

THE UNIVERSITY OF MELBOURNE  
Department of Computer Science and Software Engineering  
**433-255 Logic and Computation**  
Second Semester, 2002  
Project 1

*Due: r̄ap Thursday, 12 September 2002*

The objective of this project is to practice and assess your understanding of logic programming and Prolog. You will write a simple text formatter, which will take as input a term describing how the output should look, and will produce an attractively formatted text file and/or a postscript output file suitable for printing.

**Note well:** any questions regarding the specification for this project should be posted to the `cs.255` newsgroup or emailed to `ask255@cs.mu.oz.au`. Questions asked to the newsgroup are likely to be answered sooner.

### Format Specification Terms

The formatting language works by laying out “boxes” on the page. A box is just a rectangular area that will usually contain with some text. Boxes can be nested, and are laid out in various configurations, as described below.

A box is specified by a *box term*, which has one of the forms shown in Table 1. Figure 1 presents the meanings of some of these box terms diagrammatically for lists of three boxes; longer lists follow the same pattern. Where multiple alternatives are shown, the first that can be constructed without violating the page width limit should be selected. Font attributes and their values are shown in Table 2. For plain text output, all font attributes are ignored.

### Postscript Output

Postscript is a file format suitable for sending to a printer. It is a text-based format, but is very different than the text which appears on the page. In fact, postscript is a full-fledged programming language, providing an unlimited number of ways to produce identical looking output pages. For this project, however, we will be using very little of the power of postscript, and the output you produce will be rigidly specified; correctness will be judged by comparing the postscript files, not by comparing the printed pages produced.

Each postscript file produced must begin with these three lines:

```
%!PS-Adobe-3.0
%%Pages: (atend)
%%EndComments
```

and end with these three:

```
%%Trailer
%%Pages: n
%%EOF
```

where  $n$  is the total number of pages produced in this file (the number of `showpage` commands; see below), as a decimal integer.

Postscript output must be paginated. Each page must begin with:

```
%%Page: n n
```

where  $n$  is the page number, starting from 1 (yes, put it there twice), and must end with: `showpage`

(note no `%%` at the beginning). A new page must be started when the next line of output would appear below y coordinate 80.

Page positions are measured in “points” from the *lower* left corner of the page; a point is about .3528 millimeters. Allowing a reasonable margin around the page, the upper left corner of the writable area of the page is at (80,762) and the lower right is (515,80). Text appears mostly above and entirely to the right of the position it is written at. For this project, all text will be written in a 12 point font (the height of each line of text), so the first line of text should be written at position (80,750) (if it is to be left justified).

Each text to be output should be preceded by a command to move to the appropriate  $(x, y)$  coordinates for that text. This command has the form:

```
x y moveto
```

where  $x$  and  $y$  are the coordinates of the lower left corner of the text to be written, in points. Each line of text should be drawn 14 points below the previous one (this allows a little space between lines).

Following the `moveto` command, text is displayed with a command of the form:

```
(text) show
```

where *text* is the characters to be displayed. Note that any character other than a letter (upper or lower case), digit, or space must be preceded by a backslash (`\`) character. For example,

```
(hello\, world\.) show
```

would display “hello, world.” at the current position. Note that each `text` box should have a single `show` command in the postscript file; you must not merge the text from multiple `text` boxes into a single `show`, nor split one `text` box into multiple `shows`. Of course, a `text` box nested inside a `width` or `height` box will not have any corresponding `show`.

Immediately before each text is shown, the desired font must be selected in the postscript file. This is done by the command

```
/font findfont 12 scalefont setfont
```

where *font* is the font name, including the desired weight and slant. The fonts we will be

using for this project are:

Courier, Courier-Bold, Courier-BoldOblique, Courier-Oblique,  
 Helvetica, Helvetica-Bold, Helvetica-BoldOblique, Helvetica-Oblique,  
 Times-Roman, Times-Bold, Times-BoldItalic, Times-Italic.

For our purposes, we can consider *Oblique* to be synonymous with *Italic*. For example, when a *helvetica* family font with *slant italic* and weight *normal* is to be selected, you should issue the command

```
/Helvetica-Oblique findfont 12 scalefont setfont
```

Thus for each `text` box in the input, there should be a `moveto` line, a `setfont` line, and a `show` line, in that order, in the generated postscript file.

For postscript output, font attributes in inner boxes override the attributes of outer boxes. For example if outer box specifies font family *helvetica* and weight *bold*, and an inner box specifies *times italic*, text in the inner box will be written in *times bold italic*, “inheriting” *bold* from the outer box.

