bin' on Unix systems or 'C:\WINNT\SYSTEM32' on Windows boxes.) On Windows systems you have to additionally edit the pathname of pdweave in the file pdweave.bat to reflect the actual location of the file.

As last step you have to copy the style files located in the directory 'latex/' into a place where they can be found by your LaTeX system. The 'latex/' directory currently contains two files: 'progdoc.sty' and 'html.sty'. 'html.sty' is only needed in the case

you don't plan to install LaTeX2HTML. Otherwise use the 'html.sty' file which comes with your LaTeX2HTML distribution.

That's it, you're done. In order to test your installation, you can enter the 'test/' directory and type make at the command prompt. This should build the file test.dvi. If you have LaTeX2HTML up and running (see section A.4), you can try to say make html in order to build the HTML version of the documentation. The page will be generated in its own subdirectory called 'test/'.

A.4 Notes for LaTeX2HTML users

If you want to produce a HTML version of your documentation with LaTeX2HTML you have to copy the file '.latex2html-init' from the 'latex2html/' directory into your home directory. If you already have your own '.latex2html-init' just insert the marked lines of the provided file into yours. Especially pay attention that the $TEXINPUTS variable holds the paths to which you copied the ProgDOC style file 'progdoc.sty'.

B Known Problems

In this section solutions for some common problems which can occur during the daily work with ProgDOC are presented. If this doesn't solve your problem or even better, if you already found a solution for a problem not mentioned here, please contact the author.

B.1 Using pdfLaTeX

ProgDOC works fine with pdfLaTeX. However there is a problem if using LaTeX and pdfLaTeX alternately. This is due to an incompatibility in the format of the auxiliary file `<filename>.aux`, which is used by both, LaTeX and pdfLaTeX to propagate information from one program run to the next.

Thus executing `pdflatex` after an execution of `latex` will usually result in a bunch of confusing error messages. The solution of the problem is quite simple: remove the LaTeX generated `.aux` file every time before executing `pdflatex`. (This can be automated for example in a makefile.) Notice that the procedure just described is not necessary if executing `latex` after `pdflatex`.

Another problem with pdfLaTeX is the fact that you get mangled names in the table of contents generated by pdfLaTeX in the left `Acrobat Reader` window, if the section titles contain not pure text, but additionally LaTeX commands. This problem has nothing to do at all with ProgDOC, but it is very annoying, especially for beginners. That's why the file progdoc.sty contains a redefinition of the pdfLaTeX macro `\texorpdfstring{<`*TeX-String*`>}{<`*PDF-String*`>}` which is known to pdfLaTeX but not to LaTeX. The macro is a conditional statement for strings. If you use LaTeX, the macro evaluates to <*TeX-String*>. If you use pdfLaTeX the macro evaluates to <*PDF-String*> in the PDF table of contents only, but to <*TeX-String*> in any other case. Thus you can write documents which you can process with LaTeX as well as with pdfLaTeX.

B.2 Using `hyperref.sty`

Because the `hyperref` package interacts with many other packages in a subtle and and often unpredictable way it is a constant source of confusion. In order to minimize these problems it is strongly recommended to use a reasonable new version of `hyperref.sty`. The other important point is to include `hyperref.sty` "at the right time". For the ProgDOC tutorial, which is processed by LaTeX, pdfLaTeX and 2html the following order of inclusion led to acceptable results:

```
\usepackage{fancyhdr}
\usepackage{hyperref}
\usepackage{html}
\usepackage{progdoc}
\usepackage{listings}
```

If you use any other sensitive packages you may figure out the right position for the inclusion of your package by trying to load it either before or just after `hyperref.sty`.

B.3 Iconic instead of numeric references in the HTML version of the documentation

In the HTML version of the documentation generated by LaTeX2HTML there are no numeric references (for example in a reference like: see section 5). Instead, the reference is displayed as a small icon. Although not obvious, this problem is closely related with the previous one. The reason why this can happen is the fact that LaTeX2HTML doesn't understand the `.aux` file generated by pdfLaTeX. So in order to get the reference numbers right execute LaTeX before you run LaTeX2HTML.

