# 1 Introduction

This is dpp, a C/C++ pretty-printer for the literate programming tool noweb. noweb does not prettyprint code by default; by inserting dpp into the noweb pipeline, you can produce output similar to that of CWEB.

Here are some of the features of dpp:

- Keywords are typeset in **bold**, variables in *italics*, comments in roman, strings in typewriter.
- Some punctuation is prettyprinted as well.
- Keywords are definable by the user with the '@ %keyword' construct.
- All instances of code quoting ([[ ]]) are prettyprinted, including quotes inside comments and chunk names.
- Comments are optionally fed straight to TEX, so that you use (for example) TEX's extensive math typesetting capabilities.
- Indentation and line-breaking are not messed with.
- dpp is written in Perl, which you may or may not consider an advantage.

To use dpp, just provide the '-filter dpp' option to noweave. There are a few options:

- '-tex' will send your comments to TEX rather than print them verbatim. It is off by default to avoid nasty surprises for people running their programs through dpp for the first time.
- '-cw' compresses whitespace in the code. It compresses all whitespace not at the beginning of a line to one space, and expands whitespace between code and comments to four characters. Basically, it undoes hand-formatting that looks good when monospaced but bad in a proportional font.
- '-N*root*' turns off prettyprinting for the chunk named *root* and all of its descendants. This way you can include a makefile or test data in your .nw file without having it printed like C code.

To tell dpp about a user-defined keyword such as Elephant, simply insert a line such as '@ %keyword Elephant' into the .nw file. Multiple keywords may be specified on a single line.

dpp was written by Dan Schmidt (dfan@alum.mit.edu). The current version is 0.2.1b.

To do:

- Put all TEX stuff in dppmac.tex or something.
- Make sure I am interacting with LATEX $2_\varepsilon$ correctly.
- Finish this section of the document! Provide an example.
- {\tt{}{\char35}{\char35}} breaks LATEX.
- Comments on lines with preprocessor directives.
- Multiline /* */ comments.
- Multiline preprecessor directives.
- Numeric suffixes like 123456789L, 0xfUL
- char another_difficult_one = '\777';
- Be able to typeset specific words in typewriter.
- Identifier crossreferencing (and index) should be typeset like code (may require changes to totex).
- finduses seems pretty slow, and I'm doing a lot of the same work. Can I fold that functionality in?
- Parameterize things like using \bf for keywords.
- Single-quoted character constants use the wrong quote mark. But so does CWEB. If it's good enough for Knuth...

- Italic correction when necessary.
- Blank lines should not be a whole line tall.
- HTML version?
- Profile.

To not do:

- Can't nest `[[ ]]`.

# 2 Code

`dpp` is written in Perl 5. To understand this code, it's probably best to first skim through the *Noweb Hacker's Guide*.

Note: I am a Perl hacker, not a TEX hacker. Therefore, there are many places in this code where it would probably make sense to accomplish tasks through TEX code, and instead I do lots of things "by hand." I appreciate any suggestions for cleanup.

2a ⟨dpp 2a⟩≡

```
#!/usr/bin/perl
# dpp version 0.2.1b
# Copyright (C) 1997 Dan Schmidt, <dfan@alum.mit.edu>,
#   see dpp.nw for full notice
# Don't modify this file, modify dpp.nw!
```

⟨Read options 2b⟩
⟨Initialize things 2c⟩
⟨Make a first pass through the input, collecting keywords and associating chunks with their roots 3c⟩
⟨Make a second pass through the input, producing output 5e⟩
⟨Subroutines 4a⟩

This code is written to file `dpp`.

Individual options will be introduced during the course of the program. Here's the shell of the option-processing section.

2b ⟨Read options 2b⟩≡

```
while ($option = shift) {
  ⟨If $option is a valid option, process it and continue 6c⟩
}
```

Defines:
`option`, used in chunks 6c, 9e, and 15c.
This code is used in chunk 2a.

## 2.1 Setting up STDIN

We make two passes through `STDIN`, so we have to make sure we'll be able to seek back to the beginning of it.

2c ⟨Initialize things 2c⟩≡

⟨Make sure that `STDIN` is seekable 2d⟩

This definition is continued in chunks 3a, 5d, 7, 11b, and 15e.
This code is used in chunk 2a.

2d ⟨Make sure that `STDIN` is seekable 2d⟩≡

```
$TMPFILE = "dpp.tmp";
if (! -f STDIN) {
  open TMPFILE, "> $TMPFILE"
    or die "Couldn't open `$TMPFILE' for writing: $!; aborting";
  print TMPFILE while <STDIN>;
  close TMPFILE;
  open STDIN, "< $TMPFILE"
    or die "Couldn't open `$TMPFILE' for reading: $!; aborting";
  unlink $TMPFILE;
}
```

This code is used in chunk 2c.

## 2.2 Keywords

Let's make our list of keywords first, so we can get it out of the way, and be done with the first pass. For quick lookup, we have a hash `%is_keyword`, so that `$is_keyword{$word}` is 1 if `$word` is a keyword and undefined otherwise. To make things easier, we build up an array `@keywords` of keywords, and insert them into `%is_keyword` all at once.

