Using the `clrscode` Package in LaTeX 2_ε

Thomas H. Cormen
thc@cs.dartmouth.edu

June 11, 2003

1 Introduction

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

2 Setup

To get the `clrscode` package, download http://www.cs.dartmouth.edu/~thc/clrscode/clrscode.sty . To use the package, include the following line in your source file:

```
\usepackage{clrscode}
```

The `clrscode` package itself includes the line

```
\usepackage{latexsym}
```

This line is necessary in order to get the character $\rhd$ for comments in pseudocode. Therefore, you will need to have the `latexsym` package installed and available on your system.

3 Typesetting names

Pseudocode in CLRS 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.

Identifiers: Identifiers are used for variable and attribute names. When a variable name is just a single letter, e.g., the identifier j in line 2 of INSERTION-SORT on page 17, we just typeset it in math mode: `$j$`.

Identifiers consisting of two or more letters, e.g., the attribute *length* in the same line of INSERTION-SORT, should not be typeset in this way. (See page 51 of Lamport [2].) Although LaTeX 2_ε provides the `\mathit` command for typesetting multiletter identifiers, we use our `\id` command instead: `\id{length}`. We recommend that you use it, too. Since the `\id` command may be used both in and out of math mode, the source text

```
We use the \id{length} attribute to denote the length of an array,
e.g., $\id{length}[A]$.
```

will produce

> We use the *length* attribute to denote the length of an array, e.g., $\mathit{length}[A]$.

To see how a dash turns into a hyphen, consider line 3 of MAX-HEAPIFY on page 130. Its source contains the text `$l \leq \id{heap-size}[A]$`, which typesets as $l \leq \textit{heap-size}[A]$. Using `$l \leq \mathit{heap-size}[A]$` would produce $l \leq \mathit{heap-size}[A]$, 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 like NIL, TRUE, and RED in small caps with the `\const` command, e.g., `\const{nil}`, `\const{true}`, and `\const{red}`. I don't think that any of our constants have dashes in them, but the `\const` command would typeset a dash within a constant name as a hyphen, so that `$\const{red-and-black}$` will produce RED-AND-BLACK.

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 42, $\Theta(g(n))$ 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 $\mathrm{out} - \mathrm{degree}$. Note that LaTeX 2_ε provides commands for many fixed functions, such as sin and log; Table 3.9 on page 44 of [2] lists these "log-like" functions.

There is one other command that doesn't really fit anywhere else, so I'll mention it here. We denote subarrays with the ". ." notation, which is produced by the `\twodots` command. Thus, the source text `$A[1 \twodots j-1]$` will produce $A[1 \mathinner{\ldotp\ldotp} j-1]$. The `\twodots` command must be used in math mode.

4 The codebox environment

We typeset pseudocode by putting it in a `codebox` environment. A `codebox` is a section of code that will not break across pages (I hope).

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, there are other means in LaTeX 2_ε, such as the `minipage` environment, that you can use. 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 near the bottom of page 306 of CLRS.) After the line containing the procedure name comes one or more lines of code, usually numbered. Some of the lines may be unnumbered, being continuations of previous lines. In rare cases, when there is just one line of code per procedure, we don't bother numbering the line; see the PARENT, LEFT, and RIGHT procedures on page 128 for example. Lines are usually numbered starting from 1, but again there are exceptions, such as the pseudocode near the bottom of page 306.

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 17:

```
\begin{codebox}
\Procname{$\proc{Insertion-Sort}(A)$}
\li \For $j \gets 2$ \To $\id{length}[A]$
\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 to every 4 characters when typing in and displaying pseudocode source with the `clrscode` 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)`. For reasons I do not understand, sometimes emacs "forgets" this setting, and I have to do a little friendly persuasion.

A `codebox` environment has a `tabbing` environment within it. The tab stops come in pairs, in that each pair of tab stops gives one level of indentation. Tab stops are paired up so that when we typeset the keywords **then** and **else**, they have the correct vertical alignment. In other words, within each pair of tab stops, the first stop is where **then** and **else** begin, and the second stop completes a full level of 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. 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, there are commands `\For`, `\Do`, and `\While` that produce the keywords **for**, **do**, and **while**. `\Do` and some other commands also affect indentation.

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 a sentence that appeared two paragraphs ago, I typed in the following:

