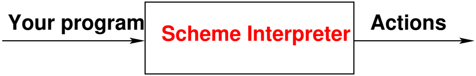
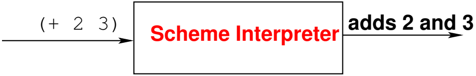
Running a Scheme Program

A program called a Scheme Interpreter takes your Scheme program as input and carries out the actions described by your program.



Running a Scheme Program

As examples:



`(+ 2 3)`

Scheme Interpreter

adds 2 and 3

`(+ 2 3)`

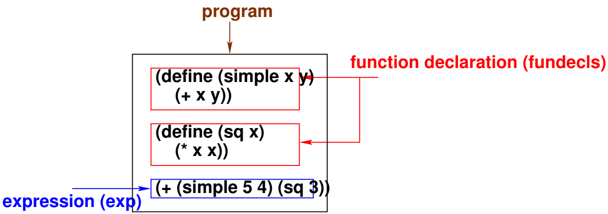
Scheme Interpreter

adds 2 and 3

Let us write a Tiny-Scheme interpreter in Scheme.

Representing Tiny Scheme in Scheme

How does one represent a Tiny-Scheme program in Scheme?

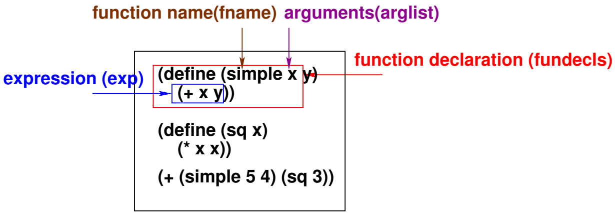


Structure of a program:

```
(struct program (fundecls exp) #:transparent)
```

Representing Tiny Scheme in Scheme

Function declarations

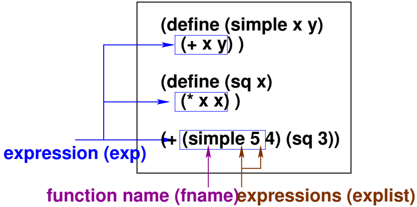


```
(define (simple x y)
  (+ x y))

(define (sq x)
  (* x x))

(+ (simple 5 4) (sq 3))
```

```
(struct fundecl (fname arglist exp) #:transparent)
```

Representing Tiny Scheme in Scheme

Expressions

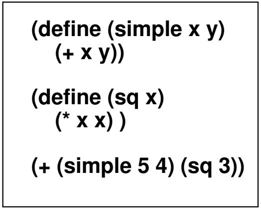


```
(struct application (name explist) #:transparent)
(struct add_(exp1 exp2)  #:transparent)
(struct sub_(exp1 exp2) #:transparent)
(struct mul_(exp1 exp2) #:transparent)
(struct eq_(exp1 exp2)  #:transparent)
(struct if_(bexp exp1 exp2) #:transparent)
```

Putting it together

```
(struct program (fundecls exp) #:transparent)
(struct fundecl (fname arglist exp) #:transparent)
(struct application (name explist) #:transparent)
(struct add_(exp1 exp2)  #:transparent)
(struct sub_(exp1 exp2) #:transparent)
(struct mul_(exp1 exp2) #:transparent)
(struct eq_(exp1 exp2)  #:transparent)
(struct if_(bexp exp1 exp2) #:transparent)
```

Putting it together



```
(define (simple x y)
   (+ x y))

(define (sq x)
   (* x x) )

(+ (simple 5 4) (sq 3))
```

```
(define fd1 (fundecl 'simple (list 'x 'y)
                 (add_ 'x 'y)))
(define fd2 (fundecl 'sq (list 'x)
                 (mul_ 'x 'x)))
(define program1 (program (list fd1 fd2)
                       (add_ (application 'simple (list 5 4))
                             (application 'sq (list 3)))))
```

Tiny-Scheme Interpreter – eval-program

- eval-program – The part of Tiny-Scheme interpreter which processes a program.
- First processes function declarations.
- Creates an environment in which every function is tied to its lambda.
- The main expression is evaluated in this environment.

```
(define (eval-program prog)
    ...
    (define (eval-exp e) ...)
    ...
    set up an initial environment initenv
    (eval-exp (program-exp prog) initenv))
```

eval-exp processes expressions.