3a ⟨Initialize things 2c⟩+≡
⟨Initialize the keyword list `@keywords` 3b⟩

3b ⟨Initialize the keyword list `@keywords` 3b⟩≡
```
@keywords = qw(asm auto bool break case catch char class const
               const_cast continue default delete do double
               dynamic_cast else enum exit explicit extern FALSE
               false float for friend goto if inline int long mutable
               namespace new NULL null operator private protected public
               register reinterpret_cast return short signed sizeof
               static static_cast struct switch template throw try
               TRUE true typedef union unsigned using virtual
               void volatile wchar_t while );
```
Defines:
`keywords`, never used.
This code is used in chunk 3a.

3c ⟨Make a first pass through the input, collecting keywords and associating chunks with their roots 3c⟩≡
⟨Scan input, adding keywords to `@keywords` and discovering parents of chunks 3d⟩
⟨Convert `@keywords` to `%is_keyword` 3e⟩
⟨Process parental information 4c⟩
⟨Reset for the second pass 3f⟩
This code is used in chunk 2a.

Keyword lines are of the form `%keyword kw1 kw2 ...` in the original input file, but look like `@text %keyword kw1 kw2 ...` by the time they get to us.

Note that `dpp` makes no use of the Perl constructs `$‘`, `$&`, and `$’`. Jeffrey Friedl's book *Mastering Regular Expressions* notes that any use of these three variables slows down all regular expressions throughout the program.

3d ⟨Scan input, adding keywords to `@keywords` and discovering parents of chunks 3d⟩≡
```
while (<>) {
  if (/^\@text %keyword (.*)/) {
    push @keywords, (split " ", $1); next;
  }
  ⟨Process this line for parental information 4b⟩
}
```
This code is used in chunk 3c.

3e ⟨Convert `@keywords` to `%is_keyword` 3e⟩≡
```
foreach $word (@keywords) {
  $is_keyword{$word} = 1;
}
```
Defines:
`is_keyword`, used in chunk 4a.
This code is used in chunk 3c.

3f ⟨Reset for the second pass 3f⟩≡
```
seek (STDIN, 0, 0);
```
This code is used in chunk 3c.

We already know how to prettyprint a word now; if it's in the keyword list it's bold, otherwise it's in italics. Originally, this was most of the entire program, but things have gotten a bit more complicated since then...

4a ⟨Subroutines 4a⟩≡
```
sub pp_word {
  my ($this_word) = shift;
  if (defined $is_keyword{$this_word}) { return "{\\bf{}$this_word}"; }
  else { return "{\\it{}$this_word}"; }
}
```
Defines:
  pp_word, used in chunk 10a.
Uses is_keyword 3e.
This definition is continued in chunks 8b, 11, 12, 14a, and 15b.
This code is used in chunk 2a.

## 2.3 Discovering roots

For each chunk, we need to know its root chunk, the farthestmost ancestor of this chunk in the forest of chunks. When we eventually print this chunk during the second pass, we will not prettyprint it if its root has been identified as not being C or C++.

While we read over the input, if we see that chunk $a invokes chunk $b, we set $ancestor{$b} = $a. A root chunk will have no ancestor, since it is never invoked.

Of course, the chunks actually define a directed acyclic graph, not a forest, since a chunk may be invoked from more than one place. But pretending it has only one parent will not affect what language its root chunk is in (I sincerely hope).

4b ⟨Process this line for parental information 4b⟩≡
```
$in_quote = 1 if (/^\@quote/);
$in_quote = 0 if (/^\@endquote/);
if (! $in_quote) {
  if (/^\@defn (.*)/) {
    $cur_chunk = $1; push @chunks, $cur_chunk; next;
  }
  $ancestor{$1} = $cur_chunk, next if (/^\@use (.*)/);
}
```
Defines:
  ancestor, used in chunks 5 and 6b.
  chunks, never used.
  in_quote, never used.
This code is used in chunk 3d.

Now we go through all the chunks, finding each one's farthestmost ancestor. By the end of this process, $ancestor{$chunk} will be the root chunk of each chunk $chunk. Each chunk is marked as 'settled' when its root is determined.

We settle all chunks on the path between the current chunk and the root chunk, remembering them by putting them in @chunks_to_settle. This saves time when processing future chunks, since as soon as we get to a settled chunk while ascending the tree, we know that its root is our root.

4c ⟨Process parental information 4c⟩≡
```
foreach $chunk (@chunks) {
   next if $settled{$chunk};
   ⟨Set $root to the root chunk of $chunk, and put unsettled chunks on the ancestral path in @chunks_to_settle 5a⟩
   ⟨Set the ancestor of each chunk in @chunks_to_settle to $root and settle it 5c⟩
}
```
Defines:
  chunk, used in chunk 5a.
  settled, used in chunk 5.
This code is used in chunk 3c.

We ascend the tree until we get to a settled chunk, or one that has no ancestor, which must be a root. Along the way, we add to our list of chunks to settle.

