**Abstract**

PROG2T\TeX is a software tool that allows the inclusion of C++, Java and BNF (Jamoos) programs in \TeX documents. Short program excerpts, whole program files or selected ranges from program files can all be added, as well as small program expressions as part of the flowing text.

1 Introduction

This is a brief documentation of the PROG2T\TeX utility, as well being a test document (as it uses practically all of PROG2T\TeX’s features). Also included is a technical discussion of the changes made since the program’s last version.

PROG2T\TeX allows the inclusion of C++, Java, and BNF (Jamoos) programs in \TeX documents. Source code written in any of these languages can be included in one of four ways:

1. **Code paragraphs** inside the \TeX document. This method can be used to include blocks of source code directly inside the \TeX document.

2. **Inlined code** inside the \TeX text, useful for quoting a small fragment of code (e.g., a single statement or expression) without breaking the flow of reading.

3. **Included files** allows the inclusion of source code directly from source files (e.g., \texttt{.java}, \texttt{.h} or \texttt{.cpp} files).

4. **Included file range** enables the inclusion of only a limited part (a named range) from a source file. The named ranges must be defined in the source file itself (using special delimiters that would seem like comments to the compiler).

Outline The next section explains how PROG2T\TeX should be run. Sections 3 to 6 detail the exact syntax for each of the four source-inclusion modes. Section 7 deals with using \TeX’s normal file inclusion commands (\texttt{\include} and \texttt{\input}). Section 8 describes how \TeX formatting commands can be added to comments in source code. Both Section 9, which contains technical notes, and Section 10, are aimed for future maintainers of PROG2T\TeX, and would be of no interest to most users. Finally, Section 11 is a brief conclusion.
2 Using Prog2TeX

To include source code in \LaTeX documents, the following steps should be taken:

1. Add \usepackage{prog2tex} to the document’s preamble.

2. Use Prog2TeX’s commands in your document. Prog2TeX’s commands include the various source-inclusion macros (detailed in the following sections), as well as the \progtex macro, used to generate Prog2TeX’s “logo”.

3. Before running \latex2e on the document, run prog2tex. This is done using the syntax “prog2tex filename”. If the filename does not include an extension, the default “.tex” extension is assumed. prog2tex modifies the .tex file by adding special macros (“\PROGxx” macros) that can be ignored (or even deleted) when the document is edited again. The program also creates an additional input file that the package uses (with a .prg extension).

4. You can now run \latex2e normally on the document.

If the source document is modified, prog2tex should be run again before each time \latex2e is used. Only one execution of prog2tex is needed every time the file is modified, even if you run \latex2e several times.

By default, prog2tex updates the .tex file on which it operates. If a second command-line argument is provided, prog2tex will generate a new output file (leaving the input file untouched). For example: “prog2tex filename outputfile”. Note, however, that if the input .tex file uses the \LaTeX macros \input or \include, the included files will be updated even if a second argument was specified (see Section 7 for details).

If prog2tex is run without any arguments, it will use stdin for input and stdout for output. In addition, it will create a file called stdin.prg where the generated macros will be stored.

3 Code Paragraphs

To enter source code directly inside your \LaTeX document, use the following syntax:

\begin{verbatim}
/lang source code \end
\end{verbatim}

Where “/lang” is one of “/BNF”, “/CPP” or “/JAVA”, for BNF, C/C++, or Java code, respectively.

For example, consider the following \LaTeX source:
Once processed by PROG2TEX and \TEX, it will produce the following result:

// hello.c
#include <stdio.h>

int main(int argc, char argv[])
{
    // just print...
    printf("Hello, World!\n");
    return 0;
}

Note that keywords appear in boldface, and comments appear in a proportional font. (The way various language parts are rendered can in fact be changed by updating the file prog2tex.sty, without otherwise modifying PROG2TEX’s source code.)

4 Inlined Code

Sometimes, a small piece of code has to be included directly inside normal text. PROG2TEX supports this by using “inline” code macros.

The three inline macros are \bnf, \cpp, and \java, for inlining BNF, C/C++, or Java code fragments, respectively.

