

Typesetting figures for computer science

Andrew Mertz, William Slough and
Nancy Van Cleave

Abstract

Presentation of concepts from computer science can benefit from informative diagrams and figures. These include trees, graphs, logic circuits, stacks and stack frames, algorithms expressed with pseudocode, code listings and memory layout, for example. Producing these types of diagrams can sometimes be challenging. Fortunately, there are a number of L^AT_EX packages that can be used for this purpose. In this paper, we will illustrate the use of a few packages — including `tikz`, `bytefield`, `forest`, `drawstack`, and `listings` — that are well-suited for constructing high-quality figures for computer science.

1 Introduction

Popular wisdom dictates that *a picture is worth a thousand words*. Illustrations can be an indispensable aid to understanding new concepts, and never more so than in the classroom. The types of figures we most often use in our computer science courses fall mainly into one of four categories:

- systems: logic circuits, instruction set details, and stack frames
- algorithms/data structures: pseudocode, graphs, binary trees, and search trees
- theory of computation: automata, grammars, and parse trees
- discrete mathematics: graphs and trees

In all courses, there is a need for clearly presenting algorithms and code.

There are two approaches one can take: either generate a graphic with some third-party software, export it in an appropriate format and subsequently include it in the L^AT_EX source with the `graphicx` package [1], or generate the graphic with one of the “friends” of T_EX, such as METAPOST [6] or TikZ [16]. Where practical, we prefer the latter approach, since the results blend well with the surrounding typeset text and produce vector graphics, contributing to their quality. This quality derives from the fact that images generated in this manner use the same fonts as the typeset text and can utilize mathematical typesetting as needed.

Additionally, there are a number of packages designed for special purposes — such as typesetting trees, stacks, and graphs, to name a few. Some of the more recent packages on CTAN make use of TikZ,

adding a syntax layer to make them easier to use while preserving excellent typeset results.

The number of packages available from CTAN can be somewhat overwhelming to a L^AT_EX user with a specific task in mind. In a recent search, we found 35 packages for the topic *tree*. How does one choose from this embarrassment of riches? While references such as [4] and [10] certainly provide useful guidance they are snapshots in time, so a number of packages do not appear there.

Our purpose here is to provide examples of a number of typesetting tasks that are of interest to computer scientists. For each such task, we have identified packages we feel are especially appropriate. Although we are not providing tutorial introductions to these packages, we hope these examples may introduce readers to a few unfamiliar packages and provide motivation for further exploration.

2 Typesetting code

We often have a need to typeset code in a specific programming language, such as Python or Java. For such situations, we recommend the use of the `listings` package [5].

The `listings` package, introduced in 1996, is relatively mature and still enjoys current support. While perhaps best-known for its ability to produce “pretty-printed” output, verbatim-like results can also be produced. It provides support for more than 100 programming languages and dialects, including (L^A)T_EX, with the ability to define styles for yet others.

It is easy to get started with this package, knowing just two commands and one environment:

```
\lstset
\lstinputlisting
{lstlisting}
```

Appearance of typeset code is controlled with `\lstset`, which provides options for languages, colors, font sizes and styles, line numbering, and a host of other possibilities. Desired options are specified using a comma-separated list of key/value pairs, as follows:

```
\lstset{key1=value1,...}
```

An example of this is shown in Figure 1, where a number of such options are specified: `language`, `showstringspaces`, `columns`, etc. To typeset code, we can use either `\lstinputlisting` or `lstlisting`, the difference being where the code to be typeset is located. The command

```
\lstinputlisting{filename}
```

typesets the contents of the file *filename*, using the options previously requested. In contrast, code appears directly within an `lstlisting` environment:

<pre> 1 def gcd(p, q): 2 """ 3 Computes the greatest common divisor of 4 two nonnegative integers p and q using 5 Euclid's method. 6 """ 7 if (q == 0): 8 return p 9 else: 10 remainder = p % q 11 return gcd(q, remainder) </pre>	<pre> \lstset{language = Python, showstringspaces = false, columns = fullflexible, numbers = left, numberstyle = \tiny, frame = single} \lstinputlisting{gcd.py} </pre>
--	---

Figure 1: Typeset Python code using the listings package, producing pretty-printed output.

<pre> 1 def gcd(p, q): 2 """ 3 Computes the greatest common divisor of 4 two nonnegative integers p and q using 5 Euclid's method. 6 """ 7 if (q == 0): 8 return p 9 else: 10 remainder = p % q 11 return gcd(q, remainder) </pre>	<pre> \lstset{basicstyle = \ttfamily, columns = fullflexible, keepspaces = true, numbers = left, numberstyle = \tiny, frame = single} \lstinputlisting{gcd.py} </pre>
--	---

Figure 2: Typeset “anonymous” code with the listings package, producing verbatim-like output.

```

\begin{lstlisting}
code to typeset
\end{lstlisting}

```

Revisiting Figure 1, we see how Euclid’s algorithm for computing the greatest common divisor, as implemented in Python and stored in the file named `gcd.py`, can be typeset.

Most of the options specified here are evident from the typeset result: a single frame enclosing the code, tiny line numbers on the left, etc. Two of these options, however, are perhaps less obvious—`showstringspaces` and `columns`. Setting the first of these to `false` prevents the visible space (–) from appearing within any of the strings in the code. To understand the `columns` setting, it helps to know that there are two broad categories possible: fixed and flexible. A fixed column format adjusts the space between letters in an attempt to align columns; a flexible format makes no such attempt. Of the flexible formats available, we prefer `fullflexible`.

As a second example of the listings package, refer to Figure 2. Here, we did not specify a language, so no keywords are highlighted. To obtain a verbatim-like appearance, `basicstyle` is set to a monospace

font. Setting the `keepspace` option causes spaces which appear in the code to be respected which, in turn, preserves column alignment and indenting.

The listings package has many other options and advanced capabilities. For example, it is possible to include \TeX markup within a listing by providing “escaped” code, making it possible to blend mathematical comments with code.

3 Typesetting algorithms

To display algorithms in pseudocode, we think the style exhibited in [3] (famously known as *CLRS*, after the authors’ initials) is quite attractive, recognizing that its use requires some markup effort. For this, the `clrscode3e` package [2] can be used. An older version, `clrscode`, is also available; the 3e version matches the style used by the 3rd edition of *CLRS*.

Presenting an algorithm in this style is done using a `codebox` environment, as follows:

```

\usepackage{clrscode3e}
...
\begin{codebox}
algorithm, with markup
\end{codebox}

```

```

\Procname{\proc{Euclid-gcd}(p, q)$}
\zi \Comment Pre: $p$ and $q$ are two nonnegative integers.
\zi \Comment Post: $\proc{Euclid-gcd}(p, q) = \func{gcd}(p, q)$ .
\li \If $q \isequal 0$
\li \Then
    \Return $p$
\li \Else
    $\id{remainder} \gets p \bmod q$
\li \Return $\proc{Euclid-gcd}(q, \id{remainder})$
\End

```

Figure 3: Typesetting an algorithm with `clrscode3e`.

```

EUCLID-GCD( $p, q$ )
  // Pre:  $p$  and  $q$  are two nonnegative integers.
  // Post:  $\text{EUCLID-GCD}(p, q) = \text{gcd}(p, q)$ .
1  if  $q == 0$ 
2    return  $p$ 
3  else  $\text{remainder} = p \bmod q$ 
4    return  $\text{EUCLID-GCD}(q, \text{remainder})$ 

```

Figure 4: An algorithm presented with `clrscode3e`.

Figure 3 shows sample markup for presenting an algorithm with this package; the typeset result is in Figure 4. A few remarks will clarify some of the markup details being used here. Consult the documentation provided with this package for more complete information.

`Euclid-gcd` is the name of this algorithm, so it is being identified as such with the `\proc` (procedure) macro. In a similar way, `remainder` is an identifier indicated by `\id`. Each of `\li` and `\zi` commands cause new lines to begin, either numbered or not. The package also supplies miscellaneous commands such as `\gets` and `\isequal` which produce assignment and equality operators. Commands for control structures, such as the conditional shown here, mirror those found in modern programming languages.

4 Logic circuits

Circuits consisting of `and`, `or`, `not`, and `nand` gates are discussed in our systems course and also play a minor role in one of our general education mathematics courses. We have recently been looking at ways to produce diagrams of these types of circuits.

`TikZ` provides support for logic circuits and produces nice results. In this context, a circuit consists of a number of nodes and a collection of interconnections. The placement of the nodes can be specified either in absolute or relative coordinates. Using relative coordinates is handy, as circuits often consist

```

\tikzstyle{dot}=[fill,
  shape = circle,
  minimum size = 4pt,
  inner sep = 0pt,
  text height = 0pt,
  text depth = 0pt]
\tikzstyle{twoAnd}=[draw,
  and gate US,
  logic gate inputs=nn]
\tikzstyle{threeOr}=[draw,
  or gate US,
  logic gate inputs=nnn,
  anchor = input 2]

```

Figure 5: Some preliminary definitions for the majority circuit.

of components arranged either horizontally or vertically. Since the interconnections can be specified symbolically (e.g., “the output of the first and gate is connected to the first input of the or gate”), it is easy to stretch or compress the final drawing with just a minor change.

To illustrate, we will dissect some of the code details required to draw a circuit which computes a 3-input majority function. In the discussion that follows, refer to Figures 5 and 6. Since we want to use the U.S.-style gates found in the `circuits` library of `TikZ`, the following is needed in the preamble:

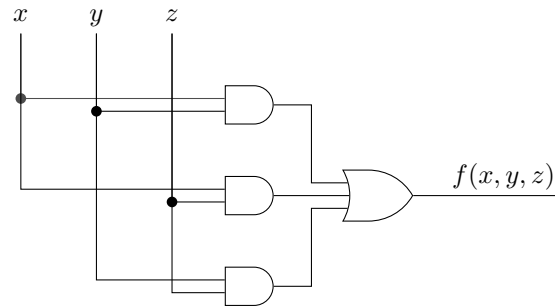
```

\usepackage{tikz}
\usetikzlibrary{circuits.logic.US}

```

In order to streamline some of the code for the circuit, Figure 5 introduces style names for each type of gate which will appear: a two-input `and`, and a three-input `or`. The specification

```
logic gate inputs = nn
```



```

\begin{tikzpicture}[x = 1cm, y = 1cm, text height = 1.5ex, text depth = 0.25ex]
  % Circuit has 3 inputs
  \node (x) at (0, 0) {$x$};
  \node (y) at (1, 0) {$y$};
  \node (z) at (2, 0) {$z$};

  % First column of AND gates
  \node[twoAnd] at ($(z) + (1, -1)$) (And1) {};
  \node[twoAnd] at ($(z) + (1, -2)$) (And2) {};
  \node[twoAnd] at ($(z) + (1, -3)$) (And3) {};

  % Connect inputs to the AND gates
  \draw (x) |- node[dot]{} (And1.input 1);
  \draw (y) |- node[dot]{} (And1.input 2);
  \draw (x) |- (And2.input 1);
  \draw (z) |- node[dot]{} (And2.input 2);
  \draw (y) |- (And3.input 1);
  \draw (z) |- (And3.input 2);

  % 3-input OR gate
  \node[threeOr] at ($(And2.output) + (1, 0)$) (Or1) {};

  % Connect AND output to OR inputs
  \draw (And1.output) -- ++(0.5, 0) |- (Or1.input 1);
  \draw (And2.output) -- (Or1.input 2);
  \draw (And3.output) -- ++(0.5, 0) |- (Or1.input 3);

  % Draw and label final output
  \draw (Or1.output) -- ++(2, 0) node[above left] {$f(x, y, z)$};
\end{tikzpicture}

```

Figure 6: A logic circuit which computes the majority function: $f(x, y, z) = \text{true}$ if two or more of its inputs are true; otherwise it is false.

indicates a two-input gate in which each input is “normal”; i.e., not inverted. The node style named `dot` will be used to produce a small, filled circle to indicate an electrical connection.