5a ⟨Set `$root` to the root chunk of `$chunk`, and put unsettled chunks on the ancestral path in `@chunks_to_settle` 5a⟩≡

```
$c = $chunk; @chunks_to_settle = ();
while ((! $settled{$c}) && ($ancestor{$c})) {
  push @chunks_to_settle, $c;
  $c = $ancestor{$c};
}
⟨Set $root based on $c 5b⟩
```

Defines:
`chunks_to_settle`, never used.
Uses `ancestor` 4b, `chunk` 4c, and `settled` 4c.
This code is used in chunk 4c.

We may have left the above loop for two different reasons. If `$c` has an ancestor, then `$ancestor{$c}` is its root (since it's settled), and ours as well.

Otherwise, `$c` is a root that we're seeing for the first time, and we settle it 'by hand.'

5b ⟨Set `$root` based on `$c` 5b⟩≡

```
if ($ancestor{$c}) { $root = $ancestor{$c}; }
else {
  $root = $c;
  $ancestor{$root} = $root;
  $settled{$root} = 1;
}
```

Defines:
`root`, used in chunk 5c.
Uses `ancestor` 4b and `settled` 4c.
This code is used in chunk 5a.

Now that `$root` is set, the last step is trivial.

5c ⟨Set the ancestor of each chunk in `@chunks_to_settle` to `$root` and settle it 5c⟩≡

```
foreach $c (@chunks_to_settle) {
  $ancestor{$c} = $root;
  $settled{$c} = 1;
}
```

Uses `ancestor` 4b, `root` 5b, and `settled` 4c.
This code is used in chunk 4c.

## 2.4 The outer loop

Our main job is to go through the input file, prettyprinting whatever lines occur in code or quoted code. The former lines are bracketed by `@begin code` and `@end code`, the latter by `@quote` and `@endquote`. We use a boolean variable `$incode` to keep track of which state we're in.

5d ⟨Initialize things 2c⟩+≡

```
$incode = 0;
```

Defines:
`incode`, used in chunks 5–7.

5e ⟨Make a second pass through the input, producing output 5e⟩≡

```
while (<>) {
  if ($incode) {
    ⟨Process and print a line of code 7d⟩
  } else {
    ⟨Process and print a line of documentation 6a⟩
  }
}
```

Uses `incode` 5d.
This code is used in chunk 2a.

### 2.4.1 Documentation lines

We'll handle documentation lines first, because they're easier. There are a few special cases that we have to handle, but basically we just do a little processing (usually unnecessary), output the line, and see if we have to go into "code mode."

6a ⟨Process and print a line of documentation 6a⟩≡
```
⟨Ignore this line if it's a %keyword line 6d⟩
⟨Print out special LaTeX code if appropriate 7b⟩
⟨Prettyprint part of a documentation line if necessary 6e⟩
print $_;
⟨Set $incode to 1 and enter "code mode" if appropriate 6b⟩
```
This code is used in chunk 5e.

When entering code mode, we set the font so that roman, not italics will be the default font of the code. We don't want to go into code mode at all if this chunk is indirectly invoked by a root chunk that has been explicitly requested to be printed plain.

6b ⟨Set $incode to 1 and enter "code mode" if appropriate 6b⟩≡
```
if (/^\@quote/ || (/^\@defn (.*)/ && (! $plain{$ancestor{$1}}))) {
  $incode = 1;
  print "\@literal \\Rm{}\n";
}
```
Uses ancestor 4b, incode 5d, and plain 6c.
This code is used in chunk 6a.

Setting up the list of "plainprinted" chunks is easy.

6c ⟨If $option is a valid option, process it and continue 6c⟩≡
```
if ($option =~ /^-N(.*)$/) { $plain{$1} = 1; }
```
Defines:
plain, used in chunk 6b.
Uses option 2b.
This definition is continued in chunks 9e and 15c.
This code is used in chunk 2b.

The %keyword lines are in the source just for our benefit, and we want to strip them out so later filters don't have to deal with them. We have to be tricky in order to keep line numbers consistent, so we use the @index nl trick introduced mentioned in the *Noweb Hacker's Guide*. We replace the %keyword line and the newline @nl that comes after it by a dummy newline.

6d ⟨Ignore this line if it's a %keyword line 6d⟩≡
```
if (/^\@text %keyword/) {
  $nextline = (<>);
  die "%keyword confusion\n" if (! ($nextline =~ /^\@nl$/));
  print "\@index nl\n";
  next;
}
```
This code is used in chunk 6a.

We may need to prettyprint something even if we're in documentation mode. The two cases are 1) the name of a chunk we're about to define contains a code quote, and 2) the index mentions a chunk whose name contains a code quote. The function pp_line is used to prettyprint a line of code.

All this line does is grab everything before [[ and after ]], and then insert the contents of the quote between them.

We execute the substitution multiple times in order to catch multiple quotes on the same line. The (?!\]) trickiness is to deal with quoting code that ends with a right bracket; we make sure to catch the outermost bracket pair.

6e ⟨Prettyprint part of a documentation line if necessary 6e⟩≡
```
1 while s/((?:\@xref chunkbegin|\@defn) .*?)\[\[(.*?)\]\](?!\])(.*)/$1 . pp_line($2) . $3/e;
```
Uses pp_line 8b.
This code is used in chunk 6a.

