Using the `clrscode3e` Package in LaTeX 2ε

Thomas H. Cormen
thc@cs.dartmouth.edu

January 27, 2010

1 Introduction

This document describes how to use the `clrscode3e` package in LaTeX 2ε to typeset pseudocode in the style of *Introduction to Algorithms*, Third edition, by Cormen, Leiserson, Rivest, and Stein (CLRS 3/e) [1]. You use the commands[1] in the same way we did in writing CLRS 3/e, and your output will look just like the pseudocode in the text.

2 Setup

To get the `clrscode3e` package, download http://www.cs.dartmouth.edu/~thc/clrscode/clrscode3e.sty, and put it where it will be found when you run LaTeX 2ε. To use the package, include the following line in your source file:

```
\usepackage{clrscode3e}
```

The `clrscode3e` package itself includes the line

```
\RequirePackage{graphics}   % needed for \scalebox command
```

This line is necessary in order to get the right spacing for the == symbol that we use for equality tests. Therefore, you will need to have the `graphics` package installed and available on your system.

3 Typesetting names

Pseudocode in CLRS 3/e uses four types of names: identifiers, procedures, constants, and fixed functions. We provide commands `\id`, `\proc`, `\const`, and `\func` for these names. Each of these commands takes one argument, which is the name being typeset. These commands work both in and out of math mode. When used in math mode, and when the name given as an argument contains a dash, the dash is typeset as a hyphen rather than as a minus sign.

[1] We use the term "command" rather than "macro" throughout this document, though "macro" would work just as well.

Identifiers: Use identifiers for variable names of more than one character. When a variable name is just a single character, e.g., the identifier j in line 1 of INSERTION-SORT on page 18, we just typeset it in math mode rather than using the `\id` command: `$j$`.

Do not typeset identifiers consisting of two or more characters, e.g., the variable *key* in line 2 of INSERTION-SORT, in this way. (See page 51 of Lamport [2].) Although LaTeX 2ε provides the `\mathit` command for typesetting multiletter identifiers, use our `\id` command instead: `\id{key}`, rather than `\mathit{key}` or—horrors!—`$key$`. Since the `\id` command may be used both in and out of math mode, the source text

```
Line~5 uses the variable \id{key} in the test $A[i] > \id{key}$.
```

will produce

> Line 5 uses the variable *key* in the test $A[i] > key$.

To see how a dash turns into a hyphen, consider line 1 of FIND-MAX-CROSSING-SUBARRAY on page 71. It contains the variable *left-sum*. Typesetting this variable name by `\id{left-sum}` produces a hyphen in the identifier, but typesetting it by `\mathit{left-sum}` would produce $left - sum$, with a minus sign—rather than a hyphen—in the identifier.

Procedures: For procedure names, use the `\proc` command. It typesets procedure names in small caps, and dashes (which occur frequently in our procedure names) are typeset as hyphens. Thus, the source `\proc{Insertion-Sort}` produces INSERTION-SORT. Since you can use the `\proc` command both in and out of math mode, the source text

```
We call \proc{Insertion-Sort} with an array $A$, so that the
call is $\proc{Insertion-Sort}(A)$.
```

will produce

> We call INSERTION-SORT with an array A, so that the call is INSERTION-SORT(A).

Constants: We typeset constants such as NIL, TRUE, and RED in small caps with the `\const` command, e.g., `\const{nil}`, `\const{true}`, and `\const{red}`. The `\const` command typesets a dash within a constant name as a hyphen, so that, as on page 409, `\const{no-such-path}` will produce NO-SUCH-PATH.

Fixed functions: We typeset the names of fixed functions in plain old roman with the `\func` command, e.g., level and out-degree. By a "fixed function," we mean a function that is a specific, given function. For example, the sin function is typically typeset in roman; $\sin x$ looks right, but wouldn't $sin\ x$ look strange? Yet, on page 44, $\Theta(g(n))$ looks right, but $\Theta(\mathrm{g}(n))$ would look wrong, since g is a variable that stands for any one of a number of functions.

