Scheme
Class outline:

- Scheme expressions
- Special forms
- Quotation
Scheme
A brief history of programming languages

The Lisp programming language was introduced in 1958. The Scheme dialect of Lisp was introduced in the 1970s, and is still maintained by a standards committee today.

Genealogical tree of programming languages

Scheme itself is not commonly used in production, but has influenced many other languages, and is a good example of a functional programming language.
Scheme expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2 3.3 #t #f + quotient
- Combinations: (quotient 10 2) (not #t)

Numbers are self-evaluating; symbols are bound to values.

Call expressions include an operator and 0 or more operands in parentheses:

```scheme
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
      (+ (* 2 4)
          (+ 3 5)))
     (+ (- 10 7)
       6))
```
Special forms
Special forms

A combination that is not a call expression is a special form:

- if expression:
  \[(\text{if } \langle \text{predicate} \rangle \ \langle \text{consequent} \rangle \ \langle \text{alternative} \rangle)\]

- and/or:
  \[(\text{and } \langle e1 \rangle \ \ldots \ \langle en \rangle)\]
  \[(\text{or } \langle e1 \rangle \ \ldots \ \langle en \rangle)\]

- Binding symbols:
  \[(\text{define } \langle \text{symbol} \rangle \ \langle \text{expression} \rangle)\]

- New procedures:
  \[(\text{define } (\langle \text{symbol} \rangle \ \langle \text{formal parameters} \rangle) \ \langle \text{body} \rangle)\]

Scheme spec: special forms
define form

define <name> <expression>

Evaluates <expression> and binds the value to <name> in the current environment. <name> must be a valid Scheme symbol.

(define x 2)

Scheme Spec: define
define procedure

\[
\text{define } (<\text{name}> \ [\text{param}] \ldots) \ <\text{body}>)
\]

Constructs a new procedure with parameters as its parameters and the body expressions as its body and binds it to name in the current environment. name must be a valid Scheme symbol. Each param must be a unique valid Scheme symbol.

\[
\text{(define (double x) (* 2 x))}
\]

Scheme Spec: define
If expression

if <predicate> <consequent> <alternative>

Evaluates predicate. If true, the consequent is evaluated and returned. Otherwise, the alternative, if it exists, is evaluated and returned (if no alternative is present in this case, the return value is undefined).

Example: This code returns the length of non-empty lists and 0 for empty lists:

```
(define nums '(1 2 3))
(if (null? nums) 0 (length nums))
```

Scheme Spec: If
and form

(and [test] ...)

Evaluate the test s in order, returning the first false value. If no test is false, return the last test. If no arguments are provided, return #t.

Example: This and form evaluates to true whenever x is both greater than 10 and less than 20.

(define x 15)
(and (> x 10) (< x 20))

Scheme Spec: And
or form

(or [test] ...)

Evaluate the test s in order, returning the first true value. If no test is true and there are no more test s left, return #f.

Example: This or form evaluates to true when either x is less than -10 or greater than 10.

(define x -15)
(or (< x -10) (> x 10))

Scheme Spec: Or
lambda expressions

Lambda expressions evaluate to anonymous procedures.

\[
\text{(lambda ([param] ...)} \ <\text{body}> \ ...)\]

Two equivalent expressions:

\[
\text{(define (plus4 x) (+ x 4))}
\]
\[
\text{(define plus4 (lambda (x) (+ x 4)))}
\]

An operator can be a call expression too:

\[
\text{((lambda (x y z) (+ x y (square z))) 1 2 3)}
\]

Scheme Spec: Lambda
The cond special form that behaves similar to if expressions in Python.

```scheme
(if x > 10:
    (print 'big'))
(elif x > 5:
    (print 'medium'))
(else:
    (print 'small'))
```

Scheme Spec: Cond
The begin form

```python
if x > 10:
    print('big')
    print('pie')
else:
    print('small')
    print('fry')
```

```scheme
(cond (> x 10) (begin (print 'big) (print 'pie)))
 (else (begin (print 'small) (print 'fry))))
```

