# Homework 3
# COSE312, Spring 2017

Hakjoo Oh

**Due: 05/28, 24:00**

The goal of this assignment is to implement the interpreter for **While**, an abstract machine **M**, and the translator that compiles **While** to **M**.

## 1 Language Specification

### 1.1 The While Language

The syntax of **While** is defined as follows:

$$\begin{array}{rcl} a & \to & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\ b & \to & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ S & \to & x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ S_1\ S_2 \mid \texttt{while}\ b\ S \end{array}$$

The semantics of arithmetic expressions is defined by function $\mathcal{A}[\![\ a\ ]\!]$:

$$\begin{array}{rcl} \mathcal{A}[\![\ a\ ]\!] & : & \textbf{State} \to \mathbb{Z} \\ \mathcal{A}[\![\ n\ ]\!](s) & = & n \\ \mathcal{A}[\![\ x\ ]\!](s) & = & s(x) \\ \mathcal{A}[\![\ a_1 + a_2\ ]\!](s) & = & \mathcal{A}[\![\ a_1\ ]\!](s) + \mathcal{A}[\![\ a_2\ ]\!](s) \\ \mathcal{A}[\![\ a_1 \star a_2\ ]\!](s) & = & \mathcal{A}[\![\ a_1\ ]\!](s) \times \mathcal{A}[\![\ a_2\ ]\!](s) \\ \mathcal{A}[\![\ a_1 - a_2\ ]\!](s) & = & \mathcal{A}[\![\ a_1\ ]\!](s) - \mathcal{A}[\![\ a_2\ ]\!](s) \end{array}$$

The semantics of boolean expressions is defined by function $\mathcal{B}[\![\ b\ ]\!]$:

$$\begin{array}{rcl} \mathcal{B}[\![\ b\ ]\!] & : & \textbf{State} \to \textbf{T} \\ \mathcal{B}[\![\ \texttt{true}\ ]\!](s) & = & \mathit{true} \\ \mathcal{B}[\![\ \texttt{false}\ ]\!](s) & = & \mathit{false} \\ \mathcal{B}[\![\ a_1 = a_2\ ]\!](s) & = & \mathcal{A}[\![\ a_1\ ]\!](s) = \mathcal{A}[\![\ a_2\ ]\!](s) \\ \mathcal{B}[\![\ a_1 \leq a_2\ ]\!](s) & = & \mathcal{A}[\![\ a_1\ ]\!](s) \leq \mathcal{A}[\![\ a_2\ ]\!](s) \\ \mathcal{B}[\![\ \neg b\ ]\!](s) & = & \mathcal{B}[\![\ b\ ]\!](s) = \mathit{false} \\ \mathcal{B}[\![\ b_1 \wedge b_2\ ]\!](s) & = & \mathcal{B}[\![\ b_1\ ]\!](s) \wedge \mathcal{B}[\![\ b_2\ ]\!](s) \end{array}$$

The semantics of statements is defined by function $\mathcal{C}[\![\ S\ ]\!]$:

$$\begin{array}{rcl} \mathcal{C}[\![\ S\ ]\!] & : & \mathbf{State} \hookrightarrow \mathbf{State} \\ \mathcal{C}[\![\ S\ ]\!](s) & = & \begin{cases} s' & \text{if } \langle S, s\rangle \to s' \\ \mathsf{undef} & \text{otherwise} \end{cases} \end{array}$$

where $(\to) \subseteq \mathbf{State} \times \mathbf{State}$ is defined by the big-step operational semantics:

$$\frac{}{\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![\ a\ ]\!](s)]}$$

$$\frac{}{\langle \texttt{skip}, s\rangle \to s}$$

$$\frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$$

$$\frac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{true}$$

$$\frac{\langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ S_1\ S_2, s\rangle \to s'} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{false}$$

$$\frac{\langle S, s\rangle \to s' \qquad \langle \texttt{while}\ b\ S, s'\rangle \to s''}{\langle \texttt{while}\ b\ S, s\rangle \to s''} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{true}$$

$$\frac{}{\langle \texttt{while}\ b\ S, s\rangle \to s} \text{ if } \mathcal{B}[\![\ b\ ]\!](s) = \texttt{false}$$