As with the other commands for names, a dash within a function name will typeset as a hyphen, so that `\func{out-degree}` will produce out-degree rather than out $-$ degree. Note that LaTeX 2ε provides commands for many fixed functions, such as sin and log; Table 3.9 on page 44 of Lamport [2] lists these "log-like" functions.

4 Typesetting object attributes

In the first two editions of the book, we used square brackets for object attributes. For example, we represented the length of an array A by *length*$[A]$. Based on requests from readers, we switched to the object-like dot-notation in the third edition, so that we now denote the length of array A by $A.length$.

You might think that you could typeset $A.length$ by `$A.\id{length}$`, but that would produce $A.length$, which has not quite enough space after the dot. Therefore, we created a set of commands to typeset object attributes. Each one may be used either in or out of math mode.

Most of the time, we use the `\attrib` command, which takes two arguments: the name of the object and the name of the attribute. Let's make a couple of definitions:

- An ***i-string*** is a string that you would use in an `\id` command, typically one or more non-Greek letters, numerals, or dashes.
- An ***x-string*** is a string that you would not use in an `\id` command, typically because it has a subscript or one or more Greek letters.
- As a special, and very common, case, a single non-Greek letter can count as either an i-string or an x-string.

The `\attrib` command works well when the object name is an x-string and the attribute name is an i-string. For example, to produce $A.length$, use `\attrib{A}{length}`. Here, we treat the object name, A, as an x-string. The attribute name, *length*, is of course an i-string.

If all your objects are x-strings and all your attributes are i-strings, then the `\attrib` command will be all you need. We provide several other commands for other situations that arose when we produced CLRS 3/e.

The four basic attribute commands are `\attribxi`, `\attribxx`, `\attribii`, and `\attribix`. Each takes two arguments: the object name and the attribute name. The last two letters of the command name tell you what type of strings the arguments should be. The next-to-last letter tells you about the object name, and the last letter tells you about the attribute name. `i` indicates that the argument will be treated as an `\id`, in which case the command calls the `\id` command and also puts the right amount of space between the argument and the dot.

- You may use `\attribxi` precisely when you would use `\attrib`. In fact, `\attrib` is just a call to `\attribxi`.
- Use `\attribxx` when both the object name and attribute names are x-strings. For example, you would use `\attribxx` if the attribute name has a subscript, so that to produce $y.c_i$, you would use `\attribxx{y}{c_i}`. Another situation in which you would use `\attribxx` is when the attribute name is a Greek letter: to produce $v.\pi$, use `\attribxx{v}{\pi}`.
- If both the object name and attribute name are i-strings, then you should use `\attribii`. For example, `\attribii{item}{key}` produces $item.key$, and `\attribii{prev-item}{np}` produces $prev\text{-}item.np$.
- If the object name is an i-string and the attribute name is an x-string, then use `\attribix`. (We never had this situation arise in CLRS 3/e.) But if we had wanted to produce $item.\pi$, we would have used `\attribix{item}{\pi}`.

For convenience, the `clrscode3e` package also contains commands for cascading attributes, such as $x.left.size$. These commands string together calls to the appropriate `\attribxi` and `\attribxx` commands. The number of arguments they take depends on how many attributes you are stringing together.

- When you have two attributes, use `\attribb`, which takes an object name and two attribute names: `\attribb{x}{left}{size}` produces $x.left.size$. This command assumes that the object name is an x-string and both attribute names are i-strings.
- For three attributes, use `\attribbb`, which takes an object name (an x-string) and three object names (i-strings): to produce $y.p.left.size$, use `\attribbb{y}{p}{left}{size}`.
- For four attributes, use `\attribbbb`, which is like `\attribbb` but with one additional attribute tacked on. We never needed to use this command in CLRS 3/e.
- The `\attribbxxi` command is for one level of cascading where the first attribute given is an x-string. For example, `\attribbxxi{x}{c_i}{n}` produces $x.c_i.n$.