Notice however that references to pages will always be displayed as icons, because in the HTML version of the document, line numbers simply don't exist. If you want to solve this problem, you have to use the conditional statements of LaTeX2HTML. See section 4.5 in the LaTeX2HTML manual [La2HT].

B.4 Widow and club lines in listings

- DEPRECATED - (Should only happen with the *useLongtable* option.)

Sometimes it can happen that a listing starts at the end of a page only with the listing header, but without a single line of the listing. Or it may happen that a listings ends at the beginning of a page with nothing else but a listing header. These phenomena are called 'widow line' and 'club line' in typography.

There are two things to say about them. First of all, there is no general solution available, but second, there is a workaround which helps in most of the cases. The problem is not native to PROG*DOC*, but a problem of the tables generated by the `longtable` package. If you recall, the PROG*DOC* system uses the `longtable` environment to lay out the listings.

The workaround promised above is the following. Don't care to much about widow and club lines, until you really finished to write your document. If you finished the last version and the problem still persists, insert an `\enlargethispage{3pt}` command just before or just after the corresponding `\sourcebegin` environment. Use a positive or negative length as appropriate. Usually a length of some points should be enough to avoid the problem.

B.5 Errors during the execution of `pdweave`

This is most likely a problem of your `awk` interpreter. `pdweave` uses `awk` features which have been added by the authors in 1985 in a revised version of their program, which they called `nawk`. Since then, most Unix Systems contain the `nawk` interpreter under the name `awk`. But if `pdweave` doesn't work with the `awk` on your system, then you can try to replace the first line of the `pdweave` script with '`#!/usr/bin/nawk -f`', if you have the `nawk` interpreter. If this still doesn't work, you can get the GNU version of `awk` which exists for almost all platforms [gawk]. It is sometimes installed under the name `gawk`.

B.6 Using `extramarks.sty`

PROG*DOC* should work together with the `extramarks.sty` package written by Piet van Oostrum (see [fancyhdr]). However because `progdoc.sty` redefines some of the macros in `extramarks.sty`, `progdoc.sty` should be included after `extramarks.sty` was included with the `\usepackage` command.

B.7 Chosing the right font for source listings

In order to achive a plaesing source code presentation it is essential to chose the right font for the source code formatting. Computer programs are traditionally edited by using mono spaced fonts. Therefore it is desirable to use an monospaced font for printing as well, in order to conserve the visual layout which the source code had on the screen.

Because PROGDOC highlights certain parts of a listing by printing them with different faces, a monospaced font which comes with italic, bold and bold italic versions is recommanded. In this document the `Letter Gothic 12 Pitch`[7] font family from Bitstream is used for source listings because it is narrower then `Courier`, the standard Postscript monospaced font family.

Unfortunately, for the computer modern typewriter family there are no bold and bold italic series available by default. There exists a bold version of the computer modern typewriter font (see [cmbtt]), but probably you will have to install it yourself.

Recently, with the release of Version 4.2 of XFree86, the free implementation of the X Windows System, the new, freely distributable monopsaced font family `LuxiMono` became available. It is a Type 1 encoded Postscript font which comes with bold, italic and bold italic versions. It can be downloaded from [LuxiMono].

[7] `Letter Gothic 12 Pitch` is a commercial font, but available on many CD ROMs bundled with printers or DTP systems (for example CorelDraw).

C Highlighting XML source code[8]

Highlighting XML code is straight forward. The only thing you have to do is setting the *type* option of the '\sourcebegin' or '\sourceinput' commands to *xml*. Therefore this section is merely a demonstration of how a '.xml' file will look like.

Listing 10: test.xml

```
<?xml version="1.0"?>
<?xgilf time-stamp="2F3A"?>

<!ENTITY test "and here we go">
<!ATTLIST xgilf lib type #REQUIRED>

<!-- a comment -->

<xgilf>
  <!-- BEGIN RolEdd -->
  <name>Roland &lt; {} test</name>
  <name>Edda, Ms. &#8364;</name>
  <!-- END RolEdd -->
  <!-- another comment -->
  <empty att="1234" />
</xgilf>
```

Listing 11: test.xml [Line 11 to 12]

```
<name>Roland &lt; {} test</name>
<name>Edda, Ms. &#8364;</name>
```

D Highlighting Scheme source code

Highlighting Scheme code is straight forward. The only thing you have to do is setting the *type* option of the '\sourcebegin' and '\sourceinput' commands to *scm*. Therefore this section is merely a demonstration of how a '.scm' file will look like. Notice however the comment style used in the Scheme source files. In order to be recognized by ProgDOC, comments can begin with one to four semicolons.

