Scheme Programs as Data
Class outline:

• Eval
• Quasiquotation
• Generating code
• Apply
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: \[2 \ 3.3 \ #t \ + \ \text{quotient}\]
- Combinations: \[(\text{quotient} \ 10 \ 2) \ (\text{not} \ #t)\]

The built-in Scheme list data structure can represent combinations:

\[(\text{list} \ \text{quotient} \ 10 \ 2)\]
A Scheme Expression is a Scheme List

Scheme programs consist of expressions, which can be:

- Primitive expressions: \[2\ 3.3\ \#t\ +\ \textit{quotient}\]
- Combinations: \((\textit{quotient\ 10\ 2})\ (\textit{not\ \#t})\)

The built-in Scheme list data structure can represent combinations:

\[(\text{list}\ '\textit{quotient}\ 10\ 2)\ ;\ (\textit{quotient\ 10\ 2})\]
The **eval** procedure evaluates a given expression in the current environment.

```
(eval <expression>)
```

```
(eval (list 'quotient 10 2))
```
The `eval` procedure evaluates a given expression in the current environment.

```
(eval <expression>)
```

```
(eval (list 'quotient 10 2)) ; 5
```
The eval procedure

The eval procedure evaluates a given expression in the current environment.

\[
\text{(eval <expression>)}
\]

\[
\text{(eval (list 'quotient 10 2)) ; 5}
\]

Quote supresses evaluation, while eval forces evaluation. They can cancel each other out!

\[
\text{(define x 3)}
\text{'x}
\text{(eval 'x)}
\]
The **eval** procedure evaluates a given expression in the current environment.

```
(eval <expression>)
```

```
(eval (list 'quotient 10 2)) ; 5
```

Quote suppresses evaluation, while eval forces evaluation. They can cancel each other out!

```
(define x 3)

'x ; x
```

```
(eval 'x)
```
The eval procedure

The `eval` procedure evaluates a given expression in the current environment.

```scheme
(eval <expression>)
```

```scheme
(eval (list 'quotient 10 2)) ; 5
```

Quote supresses evaluation, while eval forces evaluation. They can cancel each other out!

```scheme
(define x 3)
'x ; x
(eval 'x) ; 3
```
Generating call expressions
Generating factorial expressions

Compare standard factorial:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(fact 5) ; 120
```
Generating factorial expressions

Compare standard factorial:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))))

(fact 5) ; 120
```

...to a version that generates an expression:

```
(define (fact-exp n)
  (if (= n 0)
      1
      (list '*
            n
            (fact-exp (- n 1))))

(fact-exp 5)
(eval (fact-exp 5))
```
Generating factorial expressions

Compare standard factorial:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(fact 5) ; 120
```

...to a version that generates an expression:

```
(define (fact-exp n)
  (if (= n 0)
      1
      (list '*
            n
            (fact-exp (- n 1)))))

(fact-exp 5) ; (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
(eval (fact-exp 5))
```
Generating factorial expressions

Compare standard factorial:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(fact 5) ; 120
```

...to a version that generates an expression:

```
(define (fact-exp n)
  (if (= n 0)
      1
      (list '*
            n
            (fact-exp (- n 1)))))

(fact-exp 5) ; (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(eval (fact-exp 5)) ; 5
```
Generating virfib expressions

Compare standard Virahanka-Fibonacci:

```scheme
(define (virfib n)
  (if (<= n 1)
      n
      (+ (virfib (- n 2)) (virfib (- n 1))))
)

(virfib 6) ; 8
```
Generating virfib expressions

Compare standard Virahanka-Fibonacci:

```scheme
(define (virfib n)
  (if (<= n 1)
      n
      (+ (virfib (- n 2)) (virfib (- n 1)))))

(virfib 6) ; 8
```

...to a version that generates an expression:

```scheme
(define (virfib-exp n)
  (if (<= n 1)
      n
      (list '+ (virfib-exp (- n 2)) (virfib-exp (- n 1))))

(virfib-exp 6)
(eval (virfib-exp 6))
```
Generating virfib expressions

Compare standard Virahanka-Fibonacci:

(define (virfib n)
  (if (<= n 1)
      n
      (+ (virfib (- n 2)) (virfib (- n 1)))))