If your cascading attributes do not fit any of these descriptions, you'll have to roll your own command from the `\attribxx` and `\attribxi` (or `\attrib`) commands. For example, suppose you want to produce $x.left.key_i$. Because it has a subscript, key_i is an x-string, and so you should not use `\attribb`. Instead, use `\attribxx{\attribxi{x}{left}}{\id{key_i}}`. (You could replace the call of `\attribxi` by a call of `\attrib`.) Note that this call treats key_i as an attribute of $x.left$, which is correct, rather than treating $left.key_i$ as an attribute of x, which is not correct.

Edges of a graph can have attributes, too, and the `clrscode3e` package provides two commands for attributes of edges. These commands assume that the edges are of the form (u, v), where the vertices u and v are x-strings. They take three parameters: the two vertices that define the edge and the name of the attribute.

- When the attribute name is an i-string, use `\attribe`. For example, to produce $(u, v).c$, use `\attribe{u}{v}{c}`.
- When the attribute name is an x-string, use `\attribex`. For example, to produce $(u, v).c'$, use `\attribex{u}{v}{c'}`.

5 Miscellaneous commands

The `clrscode3e` package contains three commands that don't really fit anywhere else, so let's handle them here. All three must be used in math mode.

- We denote subarrays with the ".." notation, which is produced by the `\twodots` command. Thus, the source text `$A[1 \twodots j-1]$` will produce $A[1 \mathinner{..} j-1]$.
- We use the `\gets` command for the assignment operator. For example, line 4 of INSERTION-SORT on page 18 is `$i \gets j - 1$`, producing $i = j - 1$.
- We use the `\isequal` command to test for equality with the == symbol. For example, line 1 of FIND-MAXIMUM-SUBARRAY on page 72 contains the test *high* == *low*, which we get by typesetting `$\id{high} \isequal \id{low}$`.

You might wonder why we bother with the `\gets` command when we could just typeset an equals sign directly. The answer is that in the first two editions of *Introduction to Algorithms*, we used a different symbol (a left arrow) for the assignment operator, and it made sense to use a command for that. Many readers told us that they preferred to use an equals sign for assignment—as many programming languages use—and so we made this change for the third edition. But it's a good idea to continue using the `\gets` command so that we can easily change our assignment operator should we desire to do so in the future.

Once we decided to use the equals sign for assignment, we could no longer use it for equality tests. We created the `\isequal` command for equality tests, and we decided to base it on the double equals sign used for equality tests in C, C++, and Java. Typesetting it as `==` in math mode produces $==$, which is too wide for our tastes. Our `\isequal` command calls the `\scalebox` command from the `graphics` package to narrow the symbol, and it puts a nice amount of space between the equals signs: $\mathrel{==}$.

6 The `codebox` environment

We typeset pseudocode by putting it in a `codebox` environment. A `codebox` is a section of code that does not break across pages.

Contents of a `codebox`

Each procedure should go in a separate `codebox`, even if you have multiple procedures appearing consecutively. The only possible reason I can think of to put more than one procedure in a single `codebox` is to ensure that the procedures appear on the same page. If you really need your procedures to appear on the same page, then you should consider using other means in LaTeX 2_ε, such as the `minipage` environment. Moreover, if you have written your procedures so that they have to appear on the same page, you should probably be asking yourself whether they are too interdependent.

The typical structure within a `codebox` is as follows. Usually, the first line is the name of a procedure, along with a list of parameters. (Not all `codebox`es include procedure names; for example, see the pseudocode on page 343 of CLRS 3/e.) After the line containing the procedure name come one or more lines of code, usually numbered. Some of the lines may be unnumbered, being continuations of previous lines. Lines are usually numbered starting from 1, but again there are exceptions, such as the pseudocode on page 343.

Using `\Procname` to name the procedure

The `\Procname` command specifies the name of the procedure. It takes as a parameter the procedure name and parameters, typically all in math mode. `\Procname` makes its argument flush left against the margin, and it leaves a little bit of extra space below the line. For example, here is how we typeset the INSERTION-SORT procedure on page 18:

```
\begin{codebox}
\Procname{$\proc{Insertion-Sort}(A)$}
\li \For $j \gets 2$ \To $\attrib{A}{length}$
\li     \Do
            $\id{key} \gets A[j]$
\li         \Comment Insert $A[j]$ into the sorted sequence
                $A[1 \twodots j-1]$.
\li         $i \gets j-1$
\li         \While $i > 0$ and $A[i] > \id{key}$
\li             \Do
                    $A[i+1] \gets A[i]$
\li                 $i \gets i-1$
                \End
\li         $A[i+1] \gets \id{key}$
        \End
\end{codebox}
```

Using \li and \zi to start new lines

To start a new, numbered line, use the `\li` command. To start a new, *un*numbered line, use the `\zi` command. Note that since a `codebox` is not like the `verbatim` environment, the line breaks within the source text do not correspond to the line breaks in the typeset output.

Tabs

I find that it is best to set the tab stops in the text editor to every 4 characters when typing in and displaying pseudocode source with the `clrscode3e` package. I use emacs, and to get the tabs set up the way I want them, my `tex-mode.el` file includes the line `(setq tab-width 4)`.

A `codebox` environment has a `tabbing` environment within it. Each tab stop gives one level of indentation. We designed the indentation so that the body of an **else** clause starts at just the right indentation. For the most part, you won't need to be concerned with tabs. The primary exception is when you want to include a comment at the end of a line of pseudocode, and especially when you want to include comments after several lines and you want the comments to vertically align.

If you used the `clrscode` package from the second edition of the book, you might notice different tabbing behavior when you port your pseudocode to the `clrscode3e` package. Where the `clrscode` package used two tab stops for each level of loop indentation, the `clrscode3e` package uses just one tab stop. We made this change in the `clrscode3e` package because the third edition eliminates the keyword **then** and left-aligns **else** with its corresponding **if**.

Note that the `tabbing` environment within a codebox has nothing to do with tabs that you enter in your source code; when you press the TAB key, that's the same as pressing the space bar in the eyes of LaTeX 2_ε.

Commands for keywords

As you can see from the source for INSERTION-SORT, the commands `\For` and `\While` produce the keywords **for** and **while** in boldface within a `codebox`.

Sometimes you want to include a keyword in the main text, as I have done in several places in this document. Use the `\kw` command to do so. For example, to produce the previous paragraph, I typed in the following:

```
As you can see from the source for \proc{Insertion-Sort}, the commands
\verb‘\For‘ and \verb‘\While‘ produce the keywords \kw{for} and
\kw{while} in boldface within a \texttt{codebox}.
```

The following commands simply produce their corresponding keywords, typeset in boldface: `\For`, `\To`, `\Downto`, `\By`, `\While`, `\If`, `\Return`, `\Goto` (which does not appear in CLRS 3/e, but you might wish to use), `\Error`, `\Spawn`, `\Sync`, and `\Parfor` (which produces the compound keyword **parallel for**). Although you could achieve the same effect with the `\kw` command (e.g., `\kw{for}` instead of `\For`), you will find it easier and more readable to use the above commands in pseudocode. The `\Comment` command simply produces the comment symbol *//*, followed by a space. To get the comment symbol without a following space, use `\CommentSymbol`. None of the above commands affects indentation.

Loops

The INSERTION-SORT example above shows typical ways to typeset **for** and **while** loops. In these loops, the important commands are `\Do` and `\End`. `\Do` increments the indentation level to start the body. Put `\Do` on a line starting with `\li`, but don't put either `\li` or `\zi` between the `\Do` command and the first statement of the loop body. Use `\li` or `\zi` in front of all loop-body statements after the first one. `\End` simply decrements the indentation level, and you use it to end any **for** or **while** loop, or otherwise decrement the indentation level.

In the first two editions of the book, the body of a **for** or **while** loop began with the keyword **do**. Responding to requests from readers to make pseudocode more like C, C++, and Java, we eliminated this keyword in the third edition.

As you can see from the above example, I like to place each `\Do` and `\End` on its own line. You can format your source text any way you like, but I find that the way I format pseudocode makes it easy to match up `\Do`-`\End` pairs.