**The LaTeX header.** There's just one bit left to deal with regarding documentation lines. We need to print out some special-purpose LaTeX code at some point, traditionally the end of the first documentation chunk. The variable `$delay` is 1 if we are waiting to print out said code, which is stored in `$texdefs`.

7a ⟨Initialize things 2c⟩+≡
```
$delay = 1;
my ($texdefs);
⟨Set up $texdefs 7c⟩
```
Defines:
`delay`, used in chunk 7b.
`texdefs`, used in chunks 7, 10d, 14c, and 15d.

7b ⟨Print out special LaTeX code if appropriate 7b⟩≡
```
if ($delay && /^\@end docs/) {
  print $texdefs;  $delay = 0;
}
```
Uses `delay` 7a and `texdefs` 7a.
This code is used in chunk 6a.

Here's an example of something to go in `$texdefs` (there will be more down the line). Given that quoted code uses italics for variable names, it makes much more sense for chunk names to be in roman, as in `CWEB`. So I override `noweb.sty` here. These definitions are exactly the same as `noweb.sty`'s `\LA` and `\RA` except for `\Rm` instead of `\It`.

7c ⟨Set up $texdefs 7c⟩≡
```
$texdefs .=
  "\@literal \\def\\LA{\\begingroup\\maybehbox\\bgroup\\setupmodname\\Rm\$\\langle\$}\n" .
  "\@literal \\def\\RA{\$\\rangle\$\\egroup\\endgroup}\n";
```
Uses `texdefs` 7a.
This definition is continued in chunks 10d, 14c, and 15d.
This code is used in chunk 7a.

### 2.4.2 Code lines

The outer loop for prettyprinting the lines of code is theoretically simple, but it's complicated by the fact that previous filters may have broken up lines in inconvenient places such as the middle of a comment. To compensate, as we parse between `@nl`'s we accumulate two kinds of text. Actual code is accumulated in `$text`, and other markup lines are accumulated in `$extra`. When we finally hit a `@nl`, we prettyprint `$text` and print it out, followed by the `$extra` text we've saved up. I don't think this will mess up any crossreferencing.

7d ⟨Process and print a line of code 7d⟩≡
```
if (/\@text (.*)/) { $text .= $1; }
elsif (/\@nl/ || /\@end/) {
  ⟨Print out $text and $extra, and reset them 7e⟩
  print $_;
  $incode = 0 if (/\@end/);
} else {
  ⟨Prettyprint part of a code markup line if necessary, adding it to $extra 8a⟩
}
```
Uses `incode` 5d and `text` 7f.
This code is used in chunk 5e.

7e ⟨Print out $text and $extra, and reset them 7e⟩≡
```
if (length $text)  { print "\@literal " . pp_line ($text) . "\n"; }
if (length $extra) { print "$extra"; }
$text = $extra = "";
```
Uses `extra` 7f, `pp_line` 8b, and `text` 7f.
This code is used in chunks 7d and 8a.

7f ⟨Initialize things 2c⟩+≡
```
$text = $extra = "";
```
Defines:
`extra`, used in chunks 7e and 8a.
`text`, used in chunk 7.

All the actual code is in `@text` lines, but there are three other cases in which we have to prettyprint part of the line. We may be referencing another chunk, defining a variable, or defining a chunk.

If we are referencing another chunk, then everything we've accumulated up to now should be syntactically correct, so we can prettyprint that and spit it out, print out the chunk reference, and start over.

It seems that I can't prettyprint the `@index defn` line, because `totex` will automatically escape my typesetting. I'm not sure if that line ever worked.

8a ⟨Prettyprint part of a code markup line if necessary, adding it to `$extra` 8a⟩≡

```
if (/\@use|\@defn/) {
  1 while s/((?:\@use|\@defn) .*?)\[\[(.*?)\]\](?!\])(.*)/$1 . pp_line($2) . $3/e;
  $extra .= $_;
  ⟨Print out $text and $extra, and reset them 7e⟩
} else {
# s/(\@index defn )(\S+)/$1 . pp_line($2)/e;
  $extra .= $_;
}
```

Uses `extra` 7f and `pp_line` 8b.
This code is used in chunk 7d.

## 2.5 Prettyprinting lines

Now we've reduced the problem to prettyprinting an individual line of code. Luckily, we don't have keep track of context much, since for the most part, we're going to do the same thing with a string of letters (for example) no matter where we see it.

We process the line from left to right, looking for tokens to beautify and appending them to `$preline`. As we go through this loop, `$line` contains the part of the line that we have yet to process, while `$preline` contains the already-processed beginning of the line. In some cases, we need to save off post-processed stuff to put at the end of the line, which is stored in `$postline`. Thus, at any point, `$preline . $line . $postline` will reconstruct all of the original line (although the middle part will not be prettyprinted yet). When `$line` is empty, we're done.

`$seen_token` is 0 before we do anything, and 1 after we've processed at least one token of the line. We use it if we're compressing whitespace.

There are three major kinds of constructs that really do not work and play well with others: strings, comments, and preprocessor directives. We grab these early before any of the "regular" prettyprinting code can see them.