For *helvetica* and *times* fonts, different characters have different widths. You must take this into account when determining how wide a text string will be. The file `font.pl` in the student data area, and on the subject web page, contains a predicate for each font which indicates how wide, in points, each character is, based on its character code. The first argument of each of these predicates is a character code, and the second is the width of that character, in that font, in points. These widths are floating point numbers; to determine the width of a string, you should add the widths of all its characters, and then round up to the nearest whole point.

All text is to be output a line at a time in top-to-bottom order, with each line produced left-to-right. Each line of output specified above should be placed in a separate line in the output file. Text in the postscript file should be left-justified with a single space separating its parts; there should be no blank lines.

## The Program

You will write the following predicates:

`format_text(Box, Filename)` Lay out *Box* as text to a width of 78 characters, putting the output in the file named *Filename*. Ignore all font attributes, and consider each character to have width 1. Text begins in the leftmost column, and on the first line of the output.

`format_postscript(Box, Filename)` Same as `format_text/2`, except that the output is a postscript file, rather than text.

## Hints

1. You must develop and debug two separate predicates producing two entirely different output formats. This will be a much more manageable task if you share as much of

<i>Term</i>	<i>Description</i>
<code>empty</code>	A point, having exactly zero width and height. That is, nothing at all.
<code>text(<i>Atom</i>)</code>	The text of <i>Atom</i> , a Prolog atom.
<code>font(<i>Box</i>,<i>Attribs</i>)</code>	The contents of the specified <i>Box</i> , displayed with the default font attributes modified by <i>Attribs</i> , a list of <i>Attribute=Value</i> pairs. Valid attributes are listed in Table 2.
<code>h(<i>Boxes</i>,<i>Sep</i>)</code>	The content specified by <i>Boxes</i> , a list of box terms, placed side by side, left to right, aligned vertically at their tops and separated by <i>Sep</i> , a single box.
<code>v(<i>Boxes</i>,<i>Sep</i>)</code>	The content specified by <i>Boxes</i> , a list of box terms, placed one above the other, top to bottom, aligned horizontally along their left sides and separated by <i>Sep</i> , a single box.
<code>fill(<i>Boxes</i>,<i>Sep</i>,<i>Width</i>)</code>	<i>Boxes</i> arranged in order in as many horizontal rows as necessary, with each row holding as many of <i>Boxes</i> as will fit, with <i>Sep</i> between boxes in the same horizontal row, but not between rows. The width of each line must not exceed the width of <i>Width</i> , which is a box whose content is ignored (only its width is considered). The overall width of the <code>fill</code> box is padded as necessary to reach the the same width as <i>Width</i> . This is the way words are arranged in a paragraph.
<code>horv(<i>Boxes</i>,<i>HSep</i>,<i>Indent</i>)</code>	The same as <code>h(<i>Boxes</i>,<i>Hsep</i>)</code> if that is not too wide to fit on the page. Otherwise it is a vertical box containing all the elements of <i>Boxes</i> , except that each element but the first is indented by the width of the box <i>Indent</i> relative to the first element of <i>Boxes</i> .
<code>width(<i>Box</i>)</code>	A box the same width as <i>Box</i> , but with zero height.
<code>height(<i>Box</i>)</code>	A box the same height as <i>Box</i> , but with zero width.

Table 1: Box specification terms

<i>Attribute</i>	<i>Possible Values</i>
<code>family</code>	<code>courier</code> (default), <code>helvetica</code> , or <code>times</code>
<code>weight</code>	<code>normal</code> (default) or <code>bold</code>
<code>slant</code>	<code>normal</code> (default) or <code>italic</code>