If you want your **for** loop to decrease the loop variable in each iteration, use `\Downto` rather than `\To`. If you want the stride to be a value other than 1, use the `\By` command. For example, line 6 of ITERATIVE-FFT on page 917 is typeset as

```
\For $k \gets 0$ \To $n-1$ \By $m$
```

Loops that use the **repeat-until** structure are a bit different. We use the `\Repeat` and `\Until` commands, as in the HASH-INSERT procedure on page 270:

```
\begin{codebox}
\Procname{$\proc{Hash-Insert}(T,k)$}
\li $i \gets 0$
\li \Repeat
\li     $j \gets h(k,i)$
\li     \If $T[j] \isequal \const{nil}$
\li         \Then
                $T[j] \gets k$
\li             \Return $j$
\li         \Else
                $i \gets i+1$
            \End
\li \Until $i \isequal m$
\li \Error ‘‘hash table overflow’’
\end{codebox}
```

Note that the `\Until` command has an implied `\End`.

Typesetting if statements

As you can see from the above example of HASH-INSERT, we typeset **if** statements with the commands `\If`, `\Then`, `\Else`, and `\End`. In the first two editions of the book, the keyword **then** appeared in pseudocode, but—again mindful of requests from our readers to make our pseudocode more like C, C++, and Java—we eliminated the keyword **then** in the third edition. The `\Then` command remains, however, in order to indent the code that runs when the test in the **if** clause evaluates to TRUE.

We use `\End` to terminate an **if** statement, whether or not it has an **else** clause. For an example of an **if** statement without an **else** clause, here's the MERGE-SORT procedure on page 34:

```
\begin{codebox}
\Procname{$\proc{Merge-Sort}(A, p, r)$}
\li \If $p < r$
\li     \Then
            $q \gets \floor{(p + r) / 2}$
\li         $\proc{Merge-Sort}(A, p, q)$
\li         $\proc{Merge-Sort}(A, q+1, r)$
\li         $\proc{Merge}(A, p, q, r)$
        \End
\end{codebox}
```

The HASH-INSERT procedure above shows how to typeset an **if** statement that has an **else** clause. For a more complicated example, using nested **if** statements, here's the CASCADING-CUT procedure on page 519:

```
\begin{codebox}
\Procname{$\proc{Cascading-Cut}(H,y)$}
\li $z \gets \attrib{y}{p}$
\li \If $z \neq \const{nil}$
\li     \Then
            \If $\attrib{y}{mark} \isequal \const{false}$
\li             \Then $\attrib{y}{mark} \gets \const{true}$
\li             \Else
                    $\proc{Cut}(H,y,z)$
\li                 $\proc{Cascading-Cut}(H,z)$
                \End
        \End
\end{codebox}
```

Note that `\Then` and `\Else` always follow an `\li` command to start a new numbered line. As with the `\Do` command, don't put either `\li` or `\zi` between `\Then` or `\Else` and the statement that follows.

As you can see, I line up the `\End` commands under the `\Then` and `\Else` commands. I could just as easily have chosen to line up `\End` under the `\If` command instead. I also sometimes elect to put the "then" or "else" code on the same source line as the `\Then` or `\Else` command, especially when that code is one short line, such as in line 4 of CASCADING-CUT.

Sometimes, you need more complicated "**if**-ladders" than you can get from the `\Then` and `\Else` commands. The TRANSPLANT procedure on page 296 provides an example, and it uses the `\ElseIf` and `\ElseNoIf` commands:

```
\begin{codebox}
\Procname{$\proc{Transplant}(T, u, v)$}
\li \If $\attrib{u}{p} \isequal \const{nil}$
\li     \Then $\attrib{T}{root} \gets v$
\li \ElseIf $u \isequal \attribb{u}{p}{left}$
\li     \Then $\attribb{u}{p}{left} \gets v$
\li \ElseNoIf
        $\attribb{u}{p}{right} \gets v$
    \End
\li \If $v \neq \const{nil}$
\li     \Then $\attrib{v}{p} \gets \attrib{u}{p}$
        \End
\end{codebox}
```

For an **if**-ladder, use `\Then` for the first case, `\ElseNoIf` for the last case, and `\ElseIf` followed by `\Then` for all intermediate cases. You use `\ElseNoIf` like you use `\Else` in that it follows an `\li` command, you don't follow it with `\Then`, and, because it terminates an **if**-ladder, it's followed by `\End`. I usually line up the terminating `\End` with `\If`, the `\ElseIf` commands, and `\ElseNoIf`, but the way you line it up won't change the typeset output.