8b ⟨Subroutines 4a⟩+≡

```
sub pp_line {
  my ($line) = shift;  my ($preline, $postline, $token, $seen_token);
  ⟨Process preprocessor lines 10c⟩
  ⟨Process strings and comments 13a⟩
  while (length $line) {
    ⟨Remove a token from the beginning of $line, process it, and append it to $preline 9a⟩
  } continue {
    $seen_token = 1;
  }
  return $preline . $postline;
}
```

Defines:
`line`, used in chunks 9, 10, and 13–15.
`postline`, used in chunk 15a.
`pp_line`, used in chunks 6–8 and 13–15.
`preline`, used in chunks 9, 10, 13b, and 14b.
`seen_token`, used in chunk 9d.
`token`, used in chunk 10.

There's a restriction on the ordering of these chunks: if the beginning of the line fulfills the patterns for two different kinds of tokens, we want to match the longer one, so its corresponding chunk must come first. For example, we want to typeset `0xff` like a number, so we must let the hex-checker grab the whole thing before the number-checker takes just the leading 0, leaving us with the "word" `xff`, which would be set in italics.

9a ⟨Remove a token from the beginning of `$line`, process it, and append it to `$preline` 9a⟩≡
⟨If the first token is whitespace, process it and continue 9d⟩
⟨If the first token is a hex number, process it and continue 9b⟩
⟨If the first token is a number, process it and continue 9c⟩
⟨If the first token is a word, process it and continue 10a⟩
⟨If the first token is punctuation, process it and continue 10b⟩
This code is used in chunk 8b.

### 2.5.1 The outer loop

The following chunks are presented out of order, so that we can start with the simpler ones. For these cases, we strip off the matching part, do nothing to it (so it will be typeset in roman) and add it to `$preline`.

9b ⟨If the first token is a hex number, process it and continue 9b⟩≡
```
if ($line =~ /^(0[xX][\dabcdefABCDEF]+)(.*)/) {
  $preline .= $1;  $line = $2;  next;
}
```
Uses `line` 8b and `preline` 8b.
This code is used in chunk 9a.

9c ⟨If the first token is a number, process it and continue 9c⟩≡
```
if ($line =~ /^(\d+)(.*)/) {
  $preline .= $1;  $line = $2;  next;
}
```
Uses `line` 8b and `preline` 8b.
This code is used in chunk 9a.

You'd think that whitespace should be the simplest case, but we muck with it a bit. The idea is that people often insert extra whitespace in order to align their comments or equals signs. This looks great with a monospaced font, but stupid with a variably-spaced font, so we take it all out. This explains why we need `$seen_token`; we don't want to compress any whitespace that occurs at the very beginning of the line, since that would destroy all indentation. All messing with whitespace is done only if the user has specified the '`-cw`' option.

9d ⟨If the first token is whitespace, process it and continue 9d⟩≡
```
if ($line =~ /^(\s+)(.*)/) {
  if ($compress_whitespace && $seen_token) {
    $preline .= " ";  $line = $2;  next;
  } else {
    $preline .= $1;  $line = $2;  next;
  }
}
```
Uses `compress_whitespace` 9e, `line` 8b, `preline` 8b, and `seen_token` 8b.
This code is used in chunk 9a.

9e ⟨If `$option` is a valid option, process it and continue 6c⟩+≡
```
$compress_whitespace = 1, next if ($option =~ /^-cw/);
```
Defines:
`compress_whitespace`, used in chunks 9d, 14b, and 15a.
Uses `option` 2b.

In the case of a word, we prettyprint it with the `pp_word` subroutine we defined ages ago. We also have to escape all underscore characters so TEX doesn't think they're subscripts.

We already checked that the token doesn't begin with a digit, so we can just stick `\d` in our character class without worrying about it.

10a ⟨If the first token is a word, process it and continue 10a⟩≡
```
if ($line =~ /^([a-zA-Z\d_]+)(.*)/) {
  $token = $1;  $line = $2;
  ($token = pp_word ($token)) =~ s|_|\\_|g;
  $preline .= $token;
  next;
}
```
Uses `line` 8b, `pp_word` 4a, `preline` 8b, and `token` 8b.
This code is used in chunk 9a.

The only possibility left is that we're handling a string of punctuation, in which case we hand it off to the `pp_punc` subroutine.

10b ⟨If the first token is punctuation, process it and continue 10b⟩≡
```
if ($line =~ /^([^\d\sa-zA-Z_]+)(.*)/) {
  $token = $1;  $line = $2;  $preline .= pp_punc ($token);  next;
}
```
Uses `line` 8b, `pp_punc` 12, `preline` 8b, and `token` 8b.
This code is used in chunk 9a.

We still haven't bothered to deal with preprocessor directives. We just set the first word in bold and everything else in typewriter; perhaps for certain constructs (like `#define`), we should `pp_line` the rest of the line.

`escape` is a general subroutine meant to take an arbitrary string of characters and massage it into a form that TEX won't gack on.

10c ⟨Process preprocessor lines 10c⟩≡
```
if ($line =~ /^(\s*)#(\s*\S*)((\s*)(.*))/) {
  $arg = escape ($3);
  $preline = "{\\bf{}$1\\char35{}$2}{\\tt{}$arg}";
  $line = "";
}
```
Uses `escape` 14a, `line` 8b, and `preline` 8b.
This code is used in chunk 8b.