Any text passed as a parameter to these macros will be processed to generate properly formatted source code (e.g., keywords will appear in bold, etc.).

For example, consider the following \TEX paragraph:

Constructors in Java must begin with calls to either \java{this(...)} or \java{super(...)}.

It will yield:

Constructors in Java must begin with calls to either this(...) or super(...).
The characters ‘{’ and ‘}’ need not (and cannot) be escaped inside inlined macros. This means that the text included inside the inline macro must not contain unbalanced curly braces. For example, the following code will baffle \texttt{PROG2TE\LaTeX}, and cause unexpected results:

```
\texttt{You must avoid bad usage of }\backslash\texttt{java}\{\texttt{switch} }\backslash\texttt{\} in your programs.\texttt{.}
```

Inlined C/C++ code fragments can appear inside BNF code paragraphs, and vice versa. The nesting can be repeated to any depth necessary. For example, the following block of code is a BNF program that includes inlined C++ code, generated using \texttt{"\cpp\{\ldots\}"}, inside a \texttt{"\BNF\ldots\END"} block:

```
\begin{lstlisting}
Main? \rightarrow \texttt{argv: }\{ \texttt{STRING} \ldots \}\+
\texttt{FEATURES}
\hspace{1em}p: \texttt{PascalProgram}\? \rightarrow \texttt{PARSE(argv[1].PascalProgram)};
\hspace{1em}isLegal: \texttt{OK} \rightarrow \texttt{[[\texttt{if} (! \texttt{?}) \texttt{ERROR}; ]]};
\hspace{1em}triples: \texttt{TriplesList} \rightarrow \texttt{p.triples}\{}\texttt{\};
\hspace{1em}optimized: \texttt{TriplesList} \rightarrow \texttt{triples.optimize}\{}\texttt{\};
\hspace{1em}generate: \texttt{OK} \rightarrow \texttt{optimized.dump}\{}\texttt{\};
\end{lstlisting}
```

(Note that the framebox is not automatically generated as well; it was included here, and in other examples, to increase this document’s readability.)

5 Included Files

Not all source code has to reside inside the \texttt{\LaTeX} document. Source files can be included directly, using the macros \texttt{\cppfile}, \texttt{\bnffile} and \texttt{\javafile} (all-uppercase versions of these macros are also defined, e.g., \texttt{\CPPFILE}).

These macros accept a single parameter, which is the filename to include. If the file is not found, \texttt{PROG2TE\LaTeX} will try again, appending an extension to the name. The extension used is \texttt{.cpp}, \texttt{.bnf} or \texttt{.java}, depending on the macro at hand.

As an example, here’s the complete content of the file \texttt{sample.cpp}, included using the command \texttt{"\cppfile\{sample\}"}. Again, the framebox was added here for increased readability.
This is a sample C++ file.

/*--begin:inter--*/

class Silly {
private:
  int iq;
public:
  int get_iq();
  void set_iq(int new_iq);
}
/*--end:inter--*/

/*--begin:impl--*/

int Silly:get_iq() {
  return iq;
}
/*--end:impl--*/

/*--begin:impl--*/
// This part is outside any marker's range.

int Silly:set_iq(int new_iq) {
  iq = new_iq;
}
/*--end:impl--*/

/*--begin:impl--*/
// This part is outside any marker's range.

int Silly:get_iq() {
  return iq;
}
/*--end:impl--*/

// This part is outside any marker's range.
/*--begin:impl--*/
// Note: we use the same range name again!

int Silly:set_iq(int new_iq) {
  iq = new_iq;
}
/*--end:impl--*/

// This part is outside any marker's range.

This file includes several named ranges, which will be used in the following section.

6 Included File Ranges

Using named ranges, \texttt{PROG2TEX} supports the inclusion of selected parts from a source file (rather than including the entire file).

