EC3204: Programming Languages and Compilers (Fall 2023)
Homework 3: Implementing a Lexer and a Parser

100 points in total, 5% of the total score
Due: 11/20, 23:59 (submit via GIST LMS)

Instructor: Sunbeom So

Important Notes

- **Evaluation criteria**
  For each problem, the correctness of your code will be evaluated using testcases:

  $$\frac{\#\text{Passed}}{\#\text{Total}} \times \text{point per problem}$$

  "Total" indicates a set of testcases prepared by the instructor (undisclosed before the evaluation), and "Passed" indicates testcases whose expected outputs match with the outputs produced by your implementations.

- **No Plagiarism**
  Cheating (i.e., copying assignments by any means) will get you 0 points for the entire HWs. See the slides for Lecture 0. Code-clone checking will be conducted irregularly.

- **Executable**
  Before you submit your code, please make sure that your code can be successfully compiled by the OCaml compiler. That is, the command `./build` should not report any errors. Otherwise, you will get 0 points for that HW.

- **No Template Changes**
  Your job is to complete `lexer.mll` and `parser.mly`. However, you should not modify the other existing code templates.

- **No Printing Functions**
  The parts implemented by you should not contain printing functions such as `print_int`.

- **File Extension**
  The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`).

- **File Naming Rule**
  Submit `lexer.mll` and `parser.mly` only. Do not change the file names. Do not zip the files.

1 Assignment Summary

Your goal in this assignment is to implement a lexer and a parser for a toy imperative language, using a lexer generator (`ocamllex`) and a parser generator (`ocamlyacc`).

2 Structure of the Project

You can find the following files in the `hw3` directory.

- `main.ml`: contains the driver code.
- `s.ml`: contains the definition of the abstract syntax and the interpreter for our source language ("S"). The concrete syntax of S language is the following.

$$
\begin{array}{rcll}
\textit{program} & \rightarrow & \textit{block} & \\
\textit{block} & \rightarrow & \texttt{\{}\textit{decls stmts}\texttt{\}} & \\
\textit{decls} & \rightarrow & \textit{decls decl} \mid \epsilon & \\
\textit{decl} & \rightarrow & \textit{type } x\texttt{;} & \\
\textit{type} & \rightarrow & \texttt{int} \mid \texttt{int[}n\texttt{]} & \\
\textit{stmts} & \rightarrow & \textit{stmts stmt} \mid \epsilon & \\
\textit{stmt} & \rightarrow & \textit{lv}\ \texttt{=}\ e\texttt{;} & \\
 & \mid & \textit{lv}\texttt{++;} & \\
 & \mid & \texttt{if(}e\texttt{)}\ \textit{stmt}\ \texttt{else}\ \textit{stmt} & \\
 & \mid & \texttt{if(}e\texttt{)}\ \textit{stmt} & \\
 & \mid & \texttt{while(}e\texttt{)}\ \textit{stmt} & \\
 & \mid & \texttt{do}\ \textit{stmt}\ \texttt{while(}e\texttt{);} & \\
 & \mid & \texttt{read(}x\texttt{);} & \\
 & \mid & \texttt{print(}e\texttt{);} & \\
 & \mid & \textit{block} & \\
\textit{lv} & \rightarrow & x \mid x\texttt{[}e\texttt{]} & \\
e & \rightarrow & n & \text{integer} \\
 & \mid & \textit{lv} & \text{l-value} \\
 & \mid & e\texttt{+}e \mid e\texttt{-}e \mid e\texttt{*}e \mid e\texttt{/}e \mid \texttt{-}e & \text{airthmetic operation} \\
 & \mid & e\texttt{==}e \mid e\texttt{<}e \mid e\texttt{<=}e \mid e\texttt{>}e \mid e\texttt{>=}e & \text{conditional operation} \\
 & \mid & \texttt{!}e \mid e\texttt{||}e \mid e\texttt{\&\&}e & \text{boolean operation} \\
 & \mid & \texttt{(}e\texttt{)} &
\end{array}
$$

- `lexer.mll`: contains the lexer specification in `ocamllex`.
- `parser.mly`: contains the parser specification in `ocamlyacc`. This file should contain the definition of the above concrete syntax.

Your job is to complete `lexer.mll` and `parser.mly`, and submit these two files.

3 How to Run

Once you completed `lexer.mll` and `parser.mly`, you can build the project as follows.

```
$ make
```

Then, the executable `run` will be generated. To remove some dummy files (not mandatory), run the following command:

```
$ make clean
```

Your executable `run` should output a correct value for a given source program. For example, if you run the command

```
$ ./run test/t0.s
```

the executable `run` should output the value 1 as follows.

```
== source program ==
{
 int x;
 x = 0;
 print x+1;
}
== execution result ==
1
```