Moving on to Figure 6, let’s examine how this circuit is specified. The `tikzpicture` environment specifies x and y units of 1 cm each, which could easily be adjusted to obtain a stretched or compressed variant. The text height and depth which appears is

present to ensure that the three inputs, x , y , and z are typeset on the same baseline.

Within the body of the environment, the comments provide some guideposts for understanding how the circuit is drawn. A few additional remarks may be helpful. Inputs x , y , and z are placed at absolute coordinates, spaced one centimeter apart horizontally. The three `and` gates are placed relative to the z input: “over 1, down 1”, “over 1, down 2”,

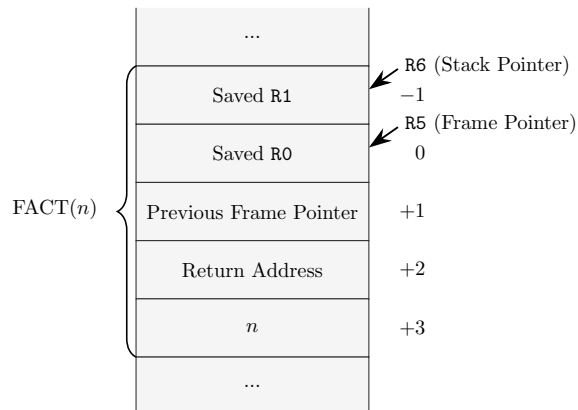


Figure 7: A simple stack frame produced by the drawstack package.

and “over 1, down 3”. The or gate is placed in a similar way.

The interconnections are simply vertical and horizontal line segments, specified with the `--` and `|-` line drawing capabilities of `TikZ`: a single line segment is produced by `--`, whereas `|-` draws an ell-shape consisting of a vertical segment followed by a horizontal one.

To illustrate specific input values, we could use a `draw` option to easily add colors (e.g., red for `false`, blue for `true`) to this circuit. For example, replacing the `\draw` commands with `\draw[red]` will change the color of the lines and dots.

5 Stacks and stack frames

To understand function calls at the machine-language level, visualization of stacks and stack frames within memory is key. The package `drawstack` [11] provides a means to illustrate memory with annotations and explicit links. Refer to Figure 7 for an example of a simple stack frame.

This package provides the `drawstack` environment. Each frame is enclosed within a pair of commands: `\startframe` and `\finishframe`. The argument provided to `\finishframe` specifies the brace label. The brace itself groups all of the cells within a given frame. In this first example, just one frame appears, but any number of frames may be produced.

Each component of a frame consists of a *cell*, an optional *comment*, and an optional cell *pointer*. The comment and pointer appear to the cell’s right. Cell comments are useful for memory addresses or offsets, for example. In this example, we are using

```
\begin{drawstack}
\startframe
\tikzset{>={Stealth[width = 2mm,
length = 3mm]}}
\cell{Saved \texttt{R1}}
\cellcom{-$-1$}
\cellptr{\texttt{R6} (Stack Pointer)}
\cell{Saved \texttt{R0}}
\cellcom{${\phantom{+}}0$}
\cellptr{\texttt{R5} (Frame Pointer)}
\cell{Previous Frame Pointer}
\cellcom{+1$}
\cellcom{+2$}
\cell{Return Address}
\cellcom{+3$}
\cell{n}
\cellcom{+3$}
\finishframe{FACT($n$)}
\end{drawstack}
```

`\tikzset` to specify an arrowhead of a customized size. Other features of `TikZ` can be used to add additional arrows, change colors, and so on. Figure 8 provides a glimpse of the possibilities.

6 Displaying fields of bits

In computer systems courses students are introduced to machine language instructions, written in binary. These instructions are subdivided into fields of bits and given mnemonic names. Similarly, memory maps, network protocols, and file formats all consist of fields of data. The package `bytefield` [12] can be used to show these layouts, and provides a wide variety of options to do so.

The `bytefield` environment may be used to create diagrams similar in format to Figure 9. The argument, 13 in this case, specifies the width of the figure in bits. The `bitwidth` option controls the amount of horizontal space given to each bit. Within this environment, `\bitbox` is used to add a field one or more bits wide in a single row. The command `\wordbox` adds a field that fills one or more rows deep. Double slashes are used to end a row in a `bytefield` diagram, much like a `tabular` environment.

Figure 10 illustrates how `bytefield` can be used to format a machine instruction; in this example, from a hypothetical computer called LC-3 [13]. The indices are added with `\bitheader`, while the `endianness` option reverses the order of the indexing.

Groups can be used to add labels that span multiple rows, and can be placed on the left or the right of the diagram. Groups do not have to nest properly, unlike most environments. Use the `leftcurly`