```
Tab stops are paired up so that when we typeset the keywords \kw{then}
and \kw{else}, they have the correct vertical alignment.
```

The following commands simply produce their corresponding keywords, typeset in boldface: `\For`, `\To`, `\Downto`, `\By`, `\While`, `\If`, `\Return`, `\Goto`, and `\Error`. 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. The `\Comment` command simply produces the comment symbol $\rhd$. None of the above commands affects indentation.

In **for** loops and **while** loops, the important commands are `\Do` and `\End`. `\Do` produces the keyword **do**, and it also increments the indentation level. `\End` simply decrements the indentation level, and it is the way to end any **for** or **while** loop or otherwise decrement the indentation level.

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

We also use `\End` to terminate an **if-then** or **if-then-else** construct. For an example of **if-then**, here's the MERGE-SORT procedure on page 32:

```
\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}
```

For a more complicated example, using **if-then-else**, here's the TREE-INSERT procedure on page 261:

```
\begin{codebox}
\Procname{$\proc{Tree-Insert}(T,z)$}
\li $y \gets \const{nil}$
\li $x \gets \id{root}[T]$
\li \While $x \neq \const{nil}$
\li     \Do
            $y \gets x$
\li         \If $\id{key}[z] < \id{key}[x]$
\li             \Then $x \gets \id{left}[x]$
\li             \Else $x \gets \id{right}[x]$
                \End
        \End
\li $p[z] \gets y$
\li \If $y = \const{nil}$
\li     \Then
            $\id{root}[T] \gets z$\>\>\>\>\>\>\>\>\Comment Tree $T$ was empty
\li     \Else
            \If $\id{key}[z] <\ id{key}[y]$
\li             \Then $\id{left}[y]\ gets z$
\li             \Else $\id{right}[y] \gets z$
                \End
        \End
\end{codebox}
```

As you can see, I like to 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 short.

The TREE-INSERT example also shows how we put a comment on the same line as code. Via the tab command `\>`, we explicitly tab to where we want the comment to begin and then use the `\Comment` command to produce the comment symbol. When there are several lines with 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 926:

```
\begin{codebox}
\Procname{$\proc{KMP-Matcher}(T,P)$}
\li $n \gets \id{length}[T]$
\li $m \gets \id{length}[P]$
\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 $\Px{q+1} \neq \Tx{i}$
\li             \Do $q \gets \pi[q]$\>\>\>\>\>\>\>\Comment Next character does not match.
                \End
\li         \If $\Px{q+1} = \Tx{i}$
\li             \Then $q \gets q+1$\>\>\>\>\>\>\>\Comment Next character matches.
                \End
\li         \If $q = m$\>\>\>\>\>\>\>\>\>\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.

For a **repeat** loop, use the `\Repeat` and `\Until` commands, as in the HASH-INSERT procedure on page 238:

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

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

Sometimes, you need more complicated "**if**-ladders" than you can get from the `\Then` and `\Else` commands. The RANDOMIZED-SELECT procedure on page 186 provides an example, and it uses the `\ElseIf` and `\ElseNoIf` commands:

```
\begin{codebox}
\Procname{$\proc{Randomized-Select}(A, p, r, i)$}
\li \If $p = r$
\li     \Then \Return $A[p]$
        \End
\li $q \gets \proc{Randomized-Partition}(A, p, r)$
\li $k \gets q - p +1$
\li \If $i = k$\>\>\>\>\Comment the pivot value is the answer
\li     \Then \Return $A[q]$
\li \ElseIf $i < k$
\li     \Then \Return $\proc{Randomized-Select}(A, p, q-1, i)$
\li \ElseNoIf \Return $\proc{Randomized-Select}(A, q+1, r, i-k)$
        \End
\end{codebox}
```