## 1.2 The Abstract Machine M

Let us consider an abstract machine called **M**. The instructions of **M** are given as follows:

$$\begin{array}{rcl} \mathit{inst} & \to & \mathsf{push}(n) \mid \mathsf{add} \mid \mathsf{mult} \mid \mathsf{sub} \\ & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{eq} \mid \mathsf{le} \mid \mathsf{and} \mid \mathsf{neg} \\ & \mid & \mathsf{fetch}(x) \mid \mathsf{store}(x) \\ & \mid & \mathsf{noop} \mid \mathsf{branch}(c, c) \mid \mathsf{loop}(c, c) \\ c & \to & \epsilon \mid \mathit{inst} : c \end{array}$$

where $\epsilon$ is the empty sequence. The semantics of **M** is defined by a small-step operational semantics. A configuration of **M** consists of three components:

$$\langle c, e, s\rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{State}$$

where $c$ is the sequence of instructions (code) to be executed, $e$ is the evaluation stack, and $s$ is the memory state. The evaluation stack is used to evaluate arithmetic and boolean expressions. Formally, it is a list of values:

$$\mathbf{Stack} = (\mathbb{Z} \cup \mathbf{T})^*$$

$$
\begin{array}{lcl}
\langle \mathsf{push}(n) :: c, e, s\rangle & \rhd & \langle c, n :: e, s\rangle \\
\langle \mathsf{add} :: c, z_1 :: z_2 :: e, s\rangle & \rhd & \langle c, (z_1 + z_2) :: e, s\rangle \\
\langle \mathsf{mult} :: c, z_1 :: z_2 :: e, s\rangle & \rhd & \langle c, (z_1 \star z_2) :: e, s\rangle \\
\langle \mathsf{sub} :: c, z_1 :: z_2 :: e, s\rangle & \rhd & \langle c, (z_1 - z_2) :: e, s\rangle \\
\langle \mathsf{true} :: c, e, s\rangle & \rhd & \langle c, \mathit{true} :: e, s\rangle \\
\langle \mathsf{false} :: c, e, s\rangle & \rhd & \langle c, \mathit{false} :: e, s\rangle \\
\langle \mathsf{eq} :: c, z_1 :: z_2 :: e, s\rangle & \rhd & \langle c, (z_1 = z_2) :: e, s\rangle \\
\langle \mathsf{le} :: c, z_1 :: z_2 :: e, s\rangle & \rhd & \langle c, (z_1 \leq z_2) :: e, s\rangle \\
\langle \mathsf{and} :: c, t_1 :: t_2 :: e, s\rangle & \rhd & \begin{cases} \langle c, \mathit{true} :: e, s\rangle & \text{if } t_1 = \mathit{true} \text{ and } t_2 = \mathit{true} \\ \langle c, \mathit{false} :: e, s\rangle & \text{otherwise} \end{cases} \\
\langle \mathsf{neg} :: c, t :: e, s\rangle & \rhd & \begin{cases} \langle c, \mathit{true} :: e, s\rangle & \text{if } t = \mathit{false} \\ \langle c, \mathit{false} :: e, s\rangle & \text{otherwise} \end{cases} \\
\langle \mathsf{fetch}(x) :: c, e, s\rangle & \rhd & \langle c, s(x) :: e, s\rangle \\
\langle \mathsf{store}(x) :: c, z :: e, s\rangle & \rhd & \langle c, e, s[x \mapsto z]\rangle \\
\langle \mathsf{noop} :: c, e, s\rangle & \rhd & \langle c, e, s\rangle \\
\langle \mathsf{branch}(c_1, c_2) :: c, t :: e, s\rangle & \rhd & \begin{cases} \langle c_1 :: c, e, s\rangle & \text{if } t = \mathit{true} \\ \langle c_2 :: c, e, s\rangle & \text{otherwise} \end{cases} \\
\langle \mathsf{loop}(c_1, c_2) :: c, e, s\rangle & \rhd & \langle c_1 :: \mathsf{branch}(c_2 :: \mathsf{loop}(c_1, c_2), \mathsf{noop}) :: c, e, s\rangle
\end{array}
$$

Figure 1: Operational semantics of **M**

A memory state $s$ is a mapping from variables to values. A configuration is a terminal configuration if its code component is the empty sequence: i.e., $\langle \epsilon, e, s\rangle$.