### 2.5.2 Processing punctuation

First, let's set up a few TEX definitions for some symbols. These two are stolen from Kaelin Colclasure's pretty printer (found in `contrib/kaelin` in the `noweb` distribution).

10d ⟨Set up `$texdefs` 7c⟩+≡
```
$bm = "\\begin{math}";  $em = "\\end{math}";
$texdefs .= "\@literal \\providecommand{\\MM}{\\kern.5pt\\raisebox{.4ex}" .
            "{$bm\\scriptscriptstyle-\\kern-1pt-$em}\\kern.5pt}\n" .
            "\@literal \\providecommand{\\PP}{\\kern.5pt\\raisebox{.4ex}" .
            "{$bm\\scriptscriptstyle+\\kern-1pt+$em}\\kern.5pt}\n";
```
Defines:
`bm`, used in chunks 11a and 14.
`em`, used in chunks 11a and 14.
Uses `texdefs` 7a.

We call `init_punc` to set up the punctuation table. `reg_punc` associates a sequence of punctuation found in the source with its typeset equivalent; matches are checked in the order that they are given to `reg_punc`. We need to enter all the longer matches first, so that we don't do something like typeset a `<` before we find out that it's really part of `<=` and we want to typeset it as $\leq$.

11a ⟨Subroutines 4a⟩+≡

```
sub init_punc {
  reg_punc ("!=", "${bm}\\neq${em}");
  reg_punc ("&&", "${bm}\\wedge${em}");
  reg_punc ("++", "\\protect\\PP");
  reg_punc ("--", "\\protect\\MM");
  reg_punc ("->", "${bm}\\rightarrow${em}");
  reg_punc ("<<", "${bm}\\ll${em}");
  reg_punc ("<=", "${bm}\\leq${em}");
  reg_punc ("==", "${bm}\\equiv${em}");
  reg_punc (">=", "${bm}\\geq${em}");
  reg_punc (">>", "${bm}\\gg${em}");
  reg_punc ("||", "${bm}\\vee${em}");

  reg_punc ("!", "${bm}\\neg${em}");
  reg_punc ("*", "${bm}\\ast${em}");
  reg_punc ("/", "${bm}\\div${em}");
  reg_punc ("<", "${bm}<${em}");
  reg_punc (">", "${bm}>${em}");
  reg_punc ("^", "${bm}\\oplus${em}");
  reg_punc ("|", "${bm}\\mid${em}");
  reg_punc ("~", "${bm}\\sim${em}");

  reg_punc ("{", "{\\nwlbrace}");
  reg_punc ("}", "{\\nwrbrace}");
}
```

Defines:
`init_punc`, used in chunk 11b.
Uses `bm` 10d, `em` 10d, and `reg_punc` 11c.

11b ⟨Initialize things 2c⟩+≡

```
init_punc();
```

Uses `init_punc` 11a.

`reg_punc` puts the punctuation sequence at the end of the list `@puncs`, and makes the hash `%punc_map` contain, for each punctuation sequence, its typeset equivalent. Earlier insertions appear before later ones.

11c ⟨Subroutines 4a⟩+≡

```
sub reg_punc {
  my ($punc, $set) = @_;
  push @puncs, $punc;
  $punc_map{$punc} = $set;
}
```

Defines:
`punc_map`, used in chunk 12.
`puncs`, never used.
`reg_punc`, used in chunk 11a.

I'm not too happy about the punctuation routine `pp_punc`; it just doesn't look that efficient, with all those `substr`'s everywhere, but I couldn't find a faster way to get at those individual characters. Profiling must be done.

`$this_punc` and `$out` are analogous to `$line` and `$preline` in `pp_line`; `$this_punc` contains what remains to be typeset, and `$out` contains the stuff that's been stripped out of `$this_punc` and typeset.

We go through `@puncs` in order looking for exact matches, and if we find one, we replace the match by the TEX code in `%punc_map`.

12 ⟨Subroutines 4a⟩+≡

```
sub pp_punc {
  my ($this_punc) = shift;  my ($out);
punc_loop:
  while (length $this_punc) {
    foreach $punc (@puncs) {
      if (substr ($this_punc, 0, length $punc) eq $punc) {
        $out .= $punc_map{$punc};
        $this_punc = substr ($this_punc, length $punc);
        next punc_loop;
      }
    }
    # No match found
    $out .= substr ($this_punc, 0, 1);
    $this_punc = substr ($this_punc, 1);
  }
  return $out;
}
```

Defines:
`pp_punc`, used in chunk 10b.
Uses `punc_map` 11c.

### 2.5.3 Processing strings and comments

The hairiest part has been saved for last. The following regular expression was mostly cribbed from Jeffrey Friedl's excellent book *Mastering Regular Expressions*. It finds the earliest (in the line) occurrence of a string or comment. We then prettyprint that element correctly and look at the rest of the line.

Because some of these constructs may occur in the middle of the line, we have to reinvoke `pp_line` recursively in order to avoid breaking our requirement that there's only one string, `$line`, left to process after leaving this chunk.