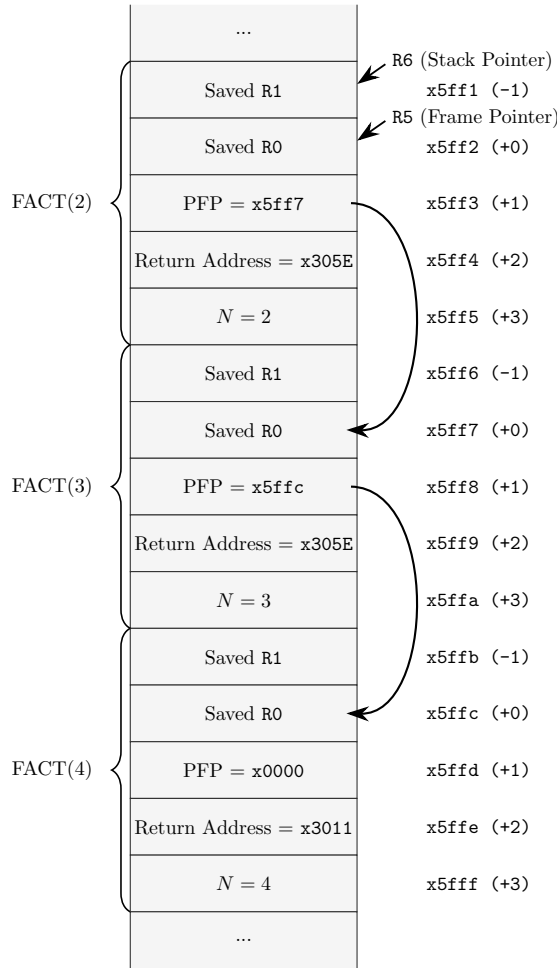


Figure 8: Linked stack frames corresponding to a recursive computation of FACT(4).

```
\begin{bytefield}[bitwidth = 1.5em]{13}
  \bitbox{1}{bit}
  \bitbox{4}{nybble}
  \bitbox{8}{byte}\\
  \wordbox{1}{one row}\\
  \wordbox{2}{two rows with
    additional text so that it will
    wrap around}
\end{bytefield}
```

bit	nybble	byte
one row		
two rows with additional text so that it will wrap around		

Figure 9: An example of basic bytefield commands, and its output.

```
\begin{bytefield}[bitwidth = 1.2em,
  leftcurly = .]{16}
  \bitheader[endianness = big]{0-15}\\
  \begin{leftwordgroup}{ADD}
    \bitbox{4}{0001}
    \bitbox{3}{DR}
    \bitbox{3}{SR1}
    \bitbox{1}{0}
    \bitbox{2}{00}
    \bitbox{3}{SR2}
  \end{leftwordgroup}
\end{bytefield}
```

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD	0001		DR		SR1		0		00		SR2					

Figure 10: Formatting a machine instruction using bytefield.

option to set the type of brace used to highlight the group. In the case of Figure 10, no brace was used.

When indices are shown it might be preferable to have each bit centered under its index. The `\bitboxes` command accomplishes this, as shown in Figure 11. The first argument to `bitboxes` specifies the number of “symbols” to place at each bit location. To remove the internal lines within a field, use `\bitboxes*`.

By default, `bytefield` vertically centers bit and word boxes without respecting the baseline of their contents. The `bytefield` documentation gives a technique for adjusting this, as shown in Figure 11, by using a `\raisebox` command within `\bytefieldsetup` to initialize `boxformatting`.

Extra space can be added between rows, similar to a `tabular` environment. The `bytefield` documentation also gives a technique for filling a `bitbox` to gray it out. However, it is not compatible with the technique given for aligning text on the baseline, so the box formatting needs to be reset, as also shown in Figure 12.

7 Automata

Central to the theory of computation is the concept of various types of automata, such as finite-state machines, pushdown machines, and Turing machines. These abstract computing devices are typically presented as directed graphs with labeled edges. These kinds of diagrams are good for understanding the *static* aspect of these machines.

How might we gain further appreciation of the *dynamic* aspect of automata — that is, the nature of how these machines process input strings? One way is to utilize a program like JFLAP [14], a Java-based



```

\newlength{\maxheight}
\setlength{\maxheight}{\heightof{W}}
\newcommand{\baselinealign}[1][\maxheight]
  {\centering\raisebox{0pt}{#1}[0pt]}

\begin{bytefield}[bitwidth = 1.2em,
  leftcurly = .]{16}
\bitheader[endianness = big]{0-15}\
\bytefieldsetup{boxformatting=
  \baselinealign}%
\begin{leftwordgroup}{BR}
  \bitboxes{1}{0000}
  \bitboxes*{1}{n z p}
  \bitbox{9}{PCoffset9}
\end{leftwordgroup}
\end{bytefield}

```

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								

Figure 11: Baseline alignment of bit boxes.

```

\begin{bytefield}[bitwidth = 1.5em,
  leftcurly = .]{16}
\bitheader[endianness = big]{0-15}\
\bytefieldsetup{boxformatting=
  \baselinealign}%
\begin{leftwordgroup}{BR}
  \bitboxes*{1}{0000}
  \bitboxes*{1}{n z p}
  \bitbox{9}{PCoffset9}
\end{leftwordgroup}\[1ex]

\bytefieldsetup{boxformatting=
  {\centering}}
\begin{leftwordgroup}{RSV}
  \bitboxes*{1}{1101}
  \bitbox{12}{\color{lightgray}
  \rule{\width}{\height}}
\end{leftwordgroup}
\end{bytefield}

```

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	n	z	p	PCoffset9								
RSV	1	1	0	1												

Figure 12: Adding space between two machine instructions and filling a bitbox with color.

formal language and automata package now often used in computer science education.

Using JFLAP, we can build automata using a convenient GUI interface, then explore their behavior on various input strings. In addition, we can experiment with constructions used in proofs, such as constructing an equivalent deterministic automaton given a non-deterministic one.

What options exist for incorporating a JFLAP-based automaton in a \LaTeX document? At present, the software exports in JPG, PNG, GIF, BMP and SVG formats. Unfortunately, these are less than ideal, as Figure 13 shows: fonts do not match those used in the document, “true” subscripts are not used, color may not be desired, and it is difficult to achieve accurate placement of the circular nodes, since JFLAP does not use a “snap to” grid for node layout.

Since TikZ has a library for drawing automata, we can imagine an export option from JFLAP which produces appropriate TikZ code which could then be included in a \LaTeX document. Such an option would be a vector-based format and allow for \TeX -based markup, thereby eliminating most of the undesirable effects of using one of the bitmap formats now available. The possibility of inaccurately placed nodes would continue to be a problem.

Although not directly incorporated into the JFLAP software, the script `jflap2tikz` [8] achieves the goal of presenting automata created with JFLAP using TikZ-based graphics. The TikZ code output by `jflap2tikz` for our example automaton is shown in Figure 14, and Figure 15 shows the resulting processed output. The TikZ code produced by `jflap2tikz` is human-readable and thus can be further edited, if desired. The script allows for a “snap to” style grid which can improve alignment of the nodes. A large grid spacing corresponds to a coarser grid. Figure 16 shows the result of using such a grid on the example finite-state machine.