The beginning of a range inside a source file is marked using the string

/*--begin:rangeame--*/

The compiler will naturally consider this marker to be a comment, and ignore it. In a similar manner, the end of a range is marked using the string

/*--end:rangeame--*/

To include a range, use the syntax

\begin{verbatim}
\cppfile{filename:rangeame}
\end{verbatim}

(The same can be done using \texttt{javfile} and \texttt{bnffile} as well.) For example, using the file \texttt{sample.cpp} from the previous Section, the command
will yield this output:

class Silly {
  private:
    int iq;
  public:
    int get_iq();
    void set_iq(int new_iq);
}  

Ranges can be nested, or even cross each other’s borders. In fact, a range does not even have to be continuous: if the same range name is used more than once, it is considered a single, non-continuous range. For example, here is the range named “impl” from sample.cpp:

int Silly::get_iq() {
    return iq;
}

    // Note: we use the same range name again!

int Silly::set_iq(int new_iq) {
    iq = new_iq;
}

The range markers themselves are not considered part of the range, and hence are not included in the \LaTeX\ document (though if a range \texttt{R}_1 contains the markers for some range \texttt{R}_2 internally, including range \texttt{R}_1 will show \texttt{R}_2’s markers, just as it would show any other comment contained within the range).

7 Including \LaTeX\ Files

\LaTeX\ documents processed with \texttt{PROG2\LaTeX} can use the normal file-inclusion commands \texttt{	extbackslash include} and \texttt{	extbackslash input}. In this case, \texttt{PROG2\LaTeX} will treat source code fragments found the included \LaTeX\ files as well.

For example, \texttt{this very paragraph} appears in the file \texttt{included.tex}, and was included using a normal \texttt{	extbackslash input} command.

The file includes a simple Java program:
public class Hello {
    /*
     * main: the program’s starting point.
     * @param args the arguments passed on the command-line.
     * @author every Java programmer.
     */
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

Note how @-keywords in javadoc comments appear in boldface.
(The file included.tex ends here.)

The included files are processed by PROG2TEX individually, and are modified if they include source fragments (code paragraphs, inlined code, included source files or included ranges). As with the main file, the modifications are limited to adding \PROGxx macros that can be ignored when the file is edited.

Every run of PROG2TEX generates a single macros (.prg) file, even if the main .tex file includes numerous source files and/or other \TeX files.

8 Using \TeX Notation in Source Code

When writing a program that will eventually be included in a \TeX document, you can include \TeX formatting commands in the program’s comments. This is done using “//{}” for single-line comments in C, C++ or Java, or “/*{}*/” for multi-line (block) comments (these are closed using the normal “*/” pair).

For example, consider the following \TeX code, containing a Java code fragment:

\JAVA
if (a[i] > max) //{} i.e., $a_i$ is the new maximum
    return Math.pow(x, Math.PI); //{} return $x^n pi$
\END

And its result:

\JAVA
if (a[i] > max) // i.e., $a_i$ is the new maximum
    return Math.pow(x, Math.PI); // return $x^n$
\END

All one-line comments (“--”) in BNF code automatically support \TeX formatting commands. javadoc comments cannot include \TeX formatting.

9 Technical Notes

This section provides a list of changes of note in PROG2TEX’s source code (mainly the file prog2tex.l) since the previous version. The previous version did not support inclusion (neither source code inclusion or the processing of included \TeX files), though the basic infrastructure for such support did exist.

1. The code was heavily retouched for uniform indentation, and detailed comments was added.
2. The Buffer construct had a problematic design and was improperly used. It was simply removed, since only a single output (.prg) file is generated anyway.

3. Several elements (functions, flex start-conditions, etc.) were defined but never used. These include BID, VAR, BVAR, Language::begin\_block(), etc. They were all were removed.

4. Several elements were renamed to more properly describe their usage (e.g., d2a() was renamed to PROG\_macro\_id(); PASS mode was renamed LATEX; CODE and CODE1 were renamed CPP and CPP1; etc.).

5. Since only a single macros file is generated, the file inclusion stack (Includes, but see below) no longer keeps track of the macro output file.