The semantics of the instructions of the abstract machine is given by the transition relation $\rhd$, which specifies how to execute the instructions:

$$\langle c, e, s\rangle \rhd \langle c', e', s'\rangle$$

The semantics is given in Figure 1. We use the notation :: for both appending two instruction sequences and prepending an element to a sequence. The evaluation stack has the top element to the left, and the empty stack is represented by $\epsilon$. We define the semantics of a sequence of instructions by the partial function:

$$
\begin{array}{rcl}
\mathcal{M}[\![\, c \,]\!] & : & \mathbf{State} \hookrightarrow \mathbf{State} \\
\mathcal{M}[\![\, c \,]\!](s) & = & \begin{cases} s' & \text{if } \langle c, \epsilon, s\rangle \to \langle \epsilon, e, s'\rangle \\ \mathsf{undef} & \text{otherwise} \end{cases}
\end{array}
$$

# 2 Implementation

Clone the Git repository for programming assignments:

```
git clone https://github.com/kupl/Compiler2017.git
```

where you can find the `hw3` directory.

## 2.1 (20pts) The interpreter for While

In `while.ml`, the syntax and semantics of **While** are defined as follows:

```
type stmt = ASSIGN of var * aexp
          | SKIP
          | SEQ of stmt * stmt
          | IF of bexp * stmt * stmt
          | WHILE of bexp * stmt
          | READ of var
          | PRINT of aexp
and aexp  = NUM of int
          | VAR of var
          | ADD of aexp * aexp
          | SUB of aexp * aexp
          | MUL of aexp * aexp
and bexp  = TRUE | FALSE
          | EQ of aexp * aexp
          | LE of aexp * aexp
          | AND of bexp * bexp
          | NEG of bexp
and var = string
type program = stmt

type state = (var, int) PMap.t
let state_empty = PMap.empty
let state_lookup x s = PMap.find x s
let state_bind x v s = PMap.add x v s

let eval_stmt : stmt -> state -> state
=fun c s -> s (* TODO *)

let run : stmt -> unit
=fun pgm -> ignore (eval_stmt pgm state_empty)
```

Complete the definition `eval_stmt`, which implements the big-step operational semantics of **While**.

## 2.2 (30pts) The interpreter for M

In `m.ml`, the syntax and semantics of **M** are defined as follows:

```
type inst =
  | Push of int
  | Add
  | Mul
  | Sub
```

```
  | True
  | False
  | Eq
  | Le
  | And
  | Neg
  | Fetch of var
  | Store of var
  | Noop
  | Branch of cmd * cmd
  | Loop of cmd * cmd
  | Read
  | Print

and cmd = inst list
and var = string

type value = Z of int | T of bool
type stack = value list
type state = (var, int) PMap.t

let state_empty = PMap.empty
let state_lookup x s = PMap.find x s
let state_bind x v s = PMap.add x v s

let next : inst list * stack * state -> inst list * stack * state
=fun (c,e,s) -> ([],e,s) (* TODO *)

let run : cmd -> state
=fun c ->
  let iconf = (c, [], state_empty) in
  let rec iter (c,e,s) =
    match next (c,e,s) with
    | [], _, s  -> s
    | c', e',s' -> iter (c',e',s') in
    iter iconf
```

Complete the definition `next`, which implements the small-step operational semantics (i.e., the $\triangleright$ relation).

## 2.3 (30pts) Translator

Finally, let us define the translator that compiles **While** to **M**. This translation function can be defined as follows:

$$\mathcal{T} : \textbf{Stmt} \to \textbf{Code}$$

where **Stmt** is a statement of **While**, and **Code** is an instruction sequence of **M**. The translation must be correct; for any statement $S$ of **While** and a memory state $s$, the following equality must hold:

$$\mathcal{C}[\![ S ]\!](s) = \mathcal{M}[\![ \mathcal{T}(S) ]\!](s). \tag{1}$$

Implement $\mathcal{T}$ in `translation.ml`:

```
let trans : stmt -> inst list
=fun c -> [] (* TODO *)
```

# 3 (Optional) Correctness Proof

Formally define the translation function $\mathcal{T}$ and prove its correctness in (1). The definition and proof must be mathematically precise and rigorous.

This is an optional assignment, which gets you extra credits. If everything is properly done, your final grade will be upgraded (e.g., A $\rightarrow$ A+). Submit your proof as a hard copy in class on Monday (5/29).