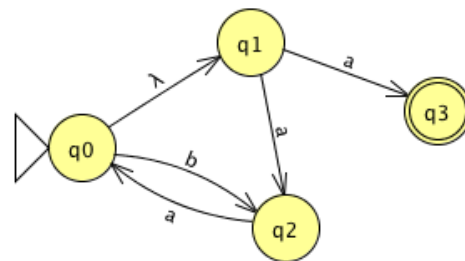


Figure 13: An example finite-state machine excerpted from [15], as exported by JFLAP using PNG format.

```

\begin{tikzpicture}[>={Stealth[width = 6pt, length = 9pt]},
    accepting/.style = {double distance = 2pt,
        outer sep = 1pt + \pgflinewidth},
    shorten >= 1pt,
    auto]
\draw (62pt, -114pt) node[state, initial, initial text =](0){$q_{0}$};
\draw (163pt, -51pt) node[state](1){$q_{1}$};
\draw (184pt, -162pt) node[state](2){$q_{2}$};
\draw (275pt, -92pt) node[state, accepting](3){$q_{3}$};
\path[->] (0) edge[bend left] node{b}(2);
\path[->] (2) edge[bend left] node{a}(0);
\path[->] (1) edge node{a}(2);
\path[->] (0) edge node{$\lambda$}(1);
\path[->] (1) edge node{a}(3);
\end{tikzpicture}

```

Figure 14: Result of processing the source file for Figure 13 with the `jflap2tikz` script (slightly edited for space).

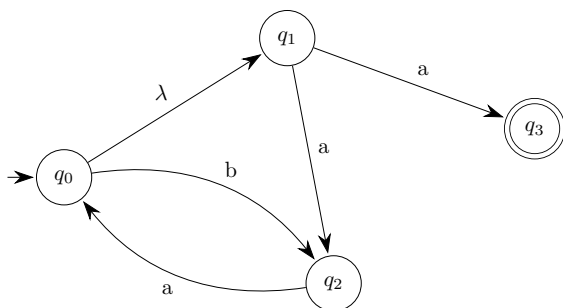


Figure 15: The finite-state machine from Figure 13, converted into TikZ and then processed as usual.

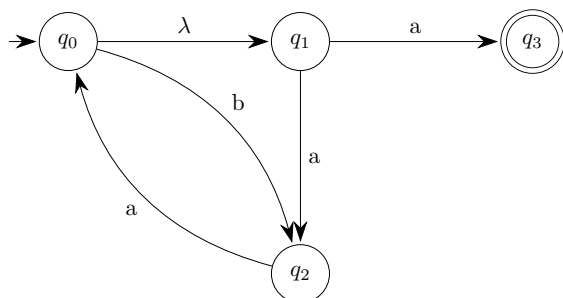


Figure 16: The finite-state machine from Figure 13, converted with a grid spacing of 100.

8 Trees

Forests and trees appear in a wide variety of forms in many areas of computer science. Drawing trees with GUI-style graphics software can be awkward and may yield imperfect results. Fortunately, there are a number of TeX-based approaches. One of these

involves the use of the `forest` package [17], which produces excellent results.

This package provides the `forest` environment, wherein a tree can be defined using *bracket notation*, a well-known syntax among linguists which captures the recursive nature of a tree. In this notation, a tree with a single root node r is represented by $[r]$. If a tree has more than a single node, then it has a root node r and n nonempty subtrees T_1, T_2, \dots, T_n . The bracket representation for this tree is $[r \text{ br}(T_1) \text{ br}(T_2) \dots \text{ br}(T_n)]$, where $\text{br}(T)$ is the bracket representation of tree T .

Figure 17 illustrates how this package can be used to draw a tree. In this example, 6 is the root with two subtrees, T_1 and T_2 , so the bracket notation for this tree is of the form $[6 \text{ br}(T_1) \text{ br}(T_2)]$. Using this same pattern to expand $\text{br}(T_1)$ and $\text{br}(T_2)$, we eventually obtain the bracketed form shown in the figure. Notice that spaces and newlines can be introduced to assist with readability and to help make clear the structure of the tree.

Another common structure in computer science is the *binary tree*, in which every node has at most two children, referred to as left and right children. The tree depicted in Figure 17 is not a binary tree, since it is not clear whether the leaf node 1 is a left child or a right child.

Indicating whether a node is a left or right child can be accomplished with the `phantom` option which reserves space for a node, but doesn't draw an edge to it. Phantom nodes may occur anywhere in the tree; when one is used at the root it produces a forest. Figure 18 shows two phantom nodes, one as the left child of 8, the other as the right child of 7.

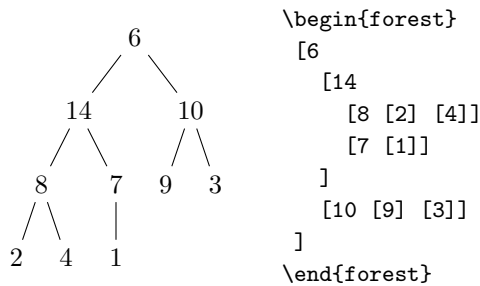


Figure 17: A tree produced by the forest package.

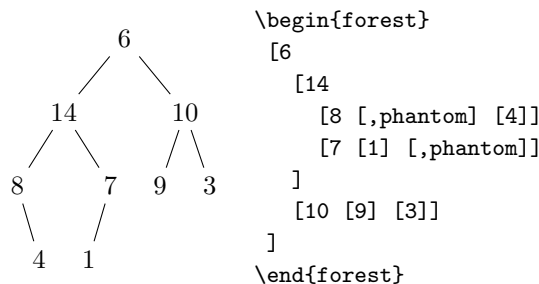


Figure 18: Use of phantom to force node alignment.