(virfib 6) ; 8

...to a version that generates an expression:

(define (virfib-exp n)
  (if (<= n 1)
      n
      (list '+ (virfib-exp (- n 2)) (virfib-exp (- n 1)))))

(virfib-exp 6) ; (+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1) (+ 0 1)) (eval (virfib-exp 6)))
Generating virfib expressions

Compare standard Virahanka-Fibonacci:

```scheme
(define (virfib n)
  (if (<= n 1)
      n
      (+ (virfib (- n 2)) (virfib (- n 1))))
)
```

```scheme
(virfib 6) ; 8
```

...to a version that generates an expression:

```scheme
(define (virfib-exp n)
  (if (<= n 1)
      n
      (list '+ (virfib-exp (- n 2)) (virfib-exp (- n 1))))
)
```

```scheme
(virfib-exp 6) ; (+ (+ (+ 0 1) (+ 1 (+ 0 1))) (+ (+ 1 (+ 0 1)) 0)
(eval (virfib-exp 6)) ; 8
```
Generating programs
Quasiquotation

There are two ways to quote an expression:

<table>
<thead>
<tr>
<th>Quote</th>
<th>`(a b) → (a b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quasiquote</td>
<td>`(a b) → (a b)</td>
</tr>
</tbody>
</table>

They are different because parts of a quasiquoted expression can be **unquoted** with `,`.

```
(define b 4)
```

<table>
<thead>
<tr>
<th>Quote</th>
<th>`(a ,(+ b 1)) → (a (unquote (+ b 1)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quasiquote</td>
<td>`(a ,(+ b 1)) → (a 5)</td>
</tr>
</tbody>
</table>
Generating code with quasiquotation

Quasiquotation is particularly convenient for generating Scheme expressions:

```scheme
(define (make-adder n) `(lambda (d) (+ d ,n)))
(make-adder 2)
```
Generating code with quasiquote

Quasiquote is particularly convenient for generating Scheme expressions:

```
(define (make-adder n) `\(\lambda\ (d) (+\ d\ ,\ n)\))

(make-adder 2) \;:\ (\lambda\ (d) (+\ d\ 2))
```
Generating code with quasiquotation

Quasiquotation is particularly convenient for generating Scheme expressions:

```scheme
(define (make-adder n) `(lambda (d) (+ d ,n)))
(make-adder 2) ; (lambda (d) (+ d 2))
```

Remember, the generated expression is a Scheme list:

```scheme
(define new-func (make-adder 2))
new-func ; (lambda (d) (+ d 2))
(list? new-func)
(car new-func)
```
Generating code with quasiquote

Quasiquote is particularly convenient for generating Scheme expressions:

```
(define (make-adder n) `(lambda (d) (+ d ,n)))
```

```
(make-adder 2) ; (lambda (d) (+ d 2))
```

Remember, the generated expression is a Scheme list:

```
(define new-func (make-adder 2))
```

```
new-func ; (lambda (d) (+ d 2))
(list? new-func) ; #t
(car new-func)
```
Generating code with quasiquotation

Quasiquotation is particularly convenient for generating Scheme expressions:

```
(define (make-adder n) `(lambda (d) (+ d ,n)))
(make-adder 2) ; (lambda (d) (+ d 2))
```

Remember, the generated expression is a Scheme list:

```
(define new-func (make-adder 2))
new-func ; (lambda (d) (+ d 2))
(list? new-func) ; #t
(car new-func) ; lambda
```
Example: While loops

Calculate the sum of the squares of even numbers less than 10, starting with 2

```python
x = 2
total = 0
while x < 10:
    total = total + x * x
    x = x + 2
```
Example: While loops

Calculate the sum of the squares of even numbers less than 10, starting with 2

\[
x = 2 \\
total = 0 \\
while x < 10:
    total = total + x \times x \\
x = x + 2 \\
\]

(begin (define (loop x total)
    (if (< x 10)
        (loop (+ x 2) (+ total (* x x)))
        total))
    (loop 2 0))
**Example: While loops**

Calculate the sum of the squares of even numbers less than 10, starting with 2

```plaintext
x = 2
total = 0
while x < 10:
    total = total + x * x
    x = x + 2
```

