Reading
Read Chapter 5 of our textbook *Compilers*

Written Assignment 5

1. Exercise 5.1.2 (page 310).
2. Exercise 5.2.4 (page 317).
3. Exercise 5.4.1 (page 336).
4. Exercise 5.4.3.

Programming Assignment 5

To our previous language we add *let* expressions, resulting in fairly unassuming Scheme (FUS) programs:

prog	→	*expr*+
expr	→	DOUBLE
	|	BOOLEAN
	|	ID
	|	'(' RATOR *expr*∗ ')'
	|	'(' 'def' ID *expr* ')'
	|	'(' 'if' $expr_1$ $expr_2$ $expr_3$ ')'
	|	'(' 'print' *expr* ')'
	|	'(' 'let' *letvardec expr* ')'
	|	'(' 'while' $expr_1$ $expr_2$ ')'
	|	'(' 'begin' *expr*+ ')'
letvardec	→	'[' ID *expr* ']'
BOOLEAN	→	'true' | 'false'
RATOR	→	ARITHMETIC | RELATIONAL | LOGICAL
ARITHMETIC	→	'+' | '–' | '*' | '/'
RELATIONAL	→	'=' | '>' | '<'
LOGICAL	→	'&' | '|' | '!'

A *let* expression has the form (let [*x* $expr_1$] $expr_2$). Here, in the *let variable declaration* [*x* $expr_1$], *x* is called a *let variable* and $expr_1$ is its *let operand*. The expression $expr_2$ is called the *let body*. In the following, *x* is the *let* variable, 2 is the *let* operand, and (+ *x x*) is the *let* body:

```
(let [x 2] (+ x x))
```

Semantically, each *let* operand is evaluated in the environment of the *let* expression and bound to the corresponding *let* variable. The environment consists of a series of nested frames, each of which contains a variable-value pair called a *binding.* To evaluate a *let* expression, the current environment is extended by a new frame that contains the *let* variable binding, and then the *let* body is evaluated in this new environment and the resulting value is returned. For example, consider this:

```
(let [x (* 2 2)] (+ x x))
```

Here the *global environment,* in which the *let* expression gets evaluated, is extended by a frame in which the *let* variable *x* is bound to the value 4 (which results from evaluating (* 2 2)), and then the *let* body (+ *x x*) is evaluated in this new environment, yielding the value 8. Next consider this:

```
(let [a 2]
  (let [b (+ a 1)]
    (+ a b)))
```

The body of the outer *let* expression is evaluated in an environment that extends the global environment with a frame in which *let* variable *a* is bound to (has the value) 2; let us call this environment *E*. Then, the *let* operand (+ a 1) is evaluated in environment *E,* yielding the value 3, which gets bound to let variable *b;* so environment *E* is extended by a frame in which *b* is bound to 3; call this environment *F*. When (+ a b), the body of the inner *let,* is evaluated in the environment *F*—which supplies the values 2 and 3 for *let* variables *a* and *b* respectively—it yields the result 5.

I will illustrate program semantics by a series of evaluations. We can use our discussion board to resolve any remaining questions of program semantics:

```
> java run
(let [a 4] a)
^Z
4.0

> java run
(def a 6)
(let [a 4] (print a))
a
^Z
4.0
6.0

> java run
(let [a (+ 3 4)]
   (let [b (* a a)]
      (+ a b)))
56
```

```
> java run
(def a 4)
(print a)
(let [a (+ a a)]
  (begin
    (print a)
    (def a 17)
    (print a)))
a
^Z
4.0
8.0
8.0
17.0
```

In the last example, note that variable assignment (using *def*) always creates a binding in the *top-level* environment, even when the assignment appears inside a *let* expression. The last print statement prints 4 since the *let* variable *a* was not affected by the assignment; only the global variable's value was changed.

In class, we discussed blocks that support declaration of local variables, and their implementation through environments formed by a series of nested scopes. You might use that approach in your implementation of *let* expressions.