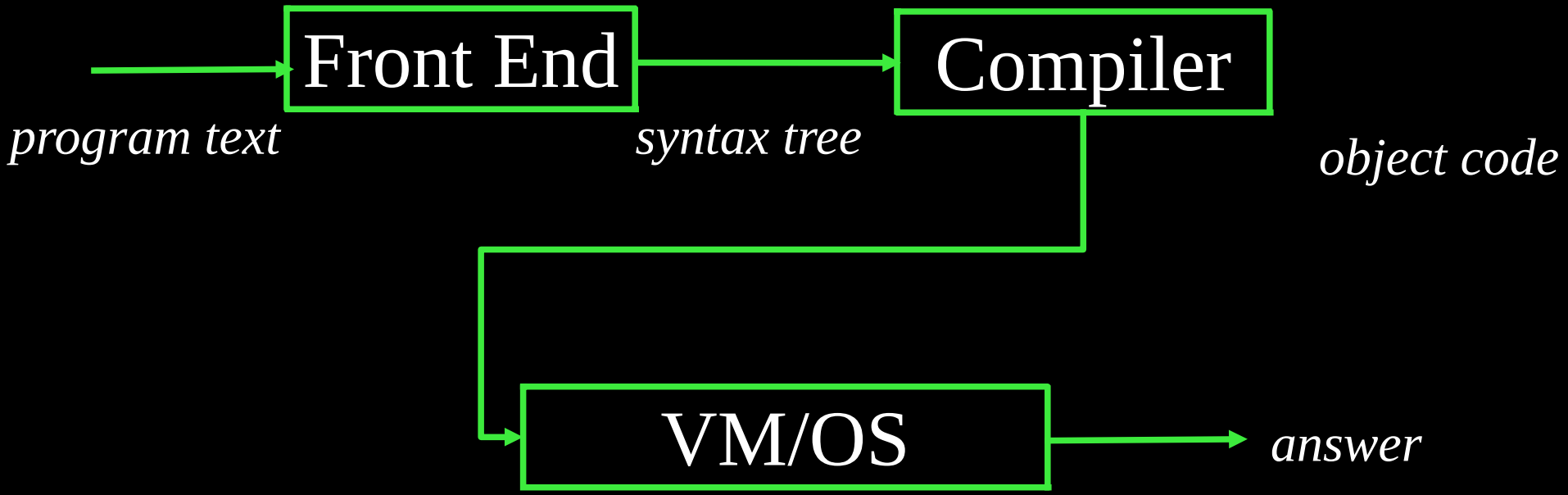
Chapter 3: Environment-Passing Interpreters

Compiler (javac) vs. *Interpreter* (drscheme):


```
> ((lambda (x) (* x x)  6)
36
```




```
% javac square6.java
% java square6
36

% g++ -o square6 square6.cpp
% ./square6
36
```

3.1 A Simple Interpreter

- *Defining language* will be Scheme (of course).
- *Defined language* will be simple arithmetic expressions:

```
3
x
+(3,x)
add1(+(3,x))
```

A Simple Interpreter

Syntax:

<program> ::= <expression>

```
a-program (exp)
```

<expression> ::= <number>

```
lit-exp (datum)
```

::= <identifier>

```
var-exp (id)
```

::= <primitive> **(**{<expression>}*(,)**)**

```
primapp-exp (prim rands)
```

<primitive> ::= **+ | - | * | add1 | sub1**

A Simple Interpreter

```
(define-datatype program program?
  (a-program
    (exp expression?))

(define-datatype expression expression?
  (lit-exp
    (datum number?))
  (var-exp
    (id symbol?))
  (primapp-exp
    (prim primitive?)
    (rands (list-of expression?)))
```

A Simple Interpreter

```
(define-datatype primitive primitive?
  (add-prim)
  (subtract-prim)
  (mult-prim)
  (incr-prim)
  (decr-prim))


> (a-program          ; add1(+(3,x))
    (primapp-exp
        (incr-prim)
        ((primapp-exp
            (add-prim)
            ((lit-exp 3) (var-exp 'x))))))
```

Interpreter Code

```
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (body)
        (eval-expression body (init-env))))))


(define init-env
  (lambda ()
    (extend-env
      '(i v x)
      '(1 5 10)
      (empty-env))))
```

Interpreter Code

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args))))))
```

Interpreter Code

```
(define eval-rands
  (lambda (rands env)
    (map (lambda (x) (eval-rand x env))rands)))

(define eval-rand
  (lambda (rand env)
    (eval-expression rand env)))
```

Interpreter Code

```
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
      (add-prim      () (+ (car args)
                           (cadr args)))
      (subtract-prim () (- (car args)
                           (cadr args)))
      (mult-prim     () (* (car args)
                           (cadr args)))
      (incr-prim     () (+ (car args) 1))
      (decr-prim     () (- (car args) 1)))))
```