1. (Pretty-printer library) You will implement a “pretty printer”, which takes semistructured data and
converts it to a string with good indentation and line breaks (for a given text width). A detailed
explanation of the algorithm is below. (It’s not something you would come up with on your own!) In
terms of the provided code:

- The type doc describes the input. (The next problem creates such input from two different
  languages.) The type sdoc is a lower-level form that is easy to convert to a string.
- The difficult part is converting a doc to an sdoc (done by format which uses fits as a helper
  function). See below.
- The easy part is converting an sdoc to a string (done by sdocToString) and putting the pieces
together (done by pretty).

**Background:** We now describe the types and algorithm: An input document (“doc”) consists of
strings (e.g., “hello”), which are separate by optional line breaks (\_) and glued together with the cons
(\:) operator (e.g., “hello” : \_) : “world”). The \_ will be printed as either a line break or a space
(if no line break is needed). We can group elements together (e.g., [doc1 : doc2 : doc3]) and use
“nesting” to increase indentation.

Line break behavior is modified by groups: either all \_ in a group are converted to spaces, or they
are all converted to newlines. Subgroups within a group are a “new scope”, and may choose between
spaces and line breaks independently of their parents, siblings, or children. However, if a group is
printed “flat” (i.e. with breaks treated as spaces), then all its subgroups are printed flat as well. Here
is the algorithm to decide about line breaks:

(a) Print every optional line break of the current group and all its subgroups as spaces. If the current
group then fits completely into the remaining space of the current line, this is the layout of the
group (and its subgroups).

(b) If the former fails, every optional line break of the current group is printed as a newline. Subgroups
and their line breaks, however, are considered individually as they are reached by the pretty
printing process.

A nesting indicates a number of spaces to be added after any line-break-turned-newline inside it (i.e.,
no spaces are added if a line-break becomes a space). Nested nestings are cumulative (i.e., the number
of spaces is the sum of all nestings in which the \_ is contained).

Examples will be posted to the class email list.

Note: This approach is due to Philip Wadler. Solutions are on the Web; do not consult them.

**Implementation:** The six constructors for doc correspond to “do nothing”, strings, \_, :, groups, and
nestings. The \_ has the flexibility of a user-defined string (so \" \" is the right choice for a single space).
Several helper functions make creating values of type doc easier.

A doc is transformed into a simpler document of type sdoc, which is then converted to a string. The
first transformation (in function format) actually lays out each group of the document, since sdoc has
no grouping mechanism. An sdoc is either empty, a string followed by another sdoc, or is a newline
followed by a number of spaces and then another sdoc.
format takes a line-width \( w \), a space-consumed-on-current-line \( k \), and a stack (represented with a list) of work to do. Stack elements are triples, including an indentation level \( i \), a mode \( m \) (either Flat or Break), and a doc \( d \). The initial stack has one element (the whole document), an indentation of 0 (we are not nested inside anything), and a mode of Flat (we are going to try to avoid line-breaks).

format has several cases. If the stack is empty, the empty sdoc suffices, else we pop the top element of the stack and examine its structure (use fancy pattern-matching). For the “do nothing” document, we just recur with a smaller stack. For a “;” document, we recur with a bigger stack (do the first contained doc first). For a “nest”, we recur with a same-size but different stack (the new top element has a larger indentation and a smaller doc). For a string, we create a string sdoc where the contained sdoc is computed by calling format recursively with a smaller stack and bigger \( k \). For a \_;\_\_;\_\_, we examine the mode, creating a string sdoc (as in the string case) or a line sdoc as appropriate (using the indentation in the latter case).

Finally, for a group, we use the helper function fits to see if the contained document can be done without line breaks. fits takes the “space remaining” on the current line and the same stack that format uses (changed here to start with the contained document and mode Flat). format recurs with the group’s contained document, the same indentation level, and the mode required by fits (i.e., Flat if an only if it fits).