Trees are commonly drawn with circular nodes. This, too, is possible with the forest package since it is built upon TikZ. Thus, options from TikZ can be used to alter the result, and can be applied to individual nodes, a subtree, or the entire tree. To add circles to the tree of Figure 18, a first attempt would be to apply this option to the entire tree:

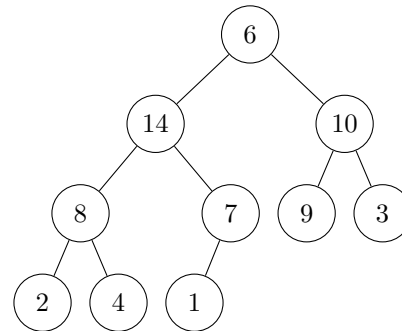
```
for tree = {draw, circle}
```

This would draw a circle at each node of the tree, but the size of each circle would depend on the dimensions of the text at each node — giving circles of varying sizes.

Specifying an appropriate minimum circle size, as shown in Figure 19, solves that problem, producing a tree with a uniform appearance.

It is also useful to be aware of the `fit` option. Figure 20 shows the same binary search tree formatted in two ways. On the left is the default result, also known as the `tight` fit, whereas the tree on the right shows the result with the `band` fit. Compact trees might be nice in many situations, but the wider tree on the right is the one typically encountered for search trees.

A red-black tree [3] is another type of binary tree, where the color of each node is important. Figure 21 displays a red-black tree. The text has been set to white, and the default color for nodes is black. The red nodes have the default color overridden with the option `fill = red` (printed in gray for this article).



```

\begin{forest}
for tree = {draw, circle,
node options = {minimum width = 5ex}}
[6
[14
[8 [2] [4]]
[7 [1] [,phantom]]
]
[10 [9] [3]]
]
\end{forest}

```

Figure 19: Uniform node size obtained by setting a minimum node width.

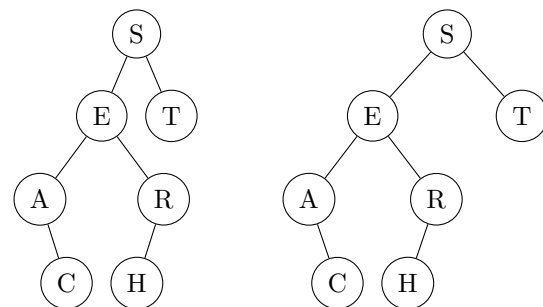
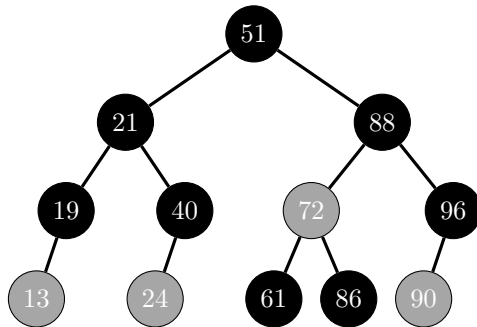


Figure 20: Compact tree on left (`fit = tight`); wider on right (`fit = band`).

9 Grammars and parse trees

Before looking at an example of a parse tree, we take a short detour into grammars. A context-free grammar is a formal system which describes a language as a set of rules.

The `syntax-mdw` package [18] may be used to typeset the syntax rules for a given language. A `grammar` environment is used, and produces nicely formatted output, as demonstrated in Figure 22. The `\alt` command is used to specify alternative rules, and when typeset produces the `|` symbol, pronounced “or”. Setting the value of `\grammarindent` determines the amount to indent each of the alternatives in the grammar definition.



```

for tree = {fit = band, circle, draw,
            fill = black, text = white,
            edge = {black, very thick}}
[51
 [21
  [19 [13, fill = red] [,phantom]]
  [40 [24, fill = red] [,phantom]]
 ]
 [88
  [72, fill = red [61][86]]
  [96 [90, fill = red] [,phantom]]
 ]
 ]

```

Figure 21: An example of a red-black tree.

```

<expr> ::= <expr> '+' <expr>
         | <expr> '-' <expr>
         | <expr> '*' <expr>
         | <expr> '/' <expr>
         | '(' <expr> ')'
         | '-'<expr>
         | 'id'

```

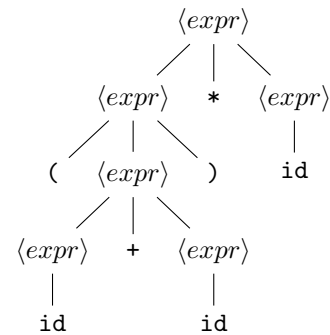
```

\setlength{\grammarindent}{5em}
\begin{grammar}
  <expr> ::= <expr> '+' <expr>
    \alt <expr> '-' <expr>
    \alt <expr> '*' <expr>
    \alt <expr> '/' <expr>
    \alt '(' <expr> ')'
    \alt '-'<expr>
    \alt 'id'
\end{grammar}

```

Figure 22: The syntax rules of a simple grammar.

A parse tree represents the syntactic structure of a string according to some context-free grammar. Given the grammar in Figure 22, a parse tree can be used to represent an expression such as $(a + b) * c$ as shown in Figure 23.



```

% Expression: (a+b)*c
\newcommand{\E}{$\langle expr \rangle$}
\newcommand{\id}{\texttt{id}}
\newcommand{\plus}{\texttt{+}}
\newcommand{\mult}{\texttt{*}}
\newcommand{\lpar}{\texttt{(}}
\newcommand{\rpar}{\texttt{)}}
\begin{forest}
[\E
 [\E
  [\lpar
  [\E
  [\E [\id]] [\plus] [\E [\id]]
 ]
 ]
 ]
 ]
 ]
 [\mult]
 [\E [\id]]
 ]
\end{forest}

```

Figure 23: A parse tree for the expression $(a + b) * c$, using the grammar from Figure 22.

