Assignment 3

Pic2PS — Part 1

Code Due: 11:59 pm, Wednesday, September 28, 2005

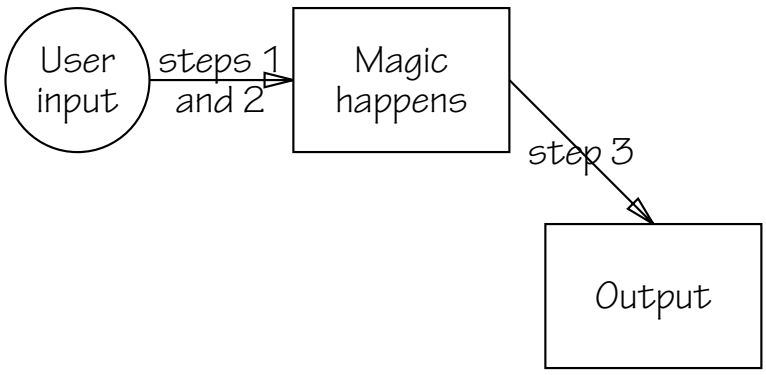

Figure 1: Example pic output.

In this assignment, you will implement a simple compiler/translator; specifically, from a subset of the pic language into the PostScript language.

Instructions

Before beginning this assignment, you should read the handout on "Little Languages" by Jon Bentley, which uses the pic language as an example. (You can learn more about the pic language by looking at the *Pic User Manual*, available from the assignments web page.) Copy all the files from `/cs/cs131/src/ass3` and modify them as specified below.

Because this assignment involves a number of files, instead of loading code with `use` you should again use the SML/NJ Compilation Manager and the `CM.make` function (i.e., type `CM.make "sources.cm";`). When testing, you may also wish to type `use "fullprint.sml";` to prevent values printed by SML/NJ from being abbreviated.

To submit code, run the command `cs131submitall` with no arguments. As before, to get credit your final submission may not have syntax or type errors. Functions must have the exact names and types given in the assignment (though you are encouraged to define other helper functions as needed), and should be reasonably well-commented in order to help the graders understand your code.

Your grade will depend not just on correctness, but on clarity (e.g., comments and how easy it is for us to understand how your code works) and elegance (e.g., factoring out common code rather than duplicating sections of code).

element :	*shape attributes* ;
	direction ;
		direction :	`right`
			`left`
			`up`
			`down`

shape :	`box`
	`circle`
	`move`
	`arrow`
	`line`
		attributes :	*string attributes*
			direction attributes
			ϵ

Figure 2: Syntax of pic inputs

Introduction

The pic language is a little language for drawing pictures. A pic program is a sequence of "elements". The syntax of the subset of pic you are to use is given in Figure 2. In this grammar, *string* represents a string constant contained in double quotes, and ϵ represents the absence of anything (in this particular grammar, a sequence of zero attributes). Thus one possible pic program would be

```
circle "user" "input"; arrow "steps 1" right "and 2";
box "Magic" "Happens"; arrow "step 3" right down; box "output";
```

(The result of this program is shown in Figure 1.)

At any given point in a pic program, there is a current position (on the page) and a current direction (left, right, up, or down). The commands `left`, `right`, up, or `down` by themselves simply change the current direction.

The `box` and `circle` commands draw a box or circle, respectively, adjacent to the current position; exactly how the shape is positioned depends on the current direction. If the current direction is "right" then the middle of the left side of the box is placed at the current position (so that the box is to the "right" of the current position); similarly, if the current direction is "down" then the box will be placed such that the middle of the top edge is at the current position. After the shape is drawn, the current position is changed to the opposite side of the shape.

The `line` and `arrow` commands also draw a line or arrow, respectively, respectively starting at the current position and extending in the current direction. The current position is then changed to the far end of the new line or arrow. The `move` command changes the current position just like `line`, though no line is actually drawn.

Finally, some commands can have a list of "attributes" that modify the command; in this limited language the only attributes are strings or directions. The string attributes are drawn as text, one line per string, centered at the geometric center of the shape being drawn. The `line`, `move`, and `arrow` commands additionally use their direction attributes to *override* the current direction, so the command `move right` is the same as the sequence `right; move`.

More interestingly, multiple direction attributes are additive: `line right down` draws a diagonal line right and down, while `line right right` draws a line twice

as far to the right as normal. (If there are any direction attributes, they *replace* the default direction; when computing the net result of the specified directions, do not add in the default direction.)

If a command has at least one direction attribute, the current direction afterwards will be the *last* of the direction attributes (down and right respectively in the previous two examples).

Any direction attributes for a shape (box or circle) should be completely ignored.

1 PostScript (40%)

The PostScript language is a special-purpose language for specifying printer output. The general idea is that a program to draw a box three inches square on a page—or to draw the text of an document—can be much smaller than a bitmap specifying the color of each pixel on the page; furthermore, these programs can work without change on different printers with different resolutions.

For the purposes of this assignment, you only need to know a little bit about PostScript:

- PostScript is a stack-based language, just like the RPN example in Assignment 1. So, you should not be surprised to find out that the code

  ```
  6 4 2 sub mul
  ```

  has the effect of pushing `12` on the stack: first the numbers `6`, `4`, and `2` are successively pushed onto the stack (the "push" is implicit), then the top two (`4` and `2`) are subtracted (obtaining `2`), and then `6` and `2` are multiplied.

