EC4219: Software Engineering (Spring 2024)
Homework 1: Program Verifier

100 points
Due: 5/22, 23:59 (submit via GIST LMS)

Instructor: Sunbeom So

Important Notes

- **Evaluation criteria**
  The correctness of your implementation 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).
  - "Passed" indicates testcases whose expected outputs match with the outputs produced by your implementations.

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

- **No Plagiarism and No Discussion**
  Cheating (i.e., copying assignments by any means) will get you an F. See the slides for Lecture 0. Code-clone checking will be conducted irregularly. Furthermore, discussions at all levels are strictly disallowed.

- **No Changes on Template/File Name/File Extension Changes**
  Your job is to complete `(* TODO *)` parts in provided templates; you should not modify the other existing code templates. Do not change the file names. The submitted files should have `.ml` extensions, not the others (e.g., `.pdf`, `.zip`, `.tar`).

- **No Posting on the Web**
  You should not post your implementations on public websites (e.g., public GitHub repositories). Violating this rule gets you an F, even after the end of the semester.

1 Goal

Your goal is to implement a verifier for simple imperative programs. Specifically, your verifier should be able to correctly verify whether a given program satisfies specifications (pre- and postconditions, loop invariants, and assertions) or not.

Before starting the homework, install `Batteries` library:

```
$ opam install -y batteries
```

2 Structure of the Project

You can find the following files in the `hw1` directory.

- `verifier.ml`: **Your job is to complete and submit this file only.**
- `main.ml`: contains the driver code.
- `lang.ml`: contains the definition of our target language. A program consists of a 6-tuple:

  $$(\mathit{pre}, \mathit{post}, \mathit{fid}, \vec{i}, o, c)$$

  where *pre* and *post* are FOL formulas, *fid* is a function identifier (name), $\vec{i}$ is a sequence of input parameters with type declarations, o is an output parameter, and c is a command. The command c is defined with l-values (lv) and expressons (e).

  $$\begin{array}{rcl} lv & \rightarrow & x \mid x[e] \\ e & \rightarrow & n \mid \texttt{len}(x) \mid lv \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \\ & & \mid \texttt{true} \mid \texttt{false} \mid e_1 \prec e_2 \mid \neg e \mid e_1 \;\&\&\; e_2 \mid e_1 \mid\mid e_2 \\ c & \rightarrow & \mathit{typ}\ x \mid lv = e \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ e\ c_1\ c_2 \mid \texttt{while}\ [f]\ e\ c \mid \texttt{assert}(e) \mid \texttt{return}(e) \end{array}$$

  where $\prec$ denotes the standard binary comparison operators ($==, !=, <, \leq, >, \geq$). The intended semantics should be clear throughout the following explanations.

  – *typ* x: Each variable or element gets assigned default values. The default values for `int` and `bool` are 0 and *false*, respectively. For example, given *int* x, 0 should be assigned to x. As another example, given *int*[*10*] x, 0 should be assigned to $x[0], \cdots, x[9]$.

  – $x = e$: given the assignment $x = y$ where both x and y are array-typed variables, we assume that all the elements in y are copied into x. That is, the array-typed variables are not aliased in our language. The other cases are standard.

  – `while` $[f]$ e c: f represents an annotated loop invariant and it does not have effects on the program semantics; it just helps to perform verification more precisely.

  – `assert`(e): `assert` does not affect the program semantics; `assert` just expresses properties expected to hold at certain locations, and nothing happens at runtime even if the conditions in `assert` do not hold.

  We will assume that programs to verify are well-typed. For example, the command like `len`(x), where x is an integer-typed variable, will not appear.

- `formula.ml`: contains the definition of FOL formulas.
- `solver.ml`: implements the interface to invoke Z3 SMT solver.
- `lexer.mll` and `parser.mly`: contain the lexer and parser specifications in `ocamllex` and `ocamlyacc`, respectively. You do not need to understand these details for this assignment.

3 How to Build

Once you complete `(* TODO *)` parts in `verifier.ml`, you can build the project as follows.

```
$ ./build
```

Then, the executable `./main.native` will be generated. You can run it as follows.

```
$ ./main.native -input TESTCODE
```

4 Running Example

If you run the command

```
$ ./main.native -input test/simple_array
```

you should obtain the following result:

```
...
=== Verificaiton Result ===
true, 5
...
```

where `true` indicates that the postcondition is verified (assuming the precondition holds), and `5` is the number of proven assertions.

As another example, on `test/linear_search`, the correct output is `true`, `0`.