Calculate the sum of numbers whose squares are less than 50, starting with 1

```plaintext
(define (loop x total)
    (if (< x 10)
        (loop (+ x 2) (+ total (* x x)))
        total))
(begin (loop 2 0))
```
Example: While loops

Calculate the sum of the squares of even numbers less than 10, starting with 2

\[
x = 2 \\
total = 0 \\
while x < 10: \\
    total = total + x * x \\
    x = x + 2
\]

\[
(begin (define (loop x total) 
    (if (< x 10) 
        (loop (+ x 2) (+ total (* x x))) 
        total)) 
    (loop 2 0))
\]

Calculate the sum of numbers whose squares are less than 50, starting with 1

\[
x = 1 \\
total = 0 \\
while x * x < 50: \\
    total = total + x \\
    x = x + 1
\]
Example: While loops

Calculate the sum of the squares of even numbers less than 10, starting with 2

\[
\begin{align*}
\text{begin} & \quad \text{(define } \text{ (loop } x \text{ total)}) \\
& \quad (\text{if } (< x 10) \\
& \quad \quad (\text{loop } (+ x 2) (+ \text{ total } (* x x))) \\
& \quad \quad \text{total})) \\
& \quad (\text{loop } 2 \ 0)) \\
\end{align*}
\]

\[
\begin{align*}
x & = 2 \\
total & = 0 \\
\textbf{while} & \quad x < 10: \\
& \quad \quad \text{total} = \text{total} + x \times x \\
x & = x + 2
\end{align*}
\]

Calculate the sum of numbers whose squares are less than 50, starting with 1

\[
\begin{align*}
\text{begin} & \quad \text{(define } \text{ (loop } x \text{ total)}) \\
& \quad (\text{if } (< \text{(* } x \ x \text{)} 50) \\
& \quad \quad (\text{loop } (+ x 1) \text{(+ total } x))) \\
& \quad \quad \text{total})) \\
& \quad (\text{loop } 1 \ 0)) \\
\end{align*}
\]

\[
\begin{align*}
x & = 1 \\
total & = 0 \\
\textbf{while} & \quad x \times x < 50: \\
& \quad \quad \text{total} = \text{total} + x \\
x & = x + 1
\end{align*}
\]
Generating while loops

Could a procedure generate custom loop expressions for us?

```
(define (sum-while initial-x condition add-to-total update-x)
)
```

The goal is for this call:

```
(sum-while 1 '(< (* x x) 50) 'x '(+ x 1))
```

...to generate this expression:

```
(begin (define (loop x total)
    (if (< (* x x) 50)
        (loop (+ x 1) (+ total x))
        total))
    (loop 1 0))
```
Generating while loops (Solution)

```
(define (sum-while initial-x condition add-to-total update-x)
  `(begin (define (loop x total)
            (if ,condition
                (loop ,update-x (+ total ,add-to-total ))
                total))
              (loop ,initial-x 0))
)
```

```
(sum-while 1 '(< (* x x) 50) 'x '(+ x 1))
; (begin (define (loop x total) (if (< (* x x) 50) (loop (+ x 1) (

(eval (sum-while 1 '(< (* x x) 50) 'x '(+ x 1))) ; 28

(eval (sum-while 2 '(< x 10) '(* x x) '(+ x 2))) ; 120
```
Apply
The apply procedure

The `apply` procedure applies a given procedure to a list of arguments.

```
(apply  <procedure>  <arguments>
```

Examples:

```
(apply  +  '(1 2 3 ))
```

```
(define  (sum  s)  (apply  +  s))
```

```
(sum  '(1 2 3))
```
Combining eval and apply

A function that can apply any function expression to any list of arguments:

```
(define (call-func func-expression func-args)
  (apply (eval func-expression) func-args)
)

(call-func '(lambda (a b) (+ a b)) '(3 4))
```
Combining eval and apply

A function that can apply any function expression to any list of arguments:

```scheme
(define (call-func func-expression func-args)
  (apply (eval func-expression) func-args))

(call-func '(lambda (a b) (+ a b)) '(3 4)) ; 7
```