Assignment 4

Pic2PS

All Parts Due: 11:59 PM, Monday, February 22, 2016

Ideally, get it done by Sunday night.

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

(a) Input description
(b) Output graphics

Figure 1: Example pic input & output.

The pic language is a domain-specific language for drawing pictures, as shown in Figure 1. 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 (but not before beginning the lab), 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 HomeworkAssignments page of the course Wiki site.)

To begin the assignment, copy the files in the usual way, as described on the Wiki.

To submit code, use the command `svn commit`. You can use `svn status` to see if there are any files in your directory that are not in your repository, and if necessary add them with `svn add`. 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).

Introduction to the pic Language

The pic language is a domain-specific language (also known as 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 for this assignment is given in Figure 1. In this grammar, `string` represents a
element: shape attributes ;  
direction ;  
shape: box  
circle  
move  
arrow  
line  
direction: right  
left  
up  
down  
attributes: string attributes  
direction attributes  
ε

Figure 2: Syntax of pic inputs (for our language subset)

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 the one shown in Figure 1(b).

THE CURRENT POSITION

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.

SHAPE COMMANDS

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.

ATTRIBUTES

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, thus, 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.

**Generating 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:

**STACK-BASED STRUCTURE**

PostScript is a stack-based language, just like the RPN example in Assignment 2 (such
languages are sometimes referred to by the slightly more general term *concatenative*
programming language). So, you should not be surprised to find out that the code

\[
6 \ 4 \ 2 \ \text{sub} \ \text{mul}
\]

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.

**COORDINATE SCHEME**

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 (\(72^{\text{nds}}\) 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.
PATHS AND DRAWING

To draw something in PostScript, you

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

- 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.)
  - `translate` Takes x and y coordinates; “translates” the whole coordinate system to move (0,0) to be at (x,y). (To undo this transformation requires translating back an equal and opposite amount.)
  - `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 and opposite amount.)
  - `scale` Takes two numbers as a scaling factor: all coordinates are then multiplied by this factor. Repeated scalings, like repeated rotates or translates, are cumulative.

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

- 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.
**Lexical Structure**

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).

**Outputting PostScript from Haskell**

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

```
lineCode :: XYPos -> XYPos -> PostScriptCode
```

is supposed to generate `PostScriptCode` for a line, such that a call to

```
PostScript.lineCode (1.0, 1.0) (3.0, 5.0)
```

should return `PostScriptCode` that draws a line from (1,1) to (3,5). (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.)

**TO DO:** Most of the functions in PostScript are unimplemented. You need to implement them. You are free to put the “smarts” in the PostScript module and output very direct/explicit PostScript, or alternatively, you can have fun writing helper functions in your PostScript header, thereby making the code for your PostScript module almost trivial (since it in that case, it merely needs to output simple calls your helper functions). Personally, I like the latter approach, but I enjoy writing helper functions in PostScript—maybe you don’t.

**Translation (60%)**

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

The main code for drawing pictures is in `pic2ps.hs`. 
TO DO: In pic2ps.hs, the core of translation to pic happens in the function

    doElem :: State -> Element -> (State, PostScriptCode)

which, 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. Unfortunately, this function is unfinished, and the code that is
written calls helper functions that aren’t implemented at all.

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 run in ghci to
create a PostScript output file from a pic input file. Running

    run "test1.pic"

in ghci reads the input file test1.pic, runs it through the lexer, the parser, and your
translator, and puts the PostScript output into the file test1.ps. Alternatively, you can
build the command-line tool using the instructions provided at the top of the source file.
You can look at your output file with gv on 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).

Checking Your Work

Remember, if you want to see what a “real” pic implementation draws for a particular
input, the Lab4WriteUp wiki page contains instructions for using groff to render a pic
illustration.