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 header’s expression.
2. Simultaneously bind the names on the left to those values, in the first frame of the current environment.
3. Evaluate the expression(s) on the right of the equal sign.

Evaluation rule for while 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 not expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true value, then the expression evaluates to False.
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, then the expression evaluates to False.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:
1. Evaluate the subexpression <left>.
2. If the result is a false value, then the expression evaluates to False, and False otherwise.

Execution rule for while statements:
1. Evaluate the header’s expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.
A function

\[
\text{square} = \lambda x,y: x \times y
\]

Evaluates to a function. No "return" keyