Assignment 3
Pic2ps, Part 1

Due: Wednesday, February 13

Goals for this Assignment

1. Defining abstract syntax
2. Writing a recursive-descent parser
3. Writing a simple compiler/translator, namely from a subset of the *pic* language into PostScript.

Office Hours

The purpose of office hours is to provide you with a chance to discuss class-related topics, to ask questions about material you find particularly interesting or particularly confusing, and to provide comments about the class.

You can drop by Olin 1253 any time to see if the professor is around; times specifically set aside are Tuesday, Thursday, and Friday from 2:30–4pm. Grutors are available in the Terminal Room on Tuesdays 7–9pm and Sundays 2–4pm. Questions can also be sent to `cs131help@cs.hmc.edu` .

Instructions

Begin with Bentley's chapter *Little Languages*, 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.) Then, copy all the files from `/cs/cs131/src/3` 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()` command.

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.

program :
    command *program*
    eof

command :
    primitive *attribute-list* ;
    direction ;

primitive : `box`
    `circle`
    `move`
    `arrow`
    `line`

direction : `right`
    `left`
    `up`
    `down`

attribute-list :
    string *attribute-list*
    direction *attribute-list*
    ϵ

Figure 1: Concrete Syntax for the Input

Introduction

The *pic* language is a little language for drawing pictures. A *pic* program is a sequence of drawing commands. The concrete syntax of a `pic` subset is given in Figure 1. In this grammar *eof* represents the end of the input file, *string* represents a string constant contained in double quotes, and ϵ represents the null string (no tokens). So, 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 2.)

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` or `circle` command draw a box or circle adjacent to the current position; exactly how the shape is positioned depends on the current direction. If the current direction is

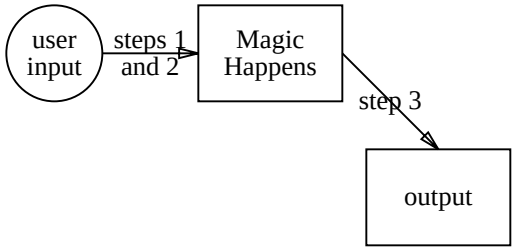

Figure 2: *pic* output

"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 so 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 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, in the sense that `line right down` draws a line diagonally right and down, while `line right right` draws a line twice as far to the right as normal. In such cases, the current direction is set to the *last* of the direction attributes (down and right respectively in the previous two examples).

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

1 Abstract Syntax (20%)

The tokens are defined in the structure `Tokens`, which contains the following definition:

```
datatype token = Left_tok | Right_tok | Up_tok | Down_tok
               | Box_tok | Circle_tok | Move_tok | Arrow_tok | Line_tok
               | String_tok of string
               | Semi_tok
               | EOF_tok
```

i.e., *pic* keywords, string constants, and the `Semi_tok` token, which represents semicolon.

The `pic2ps.sml` file already contains a function

```
tokenize : string -> Tokens.token list
```

which, given a file name, converts this file into a list of *pic* tokens.[1]

[1] Of course, this function is defined inside a structure `Pic2ps`, so if you want to run it from the SML/NJ prompt to see what the output looks like you need to refer to it as `Pic2ps.tokenize` .

The list of tokens returned by the lexer will have a single `EOF_tok` as the last token, representing the end-of-file.

TO DO: In your copy of `pic2ps.sml`, define a type `command` suitable for representing the abstract syntax of a command in a *pic* program. You are likely to want to do this in terms of other type definitions or datatype definitions. (You may want to think about how the rest of the translator will be implemented as you decide on a representation.) You have been given a couple helper definitions to start out with.

2 Parsing (40%)

TO DO: In the file `pic2ps.sml`, construct a recursive-descent parser for parsing the grammar shown in Figure 1. You are given a "main" function

```
parseProgram : token list -> command list
```

which does the full parse. (Since a program is not supposed to have any leftover tokens, none are returned.) You will want to have other helper functions corresponding to some or all of the nonterminals in the grammar, particularly

```
parseCommand : token list -> command * token list
```

which constructs an element out of a prefix of the token list, and which returns this element and any leftover tokens. You are *not* required to have the structure of your parser exactly mirror the concrete syntax (e.g., `parseCommand` could handle directions itself rather than calling a `parseDirection` function) but should be closely guided by the concrete syntax.

For debugging purposes you can always call `Pic2ps.tokenize "myfile"` to get a list of tokens from a file, and then manually pass this to `parseCommand`, `parseProgram`, etc.

3 PostScript (10%)

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.

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 (72nds of an inch), all of this can be changed. *In fact, the code you are given modifies this system so that all page coordinates are to be expressed 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 safe)
  2. Execute a series of drawing commands, including

`moveto`	Moves to the coordinates given by the top two stack values
`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.
`closepath`	Draw a line from the current position to the first point in the current shape; preferred over `lineto` when drawing the last edge of a figure
`show`	Draws the string at the current position

     Note that `lineto`, `rmoveto`, and `show` cannot be used until the first point of the path is set by a `moveto`. These commands automatically update the current position as one would expect.
  3. 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).

- In PostScript, whitespace and line breaks are not significant.

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

```
PSUtil.lineCode {xstart = 1.0, ystart=1.0, xend = 3.0, yend = 5.0}
```

returns a string containing PostScript code which draws a line from $(1, 1)$ to $(3, 5)$, i.e., a string containing the code

```
newpath
1.0 1.0 moveto
3.0 5.0 lineto
stroke
```

TO DO: Complete `psutil.sml` by fixing the definition of the function

```
boxCode : {xcenter:real, ycenter:real, width:real, height:real}
                                                        -> string
```

so that it generates PostScript to draw a box of given dimensions centered at a given point.

4 Translation (30%)

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

```
translateCmd : command * (real * real) * direction ->
                        string * (real * real) * direction
```

so that given a *pic* command, 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 this code! At a minimum you should decide how you'll break this problem down into helper functions (and possibly how these will be broken down in turn). Also, be sure to comment your code well...you never know if 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`. Running

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

in SML reads the input file `test1.pic`, runs it through the lexer, your 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.

If you want to see what the "real" pic program draws for a particular input, you can:

1. edit the file `gpicinput` to put your pic program between the `.PS` and the `.PE` lines.
2. run the following two commands on turing

```
gpic -t gpicinput > gpic.tex
tex gpic.tex
```

(The commands are `gpic` and `tex`, not `pic` and `latex`.)

The picture should show up in `gpic.dvi`, which you can look at using `xdvi`.