Table 2: Font attributes

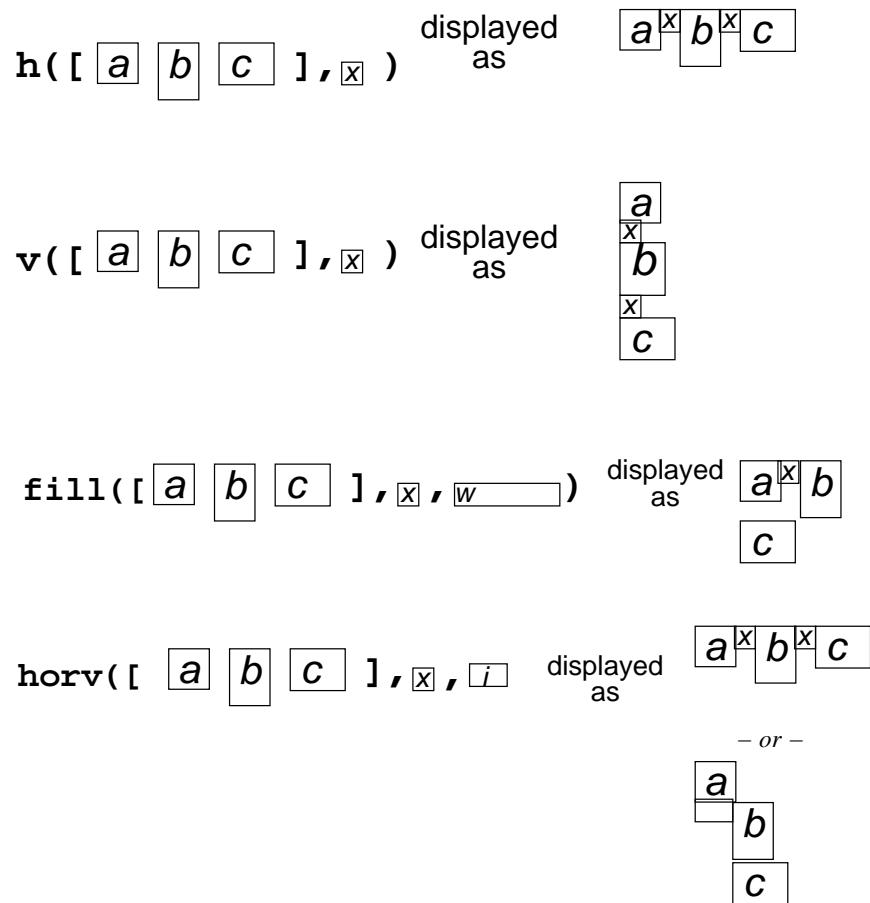


Figure 1: Box specification layouts

the code as possible between the two tasks. The specification has been designed to facilitate this sharing.

2. One way to achieve this is to divide the formatting task into two parts: (a) planning the output, and (b) producing the output. The planning part should be the same for both text and postscript output, the only difference being that the sizes of boxes are measured in characters for text, and points for postscript, and the widths of different characters may be different for postscript. Producing the output based on the result of the planning should be relatively straightforward, though completely different, for both both text and postscript.
3. A good representation for the result of the planning process would be a list of lines, where each line is a list of commands, where each command specifies either horizontal movement or text output. Alternatively, each command could specify text output and the horizontal position at which it should be placed.
4. The built-in predicate `atom_codes(Atom,Codes)` will convert between an atom and a list of ascii codes for the atom. You will need the ascii codes to find the width of the atom.
5. Save your print quota (and some trees) by using `ghostview` or another postscript previewer to check your generated postscript.

### Submission

You should write this program entirely in a single source file named `formatter.pl`. You should include your name, login, and student number in a header comment at the beginning of the file. You should also briefly document what this program does.

You should submit your `formatter.pl` file using

```
submit 255 proj1 formatter.pl
```

and verify it using

```
verify 255 proj1
```

The output from `verify` should contain the output from test runs of your program; if it does not, your program did not work. *It is your responsibility to verify your submission and check the output of `verify`.*

Late submission will be made using the command

```
submit 255 proj1.late formatter.pl
```

Late submissions will incur a penalty of 0.5% per hour late, including evening and weekend hours. This means that a perfect project that is much more than 4 days late will fail. If you have a medical or similar compelling reason for being late, you should contact Chris

Charnes (charnes@cs.mu.oz.au) as early as possible to ask for an extension (preferably before the due date).

To allow timely distribution of sample answers, **no extensions will be given beyond one week past the closing date.** Students with medical or similar compelling reasons for being more than one week late will be excused from submitting this project, and other parts of their assessment for the subject will count for more of their final mark.

Your program will be compiled by the Prolog command `[formatter]`. of the SWI Prolog system installed on the department's student sparc machines (*eg, cat, lister, holly, etc.*, but not *queeg*). It should compile with no warnings or error messages. Note that a program that compiles and runs correctly using some other Prolog system, or even the same system on a different architecture, may not compile cleanly, and may not execute correctly, when compiled and run as specified above. *It is your responsibility to test your program on a student sparc machine as specified above.*

## Assessment

Your program will be assessed partly on whether or not it correctly executes a series of test cases. Correctness will be assessed based on whether or not your output matches the expected output character for character. Therefore, you should follow the specification precisely.

Some of the test cases will shortly be made available in the student data area for the subject. Other test cases will not be made public.

Other aspects of assessment will include soundness of design, appropriate use of abstraction, clarity of documentation (documentation should briefly explain the purpose of the program and the approach taken to implementing it; it need not be voluminous), and clean, consistent coding style.

## Note Well:

**This project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties.**