13a ⟨Process strings and comments 13a⟩≡

```
while ($line =~ m{
                (.*?)                       #1
                (                           #2
                  (                         #3 double-quoted string
                   " (\\.|[^\\\"])* "       #4
                  )
                 |
                  (                         #5 single-quoted string
                   ' (\\.|[^\\\']) '        #6
                  )
                 |
                  (?:                       #7 C comment
                    (?: /\*) (.*?) (?: \*/)
                  )
                 |
                  (                         #8 C++ comment
                   //.*
                  )
                )
                (.*)                        #9
               }x) {
  if ($3 || $5) {
    ⟨Quote the string in $3 or $5 and continue; $1 precedes it and $9 follows it 13b⟩
  }
  if ($7) {
    ⟨Quote the C comment in $7 and continue; $1 precedes it and $9 follows it 14b⟩
  }
  if ($8) {
    ⟨Quote the C++ comment in $8 and continue; $1 precedes it 15a⟩
  }
  die "comment/string confusion\n"; # no match, impossible
}
```

Uses `line` 8b.
This code is used in chunk 8b.

The `$3 || $5` trick grabs `$3` if it's non-empty, and otherwise grabs `$5`. We have to escape the string, and tell `escape` that we're inside a `\tt` environment.

13b ⟨Quote the string in $3 or $5 and continue; $1 precedes it and $9 follows it 13b⟩≡

```
$before = $1; $string = $3 || $5; $after = $9;
$string = escape($string, 1);
$preline .= pp_line ($before) . "{\\tt{}$string}";
$line = $after;
next;
```

Uses `escape` 14a, `line` 8b, `pp_line` 8b, and `preline` 8b.
This code is used in chunk 13a.

We might as well define `escape` now that we know how it's used. This code is completely disgusting. There's got to be a better way; please tell me what it is!

We have to do everything in very careful order to avoid, for example, creating a lot of backslashes and then trying to escape them. Since we can't do both left and right curly braces in the same pass, we change them to a special sequence and back again. It's all too gross.

14a ⟨Subroutines 4a⟩+≡

```
sub escape {
  my ($this_line) = shift;  my ($in_tt) = shift;
  if ($in_tt) {
    $this_line =~ s|\\|\001\\char92\002|g;
    $this_line =~ s|{|\001\\char123\002|g;
    $this_line =~ s|}|\001\\char125\002|g;
  } else {
    local ($bm) = "\\begin\001math\002";
    local ($em) = "\\end\001math\002";
    $this_line =~ s|\\|${bm}\\backslash${em}|g;
    $this_line =~ s!\|!${bm}\\mid${em}!g;
    $this_line =~ s|<|${bm}<${em}|g ;
    $this_line =~ s|>|${bm}>${em}|g ;
    $this_line =~ s|{|\001\\nwlbrace\002|g;
    $this_line =~ s|}|\001\\nwrbrace\002|g;
  }
  $this_line =~ s|_|\\_|g ;
  $this_line =~ s|#|\001\\char35\002|g ;
  $this_line =~ s|\001|{|g;
  $this_line =~ s|\002|}|g;
  return $this_line;
}
```

Defines:
`escape`, used in chunks 10c, 13b, and 15b.
Uses `bm` 10d and `em` 10d.

You may have noticed that we grabbed the '`/*`' and '`*/`' out of the C comment, so we can typeset those bits a little nicer.

This is the other place that we potentially do whitespace adjustment; if the '`-cw`' option has been specified, and thus `$compress_whitespace` is true, we insert some extra space between code and comments. The space is not inserted if there is no adjacent code, so a comment on a line by itself will be aligned properly.

14b ⟨Quote the C comment in `$7` and continue; `$1` precedes it and `$9` follows it 14b⟩≡

```
$before = $1; $comment = $7; $after = $9;
$preline .= pp_line ($before);
if ($compress_whitespace && ($before =~ /\S/)) { $preline .= "    " }
$preline .= "{\\commopen}" . pp_comment ($comment) . "{\\commclose}";
if ($compress_whitespace && ($after =~ /\S/)) { $preline .= "    " }
$line = $after;
next;
```

Uses `commclose` 14c, `commopen` 14c, `compress_whitespace` 9e, `line` 8b, `pp_comment` 15b, `pp_line` 8b, and `preline` 8b.
This code is used in chunk 13a.

14c ⟨Set up `$texdefs` 7c⟩+≡

```
$texdefs .=
  "\@literal \\def\\commopen{/$bm\\ast\\,$em}\n" .
  "\@literal \\def\\commclose{\\,$bm\\ast$em\\kern-.5pt/}\n";
```

Defines:
`commclose`, used in chunk 14b.
`commopen`, used in chunk 14b.
Uses `bm` 10d, `em` 10d, and `texdefs` 7a.

We know that nothing can possibly follow a C++ comment, so we don't have to worry about `$9`.

15a ⟨Quote the C++ comment in `$8` and continue; `$1` precedes it 15a⟩≡
```
$line = $1; $postline = $8;
$postline = pp_comment ($postline);
if ($compress_whitespace && ($line =~ /\S/)) { $postline = "    " . $postline }
next;
```
Uses `compress_whitespace` 9e, `line` 8b, `postline` 8b, and `pp_comment` 15b.
This code is used in chunk 13a.