Scheme Spec: Begin
The begin form

```python
if x > 10:
    print('big')
    print('pie')
else:
    print('small')
    print('fry')
```

```scheme
(cond ((> x 10) (begin (print 'big) (print 'pie)))
    (else (begin (print 'small) (print 'fry)))))
```

```scheme
(if (> x 10) (begin
    (print 'big)
    (print 'guy))
    (begin
    (print 'small)
    (print 'fry)))
```

**Scheme Spec: Begin**
let form

The `let` special form binds symbols to values temporarily; just for one expression

```scheme
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
```

↑ a and b are still bound down here

```scheme
(define c (let ((a 3)
               (b (+ 2 2)))
           (sqrt (+ (* a a) (* b b)))))
```

↑ a and b are not bound down here

Scheme Spec: Let
Scheme lists
Constructing a list

Scheme lists are linked lists.

Python (with our Link class:)

1 <-> 2 (null)
Constructing a list

Scheme lists are linked lists.

```
1  ->  2  ()
```

Python (with our Link class):

```
Link(1, Link(2))
```
Constructing a list

Scheme lists are linked lists.

Python (with our Link class:)

```
Link(1, Link(2))
```

Scheme (with the cons form:)

```
(cons 1 (cons 2 nil))
```

nil is the empty list.

Lists are written in parentheses with space-separated elements:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil)))) ; (1 2 3 4)
```
Accessing list elements

Python access:
Accessing list elements

Python access:

```python
lst = Link(1, Link(2))
lst.first  # 1
lst.rest   # Link(2)
```
Accessing list elements

Python access:

```python
lst = Link(1, Link(2))
lst.first  # 1
lst.rest   # Link(2)
```

Scheme access:

```scheme
(define lst (cons 1 (cons 2 nil)))
(car lst) ; 1
(cdr lst) ; (2)
```

- **car**: Procedure that returns the first element of a list
- **cdr**: Procedure that returns the rest of the list

Remember: "cdr" = "Cee Da Rest"
The list procedure

The built-in `list` procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
(list 1 2 3) ; (1 2 3)
(list 1 (list 2 3) 4)
(list (cons 1 (cons 2 nil)) 3 4)
```

Procedure reference: list
The list procedure

The built-in `list` procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
(list 1 2 3)                     ; (1 2 3)
(list 1 (list 2 3) 4)            ; (1 (2 3) 4)
(list (cons 1 (cons 2 nil)) 3 4)```

Procedure reference: list
The list procedure

The built-in `list` procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
(list 1 2 3) ; (1 2 3)
(list 1 (list 2 3) 4) ; (1 (2 3) 4)
(list (cons 1 (cons 2 nil)) 3 4) ; ((1 2) 3 4)
```

Procedure reference: list
Symbolic programming
Referring to symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b)
(list 'a b)
```

The `quote` is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Referring to symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b)
(list 'a b)
```

The `quote` is shorthand for the `quote` form:

```
(list (quote a) (quote b))
```
Referring to symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b) ; (a b)
(list 'a b)
```

The ' is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Referring to symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b) ; (a b)
(list 'a b) ; (a 2)
```

The `quote` is shorthand for the **quote** form:

```
(list (quote a) (quote b))
```
Referring to symbols

Symbols typically refer to values:

```
(define a 1)
(define b 2)
(list a b) ; (1 2)
```

**Quotation** is used to refer to symbols directly:

```
(list 'a 'b) ; (a b)
(list 'a b) ; (a 2)
```

The ' is shorthand for the **quote** form:

```
(list (quote a) (quote b)) ; (a b)
```
Quoting lists

Quotation can also be applied to combinations to form lists.

'(a b c) ; (a b c)
(car '(a b c))
(cdr '(a b c))
Quoting lists

Quotation can also be applied to combinations to form lists.

'(a b c) ; (a b c)
(car ' (a b c)) ; a
(cdr ' (a b c))
Quoting lists

Quotation can also be applied to combinations to form lists.

'(a b c) ; (a b c)
(car (a b c)) ; a
(cdr (a b c)) ; (b c)
Scheme tips

- Use the references!
- Auto-format your code!
- Constrain your brain: you're now living in a world of applicative programming. Look, ma, no mutation!