Assignment 5

Pic2PS — Part 2

Code Due: 11:59 pm, Wednesday, October 26, 2005

This assignment gives you more experience in designing abstract syntax, using lexer and parser generators, and introduces you to some of the issues involved in interpreting an imperative language.

Instructions

This assignment requires extending the pic translator from the Assignment 3. The only new files in `/cs/cs131/src/ass5` is the file `picenv.sml`. The other files in that directory are from the sample solution to the assignment. You may use either your own code, or the sample solution as your starting point.

This assignment involves implementing a larger subset of pic. This subset is described below, but you may also want to refer back to the handout on "Little Languages" by Jon Bentley and to the *Pic User Manual*, which is available on the assignments web page.

As usual, your grade will depend not just on correctness, but on clarity (e.g., comments and the ability of the graders to easily understand how your code works) and elegance (e.g., factoring out common code rather than duplicating sections of code).

1 Abstract Syntax (10%)

The grammar in Figure 1 is an extended version of the pic grammar from Assignment 3. It includes (real) numbers, expressions, assignment to variables, `for` loops, curly-brace–delimited blocks (`{...}`), and optional lengths for direction specifications. (The grammar for expressions is ambiguous but standard rules apply: unary negation has the highest precedence; `*` and `/` have equal but lower precedence; and `+` and `-` have equal but lowest precedence, and all four binary operators are left associative.)

In your copy of `absyn.sml`, update the definition of the type `element` to include abstract syntax for the enlarged set of possible `pic` programs. As part of this you should define a type named `expr` that can be used to represent `pic` arithmetic expressions.

Strive for good representations: drop all unnecessary details about the concrete syntax, and enforce necessary restrictions where possible (e.g., if something is *required* to be a numeric constant, represent it with an SML integer rather than as a value of type `expr` that you hope will be representing an integer).

elements :	*element elements*
	ϵ
element :	*shape attributes* ;
	direction ;
	var = *exp* ;
	{ *elements* }
	`for` *var* = *exp* `to` *exp* `by` *exp* `do` { *elements* }
shape :	`box`
	`circle`
	`move`
	`arrow`
	`line`
attributes :	*string attributes*
	direction optexp attributes
	ϵ

direction :	`right`
	`left`
	`up`
	`down`
optexp :	*exp*
	ϵ
exp :	*number*
	var
	– *exp*
	(*exp*)
	exp + *exp*
	exp – *exp*
	exp * *exp*
	exp / *exp*

Figure 1: Extended pic syntax

2 Lexing and Parsing (40%)

Extend the `pic.grm` and `pic.lex` files to correctly handle the extended concrete syntax (shown in Figure 1) and to generate the appropriate abstract syntax. When you have completed your changes, the function `Toplevel.parse` will return values of your new abstract syntax.

3 Adding Variables (20%)

The file `picenv.sml` contains a module `PicEnv` that defines a type `PicEnv.env` of environments (functional lookup tables mapping strings to real numbers), along with functions such as `PicEnv.lookup` and `PicEnv.extend`. *Edit `sources.cm` to add this file.*

Next, we want the translator to follow the official implementation by using the pic variables `boxwid` and `boxht` when deciding what size box to draw, instead of relying on the (constant) SML variables of the same name. To do so, the translation functions will need an environment as an extra parameter. (All the relevant variable names are given in the pic documentation, and also listed near the end of `picenv.sml`). You should then delete the SML variables `boxwid`, `arrowht`, etc. from your `pic2ps.sml` file to make sure that none are still being referenced accidentally.

To this end, the main `translateElem` should be modified so that it takes an environment as an extra argument, and returns an environment as an extra result:

```
Absyn.element * (real * real) * Absyn.direction * PicEnv.env ->
   string * (real * real) * Absyn.direction * PicEnv.env
```

The argument environment gives the values of the variables (`boxwid`, `circlerad`, etc.) that provide sizes during the translation. The resulting argument is a modified copy of the given environment updated with any assignments that occurred when interpreting the given command. None of the actual drawing elements inherited from Assignment 3 change the values of variables, so these cases all just return the given environment.

Finally, you will need to modify the `loop` function inside the `pic2ps` function so that it takes an environment argument, passes it to `translateElem` to get an updated environment, and then passes this to the recursive call.

4 Adding Expressions (10%)

Define a function,

```
evalExp : PicEnv.env * Absyn.expr -> real
```

to your `pic2ps.sml` file (before the translation functions). This function should evaluate the given expression, using the given environment to find the values of variables.

Extend the interpreter to handle direction attributes that specify lengths. (Recall that if we say `arrow right (x+1)`, we expect a right-pointing arrow that is `x+1` inches long instead of the default `linewid` inches long.)

5 Adding Assignment, Blocks, and `for` Loops (20%)

Finally, change `translateElem` so that it handles assignments, curly-brace–delimited sequences, and `for` loops as described in the pic documentation. A few hints:

- An assignment element should not cause `translateElem` to change the position or direction, nor should it return new PostScript code. It should cause `translateElem` to return an updated environment in which the specified variable has a new value. (Conversely, the other elements you've implemented previously return the environment they were given.) Verify that, for example, performing assignment on the variable `circlerad` in a pic test file causes the sizes of your circles to change.

- In pic an element that is just a sequence of elements surrounded by curly braces generates PostScript code as usual, but after the block is finished the current position and direction are restored to the values they had when the sequence was started. This behavior can be implemented easily by having `translateElem` return the position and direction given rather than any values computed during the translation of this block. (The environment is not restored afterwards, however. You may be interested to know that pic actually does undo all assignments when leaving a block delimited by `[...]`, but you are not being asked to implement that part of the language.)

- `for` *var* `=` exp_1 `to` exp_2 `by` exp_3 `do {` *elements* `}` first sets *var* to exp_1, and then, while the value of *var* is less than or equal to that of exp_2, we repeatedly draw *elements* and increment *var* by the value of exp_3.

  The position, direction, and environment computed at the end of each iteration is used for the beginning of the next iteration. The PostScript returned will be the concatenation of the PostScript code created by each iteration of the loop.

  Warning: although the body of a `for` loop is syntactically required to be bounded by curly braces, this concrete syntax is misleading: the position and direction are not automatically reset once the loop body ends. If you really do want the position and direction to be reset after every iteration of the loop, the loop body must be inside yet another pair of curly braces.