Listing 12: doc/example.scm

```
;; BEGIN Factorial
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1))))))
;; END Factorial

(string=? "K. Harper, M.D." ;; Taken from ↩
       Section 6.3.3. (Symbols) of the R5RS
          (symbol->string
           (string->symbol "K. Harper, ↩
                                   M.D.")))

;; BEGIN Square
(define square
  (lambda (n)  ;; My first lambda
    (if (= n 0)
        0
;; BEGIN Recursive_Call
        (+ (square (- n 1))
           (- (+ n n) 1)))))
;; END Recursive_Call
;; END Square
```

Listing 12: doc/example.scm (continued)

Notice that we used the command \renewcommand{ \pdCommentFont}{\bfseries\itshape} in order to set the comment font of the listings beginning with Listing 12 to bold italic.

Listing 13: doc/example.scm [Line 14 to 21]

```
(define square
  (lambda (n)  ;; My first lambda
    (if (= n 0)
        0
        <see Listing 14 on page 24>
```

Listing 14: doc/example.scm [Line 19 to 20] (Referenced in Listing 13 on page 24)

```
(+ (square (- n 1))
   (- (+ n n) 1)))))
```

[8] This and the following section not only demonstrate how to highlight XML and Scheme source code, but also show how ProgDOC can be used in two-column mode. The only difference between listings in the two-column mode set with the twocolumn option of the documentclass command or inside the document with the \twocolumn command and listings in the multicols environment is the behavior of the listing caption. Because of incompatibilities between the multicols environment and the afterpage package, the caption "**Listing** x: ... (continued)" on subsequent columns or pages is not supported for listings inside the multicols environment. If in the twocolumn mode, columns are treated like pages for the caption mechanism of ProgDOC. Thus the "**Listing** x: ... (continued)" captions are repeated on the top of each new column on which the listing spans, just as if it was a new page.

E Highlighting ELisp source code

Highlighting ELisp code is similar to highlighting Scheme code. You have to set the *type* option of the '`\sourcebegin`' or '`\sourceinput`' commands to *el*. For a comment in order to be recognized as valid label, the same rules apply as stated for Scheme code. Following we show how a highlighted '`.el`' file will look like. (The base font used for the listing is `Courier`. It was specified with the option *"fontname=pcr"*.)

Listing 15: version.el

```
;;; version.el --- record version number of Emacs.
;;; Copyright (C) 1985, 1992, 1994, 1995 Free Software Foundation, Inc.

(defconst emacs-version "20.5" "\
Version numbers of this version of Emacs.")

(defconst emacs-major-version
  (progn (string-match "^[0-9]+" emacs-version)
         (string-to-int (match-string 0 emacs-version)))
  "Major version number of this version of Emacs.
This variable first existed in version 19.23.")

(defconst emacs-minor-version
  (progn (string-match "^[0-9]+\\.\\([0-9]+\\)" emacs-version)
         (string-to-int (match-string 1 emacs-version)))
  "Minor version number of this version of Emacs.
This variable first existed in version 19.23.")

(defconst emacs-build-time (current-time) "\
Time at which Emacs was dumped out.")

(defconst emacs-build-system (system-name))

(defun emacs-version  (&optional here) "\
Return string describing the version of Emacs that is running.
If optional argument HERE is non-nil, insert string at point.
Don't use this function in programs to choose actions according
to the system configuration; look at 'system-configuration' instead."
  (interactive "P")
  (let ((version-string
         (format (if (not (interactive-p))
                     "GNU Emacs %s (%s%s)\n of %s on %s"
                   "GNU Emacs %s (%s%s) of %s on %s")
                 emacs-version
                 system-configuration
                 (cond ((featurep 'motif) ", Motif")
                       ((featurep 'x-toolkit) ", X toolkit")
                       (t ""))
                 (format-time-string "%a %b %e %Y" emacs-build-time)
                 emacs-build-system)))
    (if here
        (insert version-string)
      (if (interactive-p)
          (message "%s" version-string)
        version-string))))
        ...
```

