# Homework: Verification

The goal of this assignment is to build a simple program verifier. The support code includes a parser and abstract syntax for a simple language of imperative commands, which should be the input language of your verifier. You will write a function to calculate weakest preconditions and verify that user-written assertions and loop invariants are correct using the Z3 Theorem Prover.

## 1 Restrictions

None.

## 2 Setup

To do this assignment, you will need a copy of the Z3 Theorem Prover:
`https://github.com/Z3Prover/z3/releases`

The packages on the website include Z3 bindings for several languages. However, you will only need the `z3` executable. (In particular, you do not need the OCaml bindings for Z3.)

We've written an OCaml library that let's you interface with Z3 that you can install using OPAM:

```
opam pin add z3 https://github.com/plasma-umass/ocaml-z3.git
```

## 3 Requirements

Write a program that takes one argument:
`verif.d.byte` *program*

The argument *program* should be the name of a file that contains a program written using the grammer in fig. 24.1. Your verifier should verify that the pre- and post- conditions of the program (i.e., the predicates prefixed by `requires` and `ensures`) are valid and terminate with exit code 0. If the are invalid, your verifier should terminate with a non-zero exit code.

Feel free to output whatever you need on standard in / standard error to help you debug and understand your verifier.

## 4 Support Code

The module `Imp` defines the abstract syntax of the language shown in fig. 24.1 and includes functions to parse programs in the language. The module `Smtlib` provides an API for communicating with Z3.

## 5 Directions

1. Start by ignoring loops and get your verifier working on straight-line code:

   (a) Write a function to calculate weakest-preconditions, as discussed in class and in lecture notes. You should be able to write a recursive function with the following form:

   ```
   let rec wp (cmd : Imp.cmd) (post : Imp.bexp) : Imp.bexp = match cmd with ...
   ```

**Arithmetic Expressions**

| | | | | |
|---|---|---|---|---|
| *aexp* | ::= | $n$ | `AConst n` | where $n$ is a decimal integer |
| | \| | $x$ | `AVar x` | where $x$ has letters, digits, and underscores |
| | \| | $aexp_1$`+`$aexp_2$ | `AOp(Add, aexp1, aexp2)` | |
| | \| | $aexp_1$`-`$aexp_2$ | `AOp(Sub, aexp1, aexp2)` | |
| | \| | $aexp_1$`*`$aexp_2$ | `AOp(Mul, aexp1, aexp2)` | |
| | \| | `(`*aexp*`)` | `aexp` | |
| *bexp* | ::= | `true` | `BConst true` | |
| | \| | `false` | `BConst false` | |
| | \| | $bexp_1$`&&`$bexp_2$ | `BAnd (bexp1, bexp2)` | |
| | \| | $bexp_1$`\|\|\|\|`$bexp_2$ | `BOr (bexp1, bexp2)` | |
| | \| | `!`*bexp* | `BNot bexp` | |
| | \| | $bexp_1$`<`$bexp_2$ | `BCmp (Lt, aexp1, aexp2)` | |
| | \| | $bexp_1$`<=`$bexp_2$ | `BCmp (Lte, aexp1, aexp2)` | |
| | \| | $bexp_1$`>`$bexp_2$ | `BCmp (Gt, aexp1, aexp2)` | |
| | \| | $bexp_1$`>=`$bexp_2$ | `BCmp (Gte, aexp1, aexp2)` | |
| | \| | $bexp_1$`==`$bexp_2$ | `BCmp (Eq, aexp1, aexp2)` | |
| *cmd* | ::= | `skip;` | `CSkip` | |
| | \| | `abort;` | `CAbort` | |
| | \| | $x$ `=` *aexp*`;` | `CAssign (x, aexp)` | |
| | \| | `if (`*bexp*`)` $cmd_1$ `else` $cmd_2$ | `CIf (bexp, cmd1, cmd2)` | |
| | \| | `while` $bexp_1$ `invariant` $bexp_2$ *cmd* | `CWhile (bexp1, bexp2, cmd)` | |
| | \| | `{` *cmd* `}` | `cmd` | |
| | \| | `assert` *bexp*`;` | `CIf (bexp, CSkip, CAbort)` | |
| | \| | $cmd_1$ $cmd_2$ | `CSeq (cmd1, cmd2)` | |

**Programs**

| | | | |
|---|---|---|---|
| *prog* | ::= | `requires` $bexp_1$`; ensures` $bexp_2$`;` *cmd* | `(bexp1, cmd, bexp2)` |
| | \| | `requires` $bexp_1$`;` *cmd* | `(bexp1, cmd, BConst true)` |
| | \| | `ensures` $bexp_2$`;` *cmd* | `(BConst true, cmd, bexp2)` |

Figure 24.1: The concrete syntax and abstract syntax of the language.

(b) Write a function to transform the predicates in `bexp` (i.e., preconditions) to terms for the Z3 Theorem Prover. You should be able to write a function with the following form:

```
let rec bexp_to_term (bexp : Imp.bexp) : Smtlib.term = match bexp with ...
```

(c) Using the two functions you wrote above, write a function to verify that the user-stated pre-condition implies the calculated weakest-precondition. You should be able to write a function with the following form:

```
let verify (pre : bexp) (cmd : cmd) (post : bexp) : bool = ...
```

**Note:** If the terms have variables (i.e., `AVar x`), they will have to be declared in Z3 before the term can be asserted. You can declare a Z3 variable (called a constant) as follows:

```
Smtlib.(declare_const solver (Id "x") int_sort)
```

2. Once your verifier works for loop-free programs, you should tackle loops as follows.

(a) Although the weakest-precondition of a loop is just the loop-invariant, you need to check that it truly is invariant after each iteration of the loop and after the loop exits. The simplest way to do so is to modify the `wp` function to produce a list of additional predicates that have to be proven valid for the weakest precondition to be correct. You can modify the function to have the following signature:

```
let rec wp (cmd : cmd) (post : bexp) : (bexp * bexp list) = match cmd with ...
```

(b) Modify the `verify` function to check that each predicate in the new list produced by `wp` is *valid*. We suggest checking the validity of each one independently, using `Smtlib.push` and `Smtlib.pop` before and after each assertion.

## 6 Debugging

Debugging this assignment can be challenging, since a significant portion of the work is being done by Z3. You will almost certainly have to inspect commands and output of Z3 to debug your verifier. The support code doesn't allow you to directly examine its interaction with Z3. However, instead of supplying the actual path to Z3 to `make_solver`, you can use the following shell script to log Z3 commands and responses:

```
#!/bin/bash
tee -a z3-commands.txt | z3 -in -smt2 | tee -a z3-responses.txt
```

The file `z3-commands.txt` is a valid Z3 script. Alternatively, to see commands and responses interleaved:

```
#!/bin/bash
tee -a interaction.txt | z3 -in -smt2 | tee -a interaction.txt
```

Finally, the Z3 interactive console is quite impoverished and doesn't support history, arrow keys, etc. I recommend using "rlwrap" to wrap Z3 before you try to use the console directly:

https://github.com/hanslub42/rlwrap

Rlwrap is available on standard Linux package repositories and via Homebrew for Mac OS X.