As another example, here is the SEGMENTS-INTERSECT procedure on page 1018:

```
\begin{codebox}
\Procname{$\proc{Segments-Intersect}(p_1, p_2, p_3, p_4)$}
\li $d_1 \gets \proc{Direction}(p_3, p_4, p_1)$
\li $d_2 \gets \proc{Direction}(p_3, p_4, p_2)$
\li $d_3 \gets \proc{Direction}(p_1, p_2, p_3)$
\li $d_4 \gets \proc{Direction}(p_1, p_2, p_4)$
\li \If $((d_1 > 0 \mbox{ and } d_2 < 0) \mbox{ or }
        (d_1 < 0 \mbox{ and } d_2 > 0))$ and
        \Indentmore
\zi     $((d_3 > 0 \mbox{ and } d_4 < 0) \mbox{ or }
        (d_3 < 0 \mbox{ and } d_4 > 0))$
        \End
\li     \Then \Return \const{true}
\li \ElseIf $d_1 \isequal 0$ and $\proc{On-Segment}(p_3, p_4, p_1)$
\li     \Then \Return \const{true}
\li \ElseIf $d_2 \isequal 0$ and $\proc{On-Segment}(p_3, p_4, p_2)$
\li     \Then \Return \const{true}
\li \ElseIf $d_3 \isequal 0$ and $\proc{On-Segment}(p_1, p_2, p_3)$
\li     \Then \Return \const{true}
\li \ElseIf $d_4 \isequal 0$ and $\proc{On-Segment}(p_1, p_2, p_4)$
\li     \Then \Return \const{true}
\li \ElseNoIf \Return \const{false}
    \End
\end{codebox}
```

This example also shows our first use of an unnumbered line: the second half of the tests on line 5. We use `\zi` to indicate that we're starting an unnumbered line.

Indentation levels

In the SEGMENTS-INTERSECT procedure, we indent the unnumbered line after line 5 by one level more than the line above it. We do so with the `\Indentmore` command. The `\End` command following the indented line decrements the indentation level back to what it was prior to the `\Indentmore`. If I had wanted to indent the line by two levels, I would have used two `\Indentmore` commands before the line and two `\End` commands afterward. (Recall that `\End` simply decrements the indentation level.)

Upon seeing the `\end{codebox}` command, the `codebox` environment checks that the indentation level is back to where it was when it started, namely an indentation level of 0. If it is not, you will get a warning message like the following:

```
Warning: Indentation ends at level 1 in codebox on page 1.
```

This message would indicate that there is one missing `\End` command. On the other hand, you might have one too many `\End` commands, in which case you would get

```
Warning: Indentation ends at level -1 in codebox on page 1.
```

Whenever the indentation level is nonzero upon hitting an `\end{codebox}` command, you'll get a warning telling you what the indentation level was.