References

[Childs] Bart Childs *"Literate Programming, A Practitioner's View"*
TUGboat, Volume 13, No. 2, 1992
available at: http://www.literateprogramming.com/farticles.html

[cmbtt] *"A Bold Computer Modern Typewriter Font"*, available at:
http://www.ctan.org/tex-archive/fonts/cm/mf-extra/bold

[CWeb] Donald. E. Knuth and Silvio Levy
"The CWEB System of Structured Documentation"
Addison-Wesley, Reading, Mass., 1993

[CWebx] by Marc van Leeuwen
available at: http://wallis.univ-poitiers.fr/~maavl/CWEBx/

[DOCpp] by Roland Wunderling and Malte Zöckler
available at: http://www.zib.de/Visual/software/doc++/

[Doxygen] by Dimitri van Heesch, available at: http://www.doxygen.org

[fancyhdr] Piet van Oostrum *"Page layout in LaTeX"*, available at:
ftp://ftp.dante.de/tex-archive/help/Catalogue/entries/fancyhdr.html

[funnelWeb] by Ross N. Williams, available at: http://www.ross.net/funnelweb/

[fWeb] by John Krommes, available at: http://w3.pppl.gov/~krommes/fweb.html

[gawk] by the Free Software Foundation
available at: http://www.gnu.org/directory/gawk.html

[JDoc] James Gosling, Bill Joy and Guy Steele
"Java Language Specification" Addison-Wesley, 1996

[La2HT] by Nikos Drakos, Ross Moore and many others ...
available at: http://saftsack.fs.uni-bayreuth.de/~latex2ht/
or: http://ctan.tug.org/ctan/tex-archive/support/latex2html

[LitProg] Donald E. Knuth *"Literate Programming"*
The Computer Journal, Vol. 27, No. 2, 1984

[listings] Carsten Heinz *"The Listings package"*, available at:
ftp://ftp.dante.de/tex-archive/help/Catalogue/entries/listings.html

[longtable] David Carlisle *"The longtable package"*, available at:
ftp://ftp.dante.de/tex-archive/help/Catalogue/entries/longtable.html

[LuxiMono] Walter Schmidt *"The LuxiMono package"*, available at:
http://www.ctan.org/tex-archive/fonts/LuxiMono

[multicol] Frank Mittelbach *"An environment for multicolumn output"*, available at: ftp://ftp.dante.de/tex-archive/help/Catalogue/entries/multicol.html

[noWeb] Norman Ramsey *"Literate Programming Simplified"*
IEEE Software, Sep. 1994, p. 97
available at: http://www.eecs.harvard.edu/~nr/noweb/intro.html

[nuWeb] by Preston Briggs
available at: http://ctan.tug.org/tex-archive/web/nuweb

[progDoc] by Volker Simonis, available at: http://www.progdoc.org

[RamMarc] N. Ramsey and C. Marceau *"Literate Programming on a Team Project"*
Software - Practice & Experience, 21(7), Jul. 1991, p. 667-683
available at: http://www.literateprogramming.com/farticles.html

[ShumCook] Stephan Shum and Curtis Cook
"Using Literate Programming to Teach Good Programming Practices"
25th. SIGCSE Symp. on Computer Science Education, 1994, p. 66-70

[TexB] Donald E. Knuth *"The TEXbook"*
Addison-Wesley, Reading, Mass., 11. ed., 1991

[Tex] Donald E. Knuth *"TEX: The Program"*
Addison-Wesley, Reading, Mass., 4. ed., 1991

[VanWyk] Christopher J. Van Wyk *"Literate Programming Column"*
Communications of the ACM, Volume 33, Nr. 3, March 1990. p. 361-362

[Web] Donald E. Knuth *"Literate Programming"*
CSLI Lecture Notes, no. 27, 1992 or Cambridge University Press

[webWeb] by Uwe Kreppel
available at: http://www-ca.informatik.uni-tuebingen.de/people/kreppel/