10 Combinatorial graphs

In graph theory and similar courses, combinatorial graphs must be produced in great numbers. The package `tkz-graph` [7] provides a variety of styles and macros for creating high-quality representations of graphs. The documentation is available only in French, but is so abundantly illustrated with well-done examples that it is accessible to those with little or even no knowledge of French.

Figure 24 displays several of the options available when drawing graphs. To select the vertex style (from a list of ten possibilities, for example `Simple`, `Classic`, or `Shade`), set `vstyle` to your choice using the `\GraphInit` command.

`\SetUpVertex` can be used to set such options as the position of the label relative to the node (`Lpos`), the distance of the label from the node (`Ldist`), and whether the label is outside of the node (`LabelOut`).

```

\begin{tikzpicture}
  % Select vertex style
  \GraphInit[vstyle = Classic]
  % Indicate vertex color
  \SetUpVertex[FillColor = gray!30]
  % Set vertex size relative to label
  \renewcommand*\VertexInnerSep}{3pt}
  % Position of vertex labels
  \SetVertexLabelIn
  % Establish Line width
  \tikzset{EdgeStyle/.append style =
           {line width = 2pt}}

  % --- tkz-graph commands here ---
\end{tikzpicture}

```

Figure 24: Details of initializing values for a graph.

Alternately, macros such as `\VertexInnerSep` can be redefined. Also, several macros are provided to modify default options, for example, `\SetVertexLabelIn`. And finally, `\tikzset` can be used to modify both the vertex and edge styles.

Figure 25 illustrates the format of a simple graph using relative positioning to place the vertices. Vertex A is established, then all the other vertices are relative to A or the position of another vertex. It is of interest to note that the distance given to combined directions such as `\SOEA`, is applied in *both* directions. Thus `\SOEA[unit = 1](A){C}` results in vertex C being one unit south and one unit east of vertex A. Recall that this code resides in the `tikzpicture` environment following the code given in Figure 24. Hence, native TikZ commands such as

```
\draw[help lines](0, 0) grid (4, -4);
```

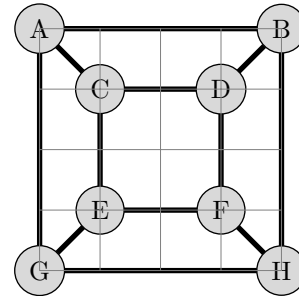
can be issued here.

An alternative way to place the vertices can be seen in Figure 26. Here the vertices are placed using absolute coordinates, and all edges can be included in the single `Edges` macro.

By utilizing `\tikzset`, it is possible to design a new vertex style, as shown in Figure 27. Here the shape, size, and color have been redefined.

It is also possible to modify the vertex style within the `\SetVertexSimple` macro, as shown in Figure 28, where the shape, fill color, size and line have all been specified.

As Figure 29 shows, creating directed edges is as simple as setting the edge style. To improve the default arrowheads, `\tikzset` is used to select the `Stealth` shape, with the width and height enlarged



```

% Two nested squares of vertices
\Vertex[A]          \EA[unit = 4](A){B}
\SO [unit = 4](B){H} \WE[unit = 4](H){G}
\SOEA[unit = 1](A){C} \EA[unit = 2](C){D}
\SO [unit = 2](D){F} \WE[unit = 2](F){E}
% Outer square
\Edge(A)(B)        \Edge(B)(H)
\Edge(H)(G)        \Edge(G)(A)
% Inner square
\Edge(C)(D)        \Edge(D)(F)
\Edge(F)(E)        \Edge(E)(C)
% Connect the squares
\Edge(A)(C)        \Edge(B)(D)
\Edge(H)(F)        \Edge(G)(E)
% Cause a grid to appear
\draw[help lines](0, 0) grid (4, -4);

```

Figure 25: A simple graph using relative positioning.

```

% Two nested squares of vertices
\Vertex[x = 0, y = 4]{A}
\Vertex[x = 0, y = 0]{G}
\Vertex[x = 1, y = 3]{C}
\Vertex[x = 1, y = 1]{E}
\Vertex[x = 3, y = 3]{D}
\Vertex[x = 3, y = 1]{F}
\Vertex[x = 4, y = 4]{B}
\Vertex[x = 4, y = 0]{H}

% All edges
\Edges(A, B, H, G, A, C, D, F,
       E, C, E, G, H, F, D, B)

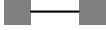
```

Figure 26: The same graph as Figure 25 using absolute placement of vertices.

to improve visibility. Since the edges are now directed, the order of vertices when creating edges becomes important.

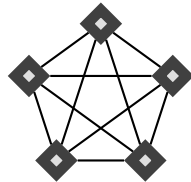
Simple changes can affect the appearance of a graph. For example, changing the vertex style to `Shade` produces a sophisticated-looking graph as seen in Figure 30.

```


\begin{tikzpicture}
  \SetVertexSimple
  \tikzset{VertexStyle/.style = {
    shape      = rectangle,
    fill       = gray,
    inner sep  = 0pt,
    outer sep  = 0pt,
    minimum size = 10pt}}
  \Vertex{A} \EA(A){B}
  \Edge(A)(B)
\end{tikzpicture}

```

Figure 27: Designing your own vertex style, version 1.



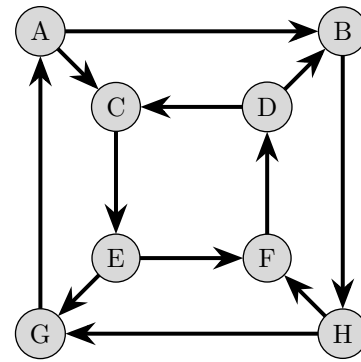
```

\begin{tikzpicture}[rotate = 18]
  \SetVertexSimple[Shape      = diamond,
                   FillColor = gray!25,
                   MinSize   = 12pt,
                   LineWidth = 4pt,
                   LineColor = black!75]
  \tikzset{VertexStyle/.append style = {
    inner sep = 0pt,
    outer sep = 2pt}}
  \Vertices{circle}{A, B, C, D, E}
  \Edges(A, B, C, D, E, A, C, E, B, D)
\end{tikzpicture}

```

Figure 28: Designing your own vertex style, version 2.