Tabs and comments

Line 3 of INSERTION-SORT shows how to make a line that is only a comment. It's a little more tricky to put a comment at the end of a line of code. Using the tab command `\>`, explicitly tab to where you want the comment to begin and then use the `\Comment` command to produce the comment symbol. When several lines contain comments, you probably want them to align vertically. I just add tab characters, using a trial-and-error approach, until I am pleased with the result. For example, here's how we produced the KMP-MATCHER procedure on page 1005:

```
\begin{codebox}
\Procname{$\procdecl{KMP-Matcher}(T,P)$}
\li $n \gets \attrib{T}{length}$
\li $m \gets \attrib{P}{length}$
\li $\pi \gets \proc{Compute-Prefix-Function}(P)$
\li $q \gets 0$\>\>\>\>\>\>\>\>\>\Comment number of characters matched
\li \For $i \gets 1$ \To $n$\>\>\>\>\>\>\>\>\>\Comment scan the text from left to right
\li     \Do
            \While $q>0$ and $\Px{q+1}\ne \Tx{i}$
\li             \Do $q \gets \pi[q]$\>\>\>\>\>\>\>\Comment next character does not match
                \End
\li         \If $\Px{q+1} \isequal \Tx{i}$
\li             \Then $q \gets q+1$\>\>\>\>\>\>\>\Comment next character matches
                \End
\li         \If $q \isequal m$\>\>\>\>\>\>\>\>\Comment is all of $P$ matched?
\li             \Then
                    print ``Pattern occurs with shift'' $i-m$
\li                 $q \gets \pi[q]$\>\>\>\>\>\>\>\Comment look for the next match
                \End
        \End
\end{codebox}
```

All six comments align nicely.

We used the command `\RComment` to justify a comment against the right margin. We used this command only in the RB-INSERT-FIXUP procedure on page 316 and the RB-DELETE-FIXUP procedure on page 326. For example, here's how we typeset line 5 of RB-INSERT-FIXUP:

```
\li                     \Then
                            $\attribb{z}{p}{color}\gets \const{black}$
                                \RComment case 1
```

Referencing line numbers

The source files for CLRS 3/e contain no absolute references to line numbers. We use *only* symbolic references. The `codebox` environment is set up to allow you to place `\label` commands on lines of pseudocode and then reference these labels. The references will resolve to the line numbers. Our convention is that any label for a line number begins with `\li:`, but you can name the labels any way that you like.

For example, here's how we *really* wrote the INSERTION-SORT procedure on page 18:

```
\begin{codebox}
\Procname{$\proc{Insertion-Sort}(A)$}
\li \For $j \gets 2$ \To $\attrib{A}{length}$
                                        \label{li:ins-sort-for}
\li     \Do
            $\id{key} \gets A[j]$       \label{li:ins-sort-pick}
                                        \label{li:ins-sort-for-body-begin}
\li         \Comment Insert $A[j]$ into the sorted sequence
                $A[1 \twodots j-1]$.
\li         $i \gets j-1$               \label{li:ins-sort-find-begin}
\li         \While $i > 0$ and $A[i] > \id{key}$
                                        \label{li:ins-sort-while}
\li             \Do
                    $A[i+1] \gets A[i]$ \label{li:ins-sort-while-begin}
\li                 $i \gets i-1$       \label{li:ins-sort-find-end}
                                        \label{li:ins-sort-while-end}
                \End
\li         $A[i+1] \gets \id{key}$     \label{li:ins-sort-ins}
                                        \label{li:ins-sort-for-body-end}
        \End
\end{codebox}
```

Note that any line may have multiple labels. As an example of referencing these labels, here's the beginning of the first item under "Pseudocode conventions" on page 19:

```
\item Indentation indicates block structure.  For example, the body of
the \kw{for} loop that begins on line~\ref{li:ins-sort-for} consists
of lines
\ref{li:ins-sort-for-body-begin}--\ref{li:ins-sort-for-body-end}, and
the body of the \kw{while} loop that begins on
line~\ref{li:ins-sort-while} contains lines
\ref{li:ins-sort-while-begin}--\ref{li:ins-sort-while-end} but not
line~\ref{li:ins-sort-for-body-end}.
```

Setting line numbers

On rare occasions, we needed to start line numbers somewhere other than 1. Use the `setlinenumber` command to set the next line number. For example, in Exercise 24.2-2 on page 657, we want the line number to be the same as a line number within the DAG-SHORTEST-PATHS procedure on page 655. Here's the source for the exercise:

```
Suppose we change line~\ref{li:dag-sp-loop-begin} of
\proc{Dag-Shortest-Paths} to read

\begin{codebox}
\setlinenumber{li:dag-sp-loop-begin}
\li \For the first $\card{V}-1$ vertices, taken in topologically sorted order
\end{codebox}
Show that the procedure would remain correct.
```

