**UVAID :** AM2QA

**Exercise 1**

I spent about 5.5 hours on the homework (skimming ocaml tutorials included). I found *criticizing Hoare' s essay* the hardest since most of his ideas are fairly acceptable. Also writing down small-steps opsems in a granularity that is symmetric to our discussion was a bit tricky. I feel that the assignment is in the *moderate difficulty* level.

---

**Exercise 2**

It is evident from modern Programming Languages that Hoare's hints have been a strong influence on the way languages have evolved. In that sense, his advocacy for simplicity, security, speed have becomes the cliches of Programming Language discussions. I did find it difficult to empathize with several problems that he seemed to face as a programmer simply because of how rapidly programming languages have evolved and invented new problems along the way. On account of that, I wish to analyze Hoare's essay as is applicable to more recent languages.

The advocated *block structure* is among the most important Programming Language constructs. DRY (do not repeat yourself), locality of function, orthogonality of modules are just some of the programming philosophies that have benefited out of having blocks. In more recent times, blocks (knows as closures in prototype based languages) have shaped the way we think about treating methods as first-class entities in functional languages. Co-routines make event-driven/ asynchronous programming possible in languages like Ruby and Javascript by helping programmers avoid writing spaghetti-code. Event driven I/O frameworks like Nodejs and EventMachine which have become synonymous with their implementation-languages (javascript and Ruby) are examples of beneficiaries.

One of the over-emphasized suggestions that I think is only supplementary to a Programming Language is *Fast Translation*. In the `real world`, the build/compile stage extends well beyond the actual code-compilation process. Unit Testing as a practice, is used by programmers to extend the type system of the programming language into a DSL. As a consequence running these unit-tests typically fits into the 'build' process. This tends consumes 10X times the amount of time taken by the compiler to do its part of the 'building'. In that sense, fast translation is only a supplementary feature -- especially in dynamic languages. Although its significance is not debatable, the essay over-emphasizes this factor considering that this process is rarely isolated and has other ingredients that are significantly more time consuming and equally important.

**Bonus 4th paragraph**
Critically speaking, in an essay about the inherent properties of a programming language, speaking about fast translation and other peripheral features of a language (in this case the compiler and debugger) is asymmetric with the rest of the essay.

---

**Exercise 3**
Introducing a division operation with the Aexp sub-language bears more consequences than other operations. Since division operations are prone to "Divide by Zero" error, this needs to be accommodated by one of the 2 ways:
1) Introducing side rules that ensure that in an operation $a_0/a_1$, a1 is never a zero. However this approach may lead to cases where the system will find no proof derivation for divide by zero operations.
2) Extending the syntactic sets and have a new ERROR set that can identify erroneous operations. However, we will need to rewrite/extend the existing rules to allow values in this ERROR set to propagate through the existing syntactic constructs.

New Syntactic Set E where element e0 is the 'divide by zero' error.
**Rules to incorporate division**
$(e, \sigma) \to e$
$(a, \sigma) \to n|e$
where e is an error (division by zero in this case).

**Addition**
1)
$$\frac{(a_0, \sigma) \to n_0 \quad (a_1, \sigma) \to n_1}{(a_0 + a_1, \sigma) \to n}$$
where n is the sum of n0 and n1

2)
$$\frac{(a_0, \sigma) \to e_0}{(a_0 + a_1, \sigma) \to e_0} \qquad \frac{(a_1, \sigma) \to e_0}{(a_0 + a_1, \sigma) \to e_0}$$

where $e_0$ is a divide by zero error. Both are short circuited operations.

**Subtraction**

$$\frac{(a_0\sigma) \rightarrow n_0 \quad (a_1\sigma) \rightarrow n_1}{(a_0 - a_1) \rightarrow n}$$

where n is difference between $n_0$ and $n_1$

$$\frac{(a_1, \sigma) \rightarrow e_0}{(a_0 - a_1, \sigma) \rightarrow e_0} \qquad \frac{(a_0, \sigma) \rightarrow e_0}{(a_0 - a_1, \sigma) \rightarrow e_0}$$

where $e_0$ is a divide by zero error. Both are short circuited operations.

**Products**

$$\frac{(a_0, \sigma) \rightarrow n_0 \quad (a_1, \sigma) \rightarrow n_1}{(a_0 * a_1, \sigma) \rightarrow n}$$

where n is the product of $n_0$ and $n_1$

$$\frac{(a_0, \sigma) \rightarrow e_0}{(a_0 * a_1, \sigma) \rightarrow e_0} \qquad \frac{(a_1, \sigma) \rightarrow e_0}{(a_0 * a_1, \sigma) \rightarrow e_0}$$

where $e_0$ is a divide by zero error. Both are short circuited operations.

**Division**

$$\frac{(a_0, \sigma) \rightarrow n_0 \quad (a_1, \sigma) \rightarrow n_1}{(a_0 / a_1, \sigma) \rightarrow n}$$

where n is result of dividing $n_0$ by $n_1$ . These can be short circuited if the numerators are zero (Not done here though).

$$\frac{(a_0, \sigma) \rightarrow e_0}{(a_0 / a_1, \sigma) \rightarrow e_0} \qquad \frac{(a_1, \sigma) \rightarrow e_0}{(a_0 / a_1, \sigma) \rightarrow e_0}$$

where $e_0$ is a divide by zero error. Both are short circuited operations.

$$\frac{(a_1, \sigma) \rightarrow 0}{(a_0 / a_1, \sigma) \rightarrow e_0}$$

Note that this is the side rule where the error originates. This is not a rule instance but the abstract-rule itself.

**Exercise 4**

1)Natural Style opsem extension

$$\frac{(a_0,\sigma_0)\rightarrow m \qquad \sigma_0(X)\rightarrow n \qquad \sigma_0(m|X)\rightarrow\sigma_3 \qquad (c_0;c_1,\sigma_3)\rightarrow\sigma_2 \qquad (X:=n,\sigma_2)\rightarrow\sigma_1}{(Let\ X:=a_0\ in\ c_0;c_1,\sigma_0)\rightarrow\sigma_1}$$

2)Contextual Style opsem extensions

**--New reduction rule**:

$< Let\ x=n\ in\ c,\ \sigma > \rightarrow < temp:=x;\ x:=n;\ c;\ x:=temp,\ temp:= 0,\ \sigma >$

where x belongs to LOC.

Note that temp (ideally) must return to its *zero-initialized* state.

**--Redexes**

r :== X
| n1 + n2
| x := n
| skip; c
| if true then c1 else c2
| if false then c1 else c2
| while b do c
| let x = n in $c_0$

**--Contexts** (remain the same)

H ::= •
| n + H
| H + e
| x := H
| if H then c1 else c2
| H; c

Example

| Comm, State | Redex | Context |
|---|---|---|
| <{let x=3 in p:=x+3}; y :=4> | {let x=3 in p:=x+3} | ▪ ; y:=4 |
| <temp:=x; x:=3; p:=x+3; x:= temp; temp :=0> | x | temp:= ▪ ; x:=3;p:=x+3... |

There are other ways this can be written, but I have tried to retain the same level of granularity as our discussion in class.