For an **if**-ladder, use `\Then` for the first case, `\ElseNoIf` for the last case, and `\ElseIf` for all intermediate cases. An **if**-ladder is terminated by `\End`. As another example, here is the SEGMENTS-INTERSECT procedure on page 937:

```
\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 $d_1 = 0$ and $\proc{On-Segment}(p_3, p_4, p_1)$
\li     \Then \Return \const{true}
\li \ElseIf $d_2 = 0$ and $\proc{On-Segment}(p_3, p_4, p_2)$
\li     \Then \Return \const{true}
\li \ElseIf $d_3 = 0$ and $\proc{On-Segment}(p_1, p_2, p_3)$
\li     \Then \Return \const{true}
\li \ElseIf $d_4 = 0$ 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

We also wish to 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.
```

Right justification

The final two commands, `\RComment` and `\Flushright`, are infrequently used and pertain to right justification. `\RComment` produces a comment that is flush against the right margin. For example, here is how we typeset the BINOMIAL-HEAP-UNION procedure on page 463:

```
\begin{codebox}
\Procname{$\proc{Binomial-Heap-Union}(H_1,H_2)$}
\li $H \gets \proc{Make-Binomial-Heap}()$
\li $\id{head}[H] \gets \proc{Binomial-Heap-Merge}(H_1,H_2)$
\li free the objects $H_1$ and $H_2$ but not the lists they point to
\li \If $\id{head}[H] = \const{nil}$
\li     \Then \Return $H$
        \End
\li $\id{prev-x} \gets \const{nil}$
\li $x \gets \id{head}[H]$
\li $\id{next-x} \gets \id{sibling}[x]$
\li \While $\id{next-x} \neq \const{nil}$
\li     \Do
            \If $(\id{degree}[x] \neq \id{degree}[\id{next-x}])$ or
\zi             \Flushright $(\id{sibling}[\id{next-x}] \neq \const{nil}$
                and $\id{degree}[\id{sibling}[\id{next-x}]] = \id{degree}[x])$
\li             \Then
                    $\id{prev-x} \gets x$
                        \RComment Cases 1 and 2
\li                 $x \gets \id{next-x}$
                        \RComment Cases 1 and 2
\li             \Else
                    \If $\id{key}[x] \leq \id{key}[\id{next-x}]$
\li                     \Then
                            $\id{sibling}[x] \gets \id{sibling}[\id{next-x}]$
                                \RComment Case 3
\li                         $\proc{Binomial-Link}(\id{next-x},x)$
                                \RComment Case 3
\li                     \Else
                            \If $\id{prev-x} = \const{nil}$
                                \RComment Case 4
\li                             \Then $\id{head}[H] \gets \id{next-x}$
                                    \RComment Case 4
\li                             \Else $\id{sibling}[\id{prev-x}] \gets
                                    \id{next-x}$
                                    \RComment Case 4
                                \End
\li                         $\proc{Binomial-Link}(x,\id{next-x})$
                                    \RComment Case 4
\li                         $x \gets \id{next-x}$
                                    \RComment Case 4
                        \End
                \End
\li         $\id{next-x} \gets \id{sibling}[x]$
        \End
\li \Return $H$
\end{codebox}
```

This procedure is also the only place in CLRS in which we used the `\Flushright` command. It puts the unnumbered line following line 10 flush against the right margin.

Referencing line numbers

In the source files for CLRS, there are 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 17:

```
\begin{codebox}
\Procname{$\proc{Insertion-Sort}(A)$}
\li \For $j \gets 2$ \To $\id{length}[A]$
                                            \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 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 594, we want the line number to be the same as a line number within the DAG-SHORTEST-PATHS procedure on page 592. 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 $v \in \id{Adj}[u]$
                                                \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 near the bottom of page 306, where the first line number had to be one greater than the number of the last line of LEFT-ROTATE on page 278. Use the `setlinenumberplus` command:

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

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

5 Reporting bugs

If you find errors in the `clrscode` 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 `clrscode.sty` file contains the following disclaimer:

```
% Written for general distribution by Thomas H. Cormen, June 2003.

% 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!

6 Revision history

- 11 June 2003. Initial revision of document and code.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, second edition. The MIT Press and McGraw-Hill, 2001.

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