All that remains is implementing fits. It returns false if \( w \) is less than 0, else true if the stack becomes empty or the first element is a \_\_\_\_\_\_\_\_\_ with mode Break (the latter meaning we have hit a line break, so what preceded it does fit on a line). The remaining cases require recursion, adjusting either \( w \) (if the next stack element “takes up space”) or the stack (if the next element is a “compound document”). (We are being less specific than with format; the ideas are similar.) For the “group” case, the mode for the new top-element should be Flat because groups inside flat groups must be flat.

2. (Pretty-printer clients)

(a) Write stmtToDoc to complete the pretty-printer for IMP programs we have started for you. (We have provided the definition of IMP syntax in ast.ml, an IMP parser in lex.ml and parse.mly, and an example program in facten.imp). The syntax rules for IMP statements are:

- A skip statement is the keyword skip.
- An assignment statement is the variable name then \:= then the expression.
- A sequence statement is the first statement then \; then the second statement.
- An if statement is the keyword if then the expression then the first statement in parentheses then the second statement in parentheses.
- A loop statement is the keyword while then the expression then the statement in parentheses.

To be careful, put sequences, ifs, and loops in parentheses. Make sensible indentation and line-break choices.

(b) Implement xmlToDoc : xmlAst \rightarrow doc for pretty-printing an XML document, where xmlAst is defined for you. (Let the course staff know if you need an explanation of the concrete syntax of XML.) Make sensible indentation and line-break choices.

3. (Pretty-printer test) Implement test_prog : string\rightarrow\text{bool} to test whether your IMP pretty-printer is correct for the filename passed as an argument. A pretty-printer is correct if “parse-from-file then print-to-string” produces the same string as “parse-from-file then print-to-string then parse-from-string then print-to-string”. (Hint: Use get_prog_f and get_prog_s. This is not a difficult problem.)

4. (String functions) The file stringfuns.ml has functions you need to implement as described below. You should not use String library functions except String.length and String.get (and for the latter you can write \( s.[n] \) to get character \( n \) from string \( s \)). You may use Char library functions. Tail-recursion is fine but not required.
(a) **reverse**: `string -> string` returns a string where the characters are in the reverse order from the input. (Hint: just concatenate one more character on at a time even though this is inefficient.)

(b) **map**: `(char -> 'a) -> string -> 'a list` creates a list where the $i^{th}$ element is `map`'s first argument applied to the $i^{th}$ character in `map`'s second argument.

(c) **fold_left**: `('a -> char -> 'a) -> 'a -> string -> 'a` is a fold over strings that calls its first argument on the characters in left-to-right (beginning of string to end of string) order.

(d) **fold_right**: `(char -> 'a -> 'a) -> string -> 'a -> 'a` is a fold over strings that calls its first argument on the characters in right-to-left (end of string to beginning of string) order.

(e) **uppercase**: `string -> string` returns a string like its input except all English characters are uppercase. Use `map`.

(f) **lowercase**: `string -> string` returns a string like its input except all English characters are lowercase. Use `map`.

(g) **titlecase**: `string -> string` returns a string like its input except all English characters after space characters (and the first character) are uppercase and all other English characters are lowercase. Use `fold_left`. (Hint: Pass the accumulated answer and whether the previous character was a space. “When done” return the first component of this pair.)

(h) **histogram**: `string -> int list` returns a list in which the $i^{th}$ element is how many times the $i^{th}$ letter of the alphabet appears in the string. (Hints: Use `uppercase` (or `lowercase`). Also use a helper function 26 times, to count how many times a particular character appears. This helper function should use `fold_left` or `fold_right`. Use this helper function with `map` and a string containing the English alphabet. Sample solution is about 7 lines.)

5. **(Extra Credit)** Change `expToDoc` so it does not print unnecessary parentheses (but does print necessary parentheses):

- Addition and multiplication are associative, so $(x + 2) + (y + z)$ can be $x + 2 + y + z$.
- Multiplication has tighter precedence than addition, so $(x * y) + z$ can be $x * y + z$.

**Turn in:**

- Email your source code to Ben as attachments.

- **If you are using Seminal, please include your backup files.**

- Put your code in files called `pprint.ml` and `stringfuns.ml`.

- Do not modify other files.