**Evaluation rule for call expressions:**
1. Evaluate the operator and operand subexpressions.
2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

**Applying user-defined functions:**
1. Create a new local frame with the same parent as the function that was applied.
2. Bind the arguments to the function’s formal parameter names in that frame.
3. Execute the body of the function in the environment beginning at that frame.

**Execution rule for def statements:**
1. Create a new function value with the specified name, formal parameters, and function body.
2. Its parent is the first frame of the current environment.
3. Bind the name of the function to the function value in the first frame of the current environment.

**Execution rule for assignment statements:**
1. Evaluate the expression(s) on the right of the equal sign.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.

**Execution rule for conditional statements:**
1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

**Evaluation rule for or expressions:**
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

**Evaluation rule for and expressions:**
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

**Evaluation rule for not expressions:**
1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

**Function of a single argument (not called term)**

def cube(k):
    return k * k * k

A formal parameter that will be bound to a function argument value.

**Function that takes a function value as an argument**

def summation(n, term):
    """Sum the first n terms of a sequence."
    total = 0
    k = 0
    while k < n:
        total += term(k)
        k += 1
    return total

The cube function is passed as an argument value.

**Nested def statements**

A call expression and the body of the function being called are evaluated in different environments.

**Error**

"y" is not found

**Higher-order function:**

A function that takes a function value as an argument value or returns a function as a return value.

**Non-Pure Functions**

- `abs(number):`
  - 2
- `pow(x, y):`
  - 1024

**Pure Functions**

- `display(...):` None
The function has a parent, and the parent of a function is the frame in which it was defined. Every user-defined function has a parent frame, which is defined as the frame in which they were defined.

The parent of a function is the environment in which it was defined. Every local frame has a parent (often the global frame). The parent of a function is the environment in which it was defined.

When a function is called:
1. A local frame is created, titled with the name of the function being called.
2. The parent of the function is copied to the new frame.
3. The body of the function is executed in the new frame.
4. The return value of the function is evaluated.

When a function is defined:
1. The function is bound to the name in the current environment.
2. The function's signature is created with formal parameters.
3. The function's body is executed, creating a local frame.
4. The function value is a function with the same domain, range, and behavior.

Return value of `make_adder` is an adder:

```python
def make_adder(n):
    return lambda k: k + n
    
>>> add_three = make_adder(3)
```

Calling the function at the call site:

```python
>>> add_three(4)
7
```

The parent of a function is the frame in which it was defined.

Every user-defined function has a parent frame. Every local frame has a parent (often the global frame). The parent of a frame is the environment in which it was defined.

The parent of a frame is the environment in which it was defined. The parent of a frame is the frame in which it was defined.

When a function is defined:
1. Create a function value: `func = <name>(<formal parameters>)`
2. Its parent is the current frame.
3. Bind `<name>` to the function value in the current frame (which is the first frame of the current environment).
4. Execute the body of the function in the environment that starts with `<name>`.

When a function is called:
1. Add a local frame, titled with the `<name>` of the function being called.
2. Copy the parent of the function to the local frame: `<parent=<<label>>`.
3. Bind the `<formal parameters>` to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

Square = lambda x, y: x * y

Evaluate to a function. No "return" keyword!

```python
square = lambda x, y: x * y
```

Both create a local frame with formal parameters `x` and `y` that returns the value of `x * y`. Must be a single expression.

```
def square(x):
    return x * x
```

Both functions have as their parent the environment in which they were defined.

Both bind that function to the name `square`.

Both functions have as their parent the environment in which they were defined.

When a function is called:
1. Add a local frame, titled with the `<name>` of the function being called.
2. Copy the parent of the function to the local frame: `<parent=<<label>>`.
3. Bind the `<formal parameters>` to the arguments in the local frame.
4. Execute the body of the function in the environment that starts with the local frame.

```
def search(f):
    return x
```

Valid function. No "return" keyword!

```python
def search(is_three):
    return x
```

Both create a local frame with formal parameters `x` and `y` that returns the value of `x * y`. Must be a single expression.

```
def search(f):
    return x
```

Return value of `make_adder` is a function:

```python
m = curry2(add)
```

Two return values, separated by commas.

```
add, mul
```

Return value of `make_adder` is an argument to `compose1`.

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```

Return value of `compose1` is an adder:

```
def compose1(f, g):
    return lambda x: f(g(x))
```