  The stack can hold any PostScript values, including numbers and strings; strings in PostScript are delimited with parentheses instead of quotes, e.g., `(Hello!)`.

  All PostScript commands take their arguments, if any, on the stack. For example, the `moveto` command pops off two values and uses them as (x, y) coordinates.

- The default PostScript coordinate scheme puts $(0, 0)$ at the lower left corner of the page, so that increasing the x-coordinate corresponds to moving rightwards on the page, while increasing the y-coordinate corresponds to moving upwards. Although these coordinates are, by default, expected to be expressed in points (72^{nds} of an inch), the coordinate system can be changed. *In fact, the code you are given adjusts the coordinate system to express all page coordinates in inches.*

- The general scheme for drawing a shape is as follows

  1. Specify that you're starting a new shape with `newpath` (this is sometimes implied by what came before, but is always safe).

2. Execute a series of PostScript commands, including the drawing commands

`moveto`	Moves to the coordinates given by the top two stack values (e.g., `3 14 moveto` goes to position $(3, 14)$
`lineto`	Draw from the previous point to the coordinates given by the top two stack values
`rmoveto`	Do a relative move; adds the top two stack values to the current position.
`rlineto`	Like `rmoveto`, but adds a line along the way.
`closepath`	Takes no arguments; draws a line from the current position to the first point in the current shape. `closepath` is preferred over `lineto` when drawing the last edge of a closed figure
`show`	Draws the string at the current position
`arc`	Takes 5 numbers off the stack and draw an arc: the x and y coordinates of the center, the radius of the arc, and the starting and ending angles (e.g., 0 and 360 for a circle). If `arc` is not the first drawing command after `newpath`, it also draws a line from the current position to the start angle on the arc. (Be careful! The x-coordinate of the center is lowest on the stack, and the ending angle is taken from the top.)
`rotate`	Takes a number in degrees; "rotates" the whole coordinate system on the paper that many degrees. (To undo this transformation requires rotating back an equal amount.)
`scale`	Takes two numbers as a scaling factor: all coordinates are then multiplied by this factor. Repeated scalings, like repeated rotates, are cumulative.

   Note that `lineto`, `rmoveto`, `rlineto`, and `show` cannot be used until the first point of the path is set by a `moveto`.

3. Finally, say what you want to do with the path: either `stroke` (draw the path as with a pen) or `fill` (color in the interior of a closed path whose first and last points are the same). (The `lineto`, `rmoveto`, `show`, etc., commands all specify a path, but it doesn't actually appear until you do `stroke` or `fill`!)

   All these steps can be repeated; a PostScript file usually has many `newpath` ...`stroke` pairs.

- In PostScript, whitespace and line breaks are not significant (except in strings).

Thus, for example, the PostScript code

```
newpath
2 4 moveto
1 3 lineto
stroke
```

will draw a line from (2,4) to (1,3). The code

```
1 3 2 4 newpath moveto lineto stroke
```

would do the same thing, but is harder for humans to read (and also uses more stack space).

The `PSUtil` module defined in the file `psutil.sml` is supposed to contain helper code for generating PostScript code for most of the shapes needed. For example, the function call

```
PSUtil.lineCode { from = (1.0, 1.0),  to = (3.0,5.0) }
```

should return a string containing PostScript code that draws a line from $(1, 1)$ to $(3, 5)$. Note that the arguments to the `PSUtil` functions are *records*, which are described in Section 5.2 of the handout on SML.

For You to Do...

Most of the functions in `PSUtil` are unimplemented; complete the definitions in this file to behave as required by the signature `PSUTIL` (in the same file).

(For debugging purposes, it can be handy to have your code add comments to the generated PostScript output itself, but you are not required to do so.)

2 Translation (60%)

You have been given a number of other files, most of which are involved in translating pic input into SML data structures; see the file `absyn.sml` for how pic programs are being represented.

The main work of drawing pictures is done in the `Pic2ps` structure, found in `pic2ps.sml`.

For You to Do...

In your `pic2ps.sml` file, fix the definition of the function

```
translateElem : Absyn.element * (real * real) * direction ->
                    string * (real * real) * direction
```

such that given a pic element, the current page position, and the current direction, the function yields the PostScript translation of that command, the new page position, and the new current direction.

Do some planning before you start writing your code! At a minimum, you should decide how you'll break this problem down into helper functions (and possibly how those functions will be broken down in turn).

Also, be sure to comment your code well—you never know whether you might be asked to modify this translator in a later assignment, and, besides, it makes the graders happy.

Once this function is complete, you can use the predefined function `pic2ps` to create a PostScript output file from a pic input file. Running

```
Pic2ps.pic2ps "test1.pic" "out.ps";
```

in SML reads the input file `test1.pic`, runs it through the lexer, the parser, and your translator, and puts the PostScript output into the file `out.ps`. You can look at this output file with `gv` on turing, wilkes or knuth. On a Mac, you should run `pstopdf` on your PostScript file to get a PDF file, and then open the PDF file with Preview (you can, in fact, open the PostScript file directly with Preview but if you do that, you won't get any useful error messages if there is a bug in your PostScript code).

If you want to see the actual elements being translated, you can use the function `Parser.parse`, which takes a string holding the name of a pic file and which returns a list of element values.

Checking Your Work

Remember, if you want to see what a "real" pic implementation draws for a particular input, the lab handout contains instructions for using `gpic` to render a pic illustration.