Figure 31 represents a flow network, with an augmenting path (of flow 4) highlighted. The source and sink nodes have been emphasized with different colors, and the augmenting path edges are wider than other edges and also of a different color. Observe that an edge may be curved using the `bend right` or `bend left` option when setting the edge style. Additional examples from the `tkz-graph` package can be seen in [9].

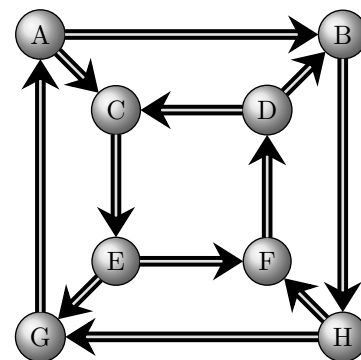


```

% Set arrow type and size
\tikzset{>={Stealth[width = 3mm,
                    length = 4mm]}}
% Set edge to arrow
\tikzset{EdgeStyle/.append style =
  {>,line width = 1.5pt}}
% Nested squares of vertices as before
% Outer square - clockwise
\Edges(A, B, H, G, A)
% Inner square - counterclockwise
\Edges(C, E, F, D, C)
% Connect the corners
\Edge(A)(C)
\Edge(D)(B)
\Edge(H)(F)
\Edge(E)(G)

```

Figure 29: Example of a directed graph.

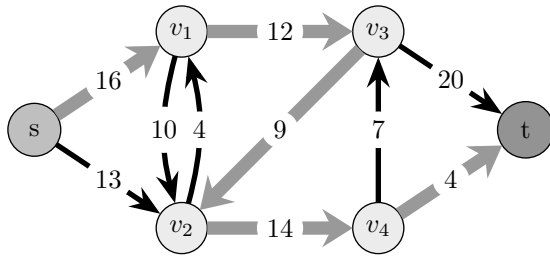


```

% Simple changes:
\GraphInit[vstyle = Shade]
\SetGraphShadeColor
  {gray!25} % ball
  {black}   % edge outline
  {gray!25} % inner edge
% vertices and edges as before

```

Figure 30: A change in vertex style. (Partial L^AT_EX code)



```
% Change vertex attributes
\SetUpVertex[FillColor = gray!50,
  InnerSep = 5pt]

% Change edge attributes
\tikzset{LabelStyle/.style =
  {shape = circle, inner sep = 2pt}}
\Edge[color = gray!80,
  lw = 5pt, label = 16](s)(v1)

% Add curve to edges
\tikzset{EdgeStyle/.append style =
  {bend right = 15}}
\Edge[lw = 2pt, label = 10](v1)(v2)
```

Figure 31: A sample flow network with changing attributes.

11 Summary

As educators in the field of computer science, we find ourselves challenged to produce a wide variety of figures and diagrams. Being able to replicate (or in some cases, exceed) the quality found in textbook presentations is a practical and intrinsically rewarding skill. Fortunately, the \TeX community has provided a wealth of resources which can be brought to bear on this problem. We hope that the examples and explanations provided in this paper will encourage others to explore these and other packages further.

References

- [1] David Carlisle. Guide to graphics in \LaTeX . <http://ctan.org/pkg/graphicx>.
- [2] Thomas Cormen. The `clrscode3e` package: Typesets pseudocode as in Introduction to Algorithms. <http://ctan.org/pkg/clrscode3e>.
- [3] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. The MIT Press, 2009.
- [4] Michel Goossens, Frank Mittelbach, Sebastian Rahtz, and Denis Roegel. *The \LaTeX Graphics Companion, 2nd Edition*. Addison-Wesley Professional, 2007.

- [5] Carsten Heinz, Brooks Moses, and Jobst Hoffmann. The listings package: Typeset source code listings using \LaTeX . <http://ctan.org/pkg/listings>.
- [6] John Hobby. MetaPost. <http://tug.org/metapost>.
- [7] Alain Matthes. The `tkz-graph` package: Draw graph-theory graphs. <http://ctan.org/pkg/tkz-graph>.
- [8] Andrew Mertz and William Slough. The `jflap2tikz` script: Convert JFLAP files to TikZ. <http://ctan.org/pkg/jflap2tikz>.
- [9] Andrew Mertz and William Slough. Graphics with PGF and TikZ. *TUGboat*, 28(1):91–99, 2007. <http://tug.org/TUGboat/tb28-1/tb88mertz.pdf>.
- [10] Frank Mittelbach, Michel Goossens, Johannes Braams, and David Carlisle. *The \LaTeX Companion, 2nd Edition*. Addison-Wesley Professional, 2004.
- [11] Matthieu Moy. The `drawstack` package: Draw execution stacks. <http://ctan.org/pkg/drawstack>.
- [12] Scott Pakin. The `bytefield` package: Create illustrations for network protocol specifications. <http://ctan.org/pkg/bytefield>.
- [13] Yale Patt and Sanjay Patel. *Introduction to Computing Systems: From bits & gates to C & beyond*. McGraw-Hill, 2003.
- [14] Susan H. Rodger. JFLAP: Java Formal Languages and Automata Package. <http://www.jflap.org>.
- [15] Susan H. Rodger and Thomas W. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Learning, 2006.
- [16] Till Tantau. The PGF package: Create PostScript and PDF graphics in \TeX . <http://ctan.org/pkg/pgf>.
- [17] Sašo Živanović. The `forest` package: Drawing (linguistic) trees. <http://ctan.org/pkg/forest>.
- [18] Mark Wooding. The `syntax-mdw` package: Typeset syntax descriptions. <http://ctan.org/pkg/syntax-mdw>.

◇ Andrew Mertz, William Slough
and Nancy Van Cleave
Department of Mathematics and
Computer Science
Eastern Illinois University
Charleston, IL 61920
`aemertz (at) eiu dot edu`,
`waslough (at) eiu dot edu`,
`nkvanleave (at) eiu dot edu`