Comments are made tricky by the fact that they can reference code with the `[[...]]` construct. If we see one, we prettyprint the quote and continue recursively with the rest of the comment (a different approach from the iterative technique used in `pp_line`). `$begcomm` and `$endcomm` get us into "comment mode" from "code mode," and out again.

What we do exactly depends on whether comments are passed directly to TEX or are meant to be printed verbatim. In the former case, we have to temporarily exit code mode; in the latter, we need to make sure that TEX doesn't interpret anything that it normally would.

15b ⟨Subroutines 4a⟩+≡
```
sub pp_comment {
  my ($this_comment) = shift;
  my ($pre, $code, $post);
  if ($this_comment =~ /(.*?)\[\[(.*?)\]\](?!\])(.*)/) {
    $pre = $1; $code = $2; $post = $3;
    if (defined $tex) {
      return $begcomm . $pre . $endcomm . pp_line ($code) . pp_comment ($post);
    } else {
      return escape ($pre) . pp_line ($code) . pp_comment ($post);
    }
  } else {
    if (defined $tex) {
      return $begcomm . $this_comment . $endcomm;
    } else {
      return escape ($this_comment);
    }
  }
}
```
Defines:
`pp_comment`, used in chunks 14b and 15a.
Uses `begcomm` 15e, `endcomm` 15e, `escape` 14a, and `pp_line` 8b.

`$begcomm` and `$endcomm` may be used, depending on a user option. If the user has specified the '`-tex`' option, then comments are typeset by TEX; otherwise they are printed verbatim. Verbatim is the default, so that if you have a big program and run it through `dpp` for the first time, you won't have any nasty surprises because you used dollar signs or something.

15c ⟨If `$option` is a valid option, process it and continue 6c⟩+≡
```
$tex = 1, next if ($option =~ /^-tex/);
```
Uses `option` 2b.

15d ⟨Set up `$texdefs` 7c⟩+≡
```
$texdefs .=
  "\@literal \\def\\begcomm{\\begingroup\\maybehbox\\bgroup\\setupmodname}\n" .
  "\@literal \\def\\endcomm{\\egroup\\endgroup}\n";
```
Uses `begcomm` 15e, `endcomm` 15e, and `texdefs` 7a.

15e ⟨Initialize things 2c⟩+≡
```
if (defined $tex) {
  $begcomm = "\\begcomm{}"; $endcomm = "\\endcomm{}";
}
```
Defines:
`begcomm`, used in chunk 15.
`endcomm`, used in chunk 15.

# 3 Appendices

## Chunk list

⟨Convert `@keywords` to `%is_keyword` 3e⟩
⟨If `$option` is a valid option, process it and continue 6c⟩
⟨If the first token is a hex number, process it and continue 9b⟩
⟨If the first token is a number, process it and continue 9c⟩
⟨If the first token is a word, process it and continue 10a⟩
⟨If the first token is punctuation, process it and continue 10b⟩
⟨If the first token is whitespace, process it and continue 9d⟩
⟨Ignore this line if it's a `%keyword` line 6d⟩
⟨Initialize the keyword list `@keywords` 3b⟩
⟨Initialize things 2c⟩
⟨Make a first pass through the input, collecting keywords and associating chunks with their roots 3c⟩
⟨Make a second pass through the input, producing output 5e⟩
⟨Make sure that `STDIN` is seekable 2d⟩
⟨Prettyprint part of a code markup line if necessary, adding it to `$extra` 8a⟩
⟨Prettyprint part of a documentation line if necessary 6e⟩
⟨Print out special LaTeX code if appropriate 7b⟩
⟨Print out `$text` and `$extra`, and reset them 7e⟩
⟨Process and print a line of code 7d⟩
⟨Process and print a line of documentation 6a⟩
⟨Process parental information 4c⟩
⟨Process preprocessor lines 10c⟩
⟨Process strings and comments 13a⟩
⟨Process this line for parental information 4b⟩
⟨Quote the C++ comment in `$8` and continue; `$1` precedes it 15a⟩
⟨Quote the C comment in `$7` and continue; `$1` precedes it and `$9` follows it 14b⟩
⟨Quote the string in `$3` or `$5` and continue; `$1` precedes it and `$9` follows it 13b⟩
⟨Read options 2b⟩
⟨Remove a token from the beginning of `$line`, process it, and append it to `$preline` 9a⟩
⟨Reset for the second pass 3f⟩
⟨Scan input, adding keywords to `@keywords` and discovering parents of chunks 3d⟩
⟨Set `$incode` to 1 and enter "code mode" if appropriate 6b⟩
⟨Set `$root` based on `$c` 5b⟩
⟨Set `$root` to the root chunk of `$chunk`, and put unsettled chunks on the ancestral path in `@chunks_to_settle` 5a⟩
⟨Set the ancestor of each chunk in `@chunks_to_settle` to `$root` and settle it 5c⟩
⟨Set up `$texdefs` 7c⟩
⟨Subroutines 4a⟩
⟨dpp 2a⟩

## Index