The DAG-SHORTEST-PATHS procedure is

```
\begin{codebox}
\Procname{$\proc{Dag-Shortest-Paths}(G,w,s)$}
\li topologically sort the vertices of $G$      \label{li:dag-sp-topo-sort}
\li $\proc{Initialize-Single-Source}(G,s)$      \label{li:dag-sp-init}
\li \For each vertex $u$, taken in topologically sorted order
                                                \label{li:dag-sp-loop-begin}
\li     \Do
            \For each vertex $v \in \attrib{G}{Adj}[u]$
                                                \label{li:dag-sp-inner-begin}
\li             \Do $\proc{Relax}(u,v,w)$       \label{li:dag-sp-loop-end}
                \End
        \End
\end{codebox}
```

Even more rarely (just once, in fact), we needed to set a line number to be some other line number plus an offset. That was in the two lines of pseudocode on page 343, where the first line number had to be one greater than the number of the last line of LEFT-ROTATE on page 313. Use the `setlinenumberplus` command:

```
\begin{codebox}
\setlinenumberplus{li:left-rot-parent}{1}
\li $\attrib{y}{size} \gets \attrib{x}{size}$
\li $\attrib{x}{size} \gets \attribb{x}{left}{size}
        + \attribb{x}{right}{size} + 1
\end{codebox}
```

Here, the last line of LEFT-ROTATE has `\label{li:left-rot-parent}`.

Indenting long argument lists in procedure calls

You might find that you have to call a procedure with an argument list so long that the call requires more than one line. When this situation arises, it often looks best to align the second and subsequent lines of arguments with the first argument. The only place we did so was in the SUM-ARRAYS′ procedure in Problem 27-1 on page 805.

To get this style of alignment, use the `\Startalign` and `\Stopalign` commands, in concert with the `\>` command of LaTeX 2_ε. The `\Startalign` command takes an argument that is the text string that you wish to align just to the right of. Start each line that you want to indent with `\>`. Use the `\Stopalign` command to restore indentation to its state from before the `\Startalign` command.

The source code for SUM-ARRAYS′ shows how to use these commands:

```
\begin{codebox}
\Procname{$\procdecltag{Sum-Arrays}{$'$}(A, B, C)$}
\li $n \gets \attrib{A}{length}$
\li $\id{grain-size} \gets\ \ ?$ \>\>\>\>\>\>\>\Comment to be determined
\li $r \gets \ceil{n/\id{grain-size}}$
\li \For $k \gets 0$ \To $r-1$
\li     \Do
            \Spawn $\proc{Add-Subarray}(A, B, C, k\cdot\id{grain-size}+1,$
\Startalign{\Spawn $\proc{Add-Subarray}($}
\>                  $\min((k+1)\cdot\id{grain-size}, n))$
\Stopalign
        \End
\li \Sync
\end{codebox}
```

The second line of arguments in the call to ADD-SUBARRAY starts right under the first parameter, A, in the call.

7 Reporting bugs

If you find errors in the `clrscode3e` package, please send me email (thc@cs.dartmouth.edu). It would be best if your message included everything I would require to elicit the error myself.

The `clrscode3e.sty` file contains the following disclaimer:

```
% Written for general distribution by Thomas H. Cormen, March 2009.

% The author grants permission for anyone to use this macro package and
% to distribute it unchanged without further restriction.  If you choose
% to modify this package, you must indicate that you have modified it
% prior to your distributing it.  I don't want to get bug reports about
% changes that *you* have made!
```

I have enough trouble keeping up with my own bugs; I don't want to hear about bugs that others have introduced in the package!

8 Revision history

- 27 January 2010. Corrected an error in the documentation. The first line after a `\Repeat` command should begin with `\li`.
- 23 March 2009. Initial revision of document and code.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, third edition. The MIT Press, 2009.

[2] Leslie Lamport. *LaTeX: A Document Preparation System User's Guide and Reference Manual*. Addison-Wesley, 1993.