The goal of this assignment is to implement the syntax analysis of a small Logo language (graphical turtle) whose interpreter is provided. No prior knowledge of Logo is required.

The tool `menhir` and the `graphics` library are requirements for this assignment. If you haven't already, install them with the command `opam install menhir graphics`.

The basic structure is provided (a tarball with OCaml files and a Makefile) : `mini-turtle.tar.gz`. Once uncompressed (for instance with `tar zxvf mini-turtle.tar.gz`, you get a directory `mini-turtle/` with the following files:

`turtle.ml(i)`	the graphical turtle (complete)
`ast.ml`	the abstract syntax mini-Turtle (complete)
`lexer.mll`	lexical analyzer (to be completed)
`parser.mly`	syntactic analyzer (to be completed)
`interp.ml`	the interpreter (complete)
`miniturtle.ml`	main file (complete)
`Makefile/dune`	to automate the build (complete)

The code compiles (run `make`, which runs `dune build`) but is incomplete. Places to be filled are marked `* to be completed */`. The program takes a file to be interpreted on the command line, with suffix `.logo`. When running `make`, the program is run on file `test.logo`.

Syntax of Mini-Turtle

Lexical Conventions

Spaces, tabs, and newlines are blanks. There are two kinds of comments: from `//` to the end of the line, or enclosed by `(*` and `*)` (and not nested). The following identifiers are keywords:

```
if else def repeat penup pendown forward turnleft
turnright color black white red green blue
```

An identifier `ident` contains letters, digits, and underscores and starts with a letter. An integer literal `integer` is a sequence of digits.

Syntax

Names in italics, such as *expr*, are nonterminals. Notation *stmt** means a repetition zero, one, or several times of nonterminal *stmt*. Notation *expr**, means a repetition of nonterminal *expr* where occurrences are separated with the terminal `,` (a comma).

$$
\begin{array}{lcl}
\textit{file} & ::= & \textit{def} * \ \textit{stmt}* \\
\textit{def} & ::= & \texttt{def ident (}\ \textit{ident}*, \texttt{)}\ \textit{stmt} \\
\textit{stmt} & ::= & \texttt{penup} \\
 & | & \texttt{pendown} \\
 & | & \texttt{forward}\ \textit{expr} \\
 & | & \texttt{turnleft}\ \textit{expr} \\
 & | & \texttt{turnright}\ \textit{expr} \\
 & | & \texttt{color}\ \textit{color} \\
 & | & \texttt{ident (}\ \textit{expr}*, \texttt{)} \\
 & | & \texttt{if}\ \textit{expr}\ \textit{stmt} \\
 & | & \texttt{if}\ \textit{expr}\ \textit{stmt}\ \texttt{else}\ \textit{stmt} \\
 & | & \texttt{repeat}\ \textit{expr}\ \textit{stmt} \\
 & | & \{\ \textit{stmt}*\ \} \\
\textit{expr} & ::= & \texttt{integer} \\
 & | & \texttt{ident} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textit{expr} * \textit{expr} \\
 & | & \textit{expr} \ / \ \textit{expr} \\
 & | & - \ \textit{expr} \\
 & | & (\ \textit{expr}\ ) \\
\textit{color} & ::= & \texttt{black} \mid \texttt{white} \mid \texttt{red} \mid \texttt{green} \mid \texttt{blue}
\end{array}
$$

Priorities of arithmetic operations are usual, and unary negation has the strongest priority.

Assignment

You have to fill files `lexer.mll` (`ocamllex`) and `parser.mly` (`Menhir`). The following questions suggest an incremental way of doing this. At each step, you can test by modifying file `test.logo`. The command `make` (at the root of the directory) runs tools `ocamllex` and `menhir` (to build/update the OCaml files `lexer.ml`, `parser.mli` and `parser.ml`), then compile the OCaml code, and finally run the program on file `test.logo`. If the parsing is successful, a graphical windows opens and displays the interpretation of the program. Pressing any key closes the window.

If needed, do `make explain` to display the conflicts detected by `menhir`.

1 Comments

Problem 1 Complete the file `lexer.mll` to ignore blanks and comments and to return the token `EOF` when the end of input is reached. The command `make` should be opening an empty window, since file `test.logo` only contains comments at this point.

2 Arithmetic expressions

Problem 2 Update the parser the accept arithmetic expressions and the `forward` statement. The file `test.logo` containing

```
    forward 100
```

should be accepted and a window should open with an horizontal line (100 pixels long). Check the priorities of arithmetic operations, for instance with

```
    forward 100 + 1 * 0
```

If the priorities are wrong, you will get a point instead of a line.

3 Other atomic statements

Problem 3 Add syntax for the other atomic statements, namely `penup`, `pendown`, `turnleft`, `turnright`, and `color`.

Test with programs such as

```
    forward 100
    turnleft 90
    color red
    forward 100
```

4 Blocks and control structures

Problem 4 Add syntax for blocks and control structures `if` and `repeat`. The two grammar rules for `if` should trigger a shift/reduce conflict. Identify it, understand it, and solve it in the way that is most appropriate.

Test with programs such as

```
repeat 4 {
    forward 100
    turnleft 90
}
```

5 Functions

Problem 5 Finally, add syntax for function declarations and function calls.

You can test using the files provided in subdirectory `tests`.

The command `make tests` runs the program on each of these files. You should get the following images (pressing a key in between):