Tiny-Scheme Interpreter – initenv

For the program:

```
(define (simple x y)
   (+ x y))
(define (sq x)
   (* x x))
(+ (simple 5 4) (sq 3))
```

eval-program calls eval-exp to evaluate (+ (simple 5 4) (sq 3)) in an environment in which:

- simple is bound to (lambda (x y) (+ x y)) and
- sq is bound to (lambda (x) (* x x))

We shall call this the global environment.

Tiny-Scheme Interpreter – `initenv`

For the program:

```
(define (simple x y)
   (+ x y))
(define (sq x)
   (* x x))
(+ (simple 5 4) (sq 3))
```

`initenv` is:

```
((simple #(struct:lambda_ (x y) #(struct:add_ x y)))
 (sq #(struct:lambda_ (x) #(struct:mul_ x x))))
```

`eval-program` calls `eval-exp` to evaluate `(+ (simple 5 4) (sq 3))` in `initenv`.

Tiny-Scheme Interpreter – eval-program

Note: The only free variables in the lambdas are the (globally declared) functions.

- eval-program – The part of Tiny-Scheme interpreter which processes a program.
- First processes function declarations.
- Creates an environment in which every function is tied to its lambda.
- The main expression is evaluated in this environment.

```
(define (eval-program prog)
    ...
    (define (eval-exp e) ...)
    ...
    set up an initial environment initenv
    (eval-exp (program-exp prog) initenv))
```

eval-exp processes expressions.

Tiny-Scheme Interpreter – eval-exp

In general, eval-exp evaluates an expression in a given environment.

- If the expression is a number, the result of the evaluation is the number itself.
- If the expression is a variable, the result is the binding of the variable in the environment.
- ...

```
(define (eval-exp exp env)
  (cond ((number? exp) exp)
        ((symbol? exp) ...)
        ...))
```

Tiny-Scheme Interpreter – eval-exp

- If the expression is (+ exp1 exp2), the result is the addition of the evaluations of exp1 and exp2.
- If the expression is (if bexp exp1 exp2), then the result is the evaluation of exp1 or exp2, depending on the value of bexp.

```
(define (eval-exp exp env)
  (cond ((number? exp) exp)
        ((symbol? exp) ...)
        ((add_? exp) (+ ...))
        ((sub_? exp) ...)
        ((mul_? exp) ...)
        ((eq_? exp)  ...)
        ((if_? exp) ...)
        ...))
```

The if of Tiny-scheme is being implemented through the if of Drracket..

Tiny-Scheme Interpreter – eval-exp

Evaluation of (simple (+ 3 4) 5) is a call to apply_ with (lambda (x y) (+ x y)) and the list of evaluated arguments 7 and 5.

```
(define (eval-exp exp env)
  (cond ((number? exp) exp)
        ((symbol? exp) (cadr (assq exp env)))
        ((add_? exp) (+ (eval-exp (add_-exp1 exp) env)
                        (eval-exp (add_-exp2 exp) env)))
        ((sub_? exp) ...
        ((mul_? exp) ...
        ((eq_? exp)  ...
        ((if_? exp)  ...


        ((application? exp) (apply_ ...  ...))
```

Tiny-Scheme Interpreter – handling applications

Finally

- apply_ evaluates the body of the lambda passed to it in the global environment extended with the bindings of the parameters.
- apply_ (lambda (x y) (+ x y)) (7 5) results in the evaluation of (+ x y) in the environment formed by extending the global environment with the bindings of x and y to 7 and 5.

```
(define (apply_ lam vallist)
  (eval-exp (lambda_-exp lam) ...))
```

Tiny-Scheme Interpreter – handling applications

Extend the interpreter to:

- Handle functions taking a function as an argument. For example:

```
(define (f g x) (g x))
```

- A let expression represented by the struct:

```
(struct let_ (var defn exp))
```