6. Added support for a default extension to included source files.

7. Some functions (e.g., process() and sprocess()) were changed so they now return a bool value (instead of int).

8. Several global status variables (e.g., line, pos, etc.) are now stored per source file. To this end, they are now defined as fields in the FileState class.

9. The file inclusion stack (Includes) is now used for both \LaTeX \ and included source files. This allows proper handling of both source code includes and \LaTeX \ \include and \\insert commands. Even the main file is considered “included”, and is pushed onto this stack at program start. Hence, the class was renamed Sources (since it is used even if not a single file is \include'd).

Several bugs in the original code were found and corrected. This includes “active” bugs, as well as “dormant” bugs in the incomplete infrastructure for supporting file inclusion.

Some (fixed) bugs of interest include:

1. The main() function relied on a random non-zero initialization of a local variable to work properly. That variable was actually not needed at all.

2. If the processed .tex file was not in Unix’s \pwd (present working directory), the temp file was generated in \pwd but never deleted. Now the temp file is always created in the .tex file’s directory, and deleted (except in cases of abnormal termination).

3. The initial state was reported (by the mode-tracking debug facilities) as INITIAL, regardless of any calls to BEGIN before yy\_lex() starts. This was due to an improper initialization of the variable state, plus an “unintercepted” call to BEGIN in process().

4. The global variable lang (of type Language *) was constantly assigned new instances of Cpp/Bnf, with no cleanup. To fix this problem, three “singleton”\(^1\) instances of the Cpp, Bnf, Java classes are used. lang is used to point to relevant instance instead of a new one each time. This is possible since the only member field of Language is name, which does not change during the object’s lifecycle.

5. If an included file ended abruptly (e.g., inside a comment), the parser would lose sync and the state-stack would become meaningless.

\(^1\)These are simply three global variables, one per class. They do not actually follow the \texttt{Singleton} design pattern, even though such a change is possible.
6. The line number report in the generated .prg file (stating from which line in the file was the code fragment taken) generated incorrect numbering, and could not support multiple input files. This was fixed, and the report format changed from “Line n” to “filename:n” (e.g., “test.tex:15”).

7. LaTeX-style support for block comments was not working, since whoever added the align mode did it in a way that broke that support (specifically, when returning from align, the code always returned to comment mode, never to \texttt{LATEX\_COMMENT}).

8. The align mode caused loss of tab characters in the source file.

10 Future Changes

“Plan to throw one away; you will, anyhow” (Fredrick P. Brooks, Jr., \textit{The Mythical Man-Month}).

It is time to “throw one away” for this program; the current design has outlived its usefulness, and if another major upgrade is needed, the best way to do it would be a complete rewrite (in a programming language worthy of the name, if possible; in other words, not in C/C++).

Other than a complete rewrite, here are a few suggestions for improving the existing design:

1. The abstract \texttt{Language} class and its concrete language-specific subclasses is a good idea, that came into use too late. Currently, it is used only for included files. However, there’s no reason not to use it for any code fragment, of any type.

2. The \texttt{Language} class can be put to additional uses other than those currently employed (which are mainly opening and closing blocks in a language-specific manner). In particular, it can be used for detecting language keywords. This would allow adding support for new languages with significantly less hassle, since the flex grammar need not be updated for the keywords of each new language. Instead, knowledge about the keywords will simply be stored in the new subclass of \texttt{Language}.

3. File inclusion is currently handled as a stack (the \texttt{Sources} variable, previously called \texttt{Includes}). It would be better to change this into a queue, with a mechanism that would prevent duplicate entries. This change would have two direct advantages: (a) if a source file is included more than once, it will only be processed once, and (b) in the unlikely case that two \LaTeX files mutually include each other (technically possible, since conditional \LaTeX commands can be used to prevent endless nesting), the current design would reach an endless inclusion loop, whereas a queue design would not.

4. \texttt{PROG2TEX} macros found inside \TeX comments are currently processed just like any other. The program can be taught about \TeX comments so it would ignore them.

11 Conclusion

It still surprises me, considering their origin and authors, that while both \TeX and \LaTeX include built-in support for poetry and verse, neither includes proper handling of source code inside documents.