The BNF definition of Logo syntax is:

\[
  e ::= \text{home} \mid \text{forward } f \mid \text{turn } f \mid \text{for } i \text{ lst} \\
  lst ::= [] \mid e :: lst
\]

where \( f \) is a floating-point number and \( i \) is an integer. Note a for-loop executes a fixed number of times and its body is a list of “moves”. A Logo program is a \( lst \), i.e., a list of moves.

Informally, the semantics of a move list is:

- A program state includes a “current x-coordinate” (call it \( x \)) a “current y-coordinate” (call it \( y \)) and a “current direction” (call it \( d \)). All are floating-point numbers. \( d \) is in radians. So direction 0.0 is “facing East” and \( \pi/2 \) is “facing North”.
- The initial program state is 0.0 for each of \( x \), \( y \), and \( d \).
- A move \( e \) takes a state and a list of “places visited so far” and produces a state and a list of “places visited so far”. A place is an \( x \) and a \( y \) (no direction).
  - \text{home} changes the state back to the initial state.
  - \text{forward } r changes the state by “moving in the current direction the distance \( r \)” . So \( x \) and \( y \) may change, but \( d \) will not.
  - \text{turn } r changes the state by “adding \( r \) radians to the current direction”. (So \( x \) and \( y \) will not change and we do not “visit a new place”.)
  - \text{for } i \text{ lst} executes its move-list \( i \) times.
- A move-list executes each move in order. (The empty list does nothing.)

Notes:

- The trace of places visited could have repeats.
- It is best (but not strictly necessary) to “normalize” the current direction to always be between 0 and \( 2\pi \).
- You will notice floating-point rounding errors. Do not worry about them.
- Relevant high-school geometry:
  - A regular polygon with \( n \) sides has angles of \( 2\pi/n \) radians.
  - Starting from \( (x, y) \), the point distance \( r \) away in direction \( d \) is \( (x + r \cos d, y + r \sin d) \).
  - After turning \( d_1 \) radians from direction \( d_2 \), the new direction is \( d_1 + d_2 \).
- Relevant OCaml / F# differences:
  * OCaml does not have operator overloading. For example, while \( + \) works on ints, \( .+ \) works on floats. F# does have operator overloading, so you just use \( + \).
  * In OCaml, \texttt{mod_float} is useful for normalizing directions, but in F#, \( \% \) works just fine and therefore, understandably, there is no \texttt{mod_float} in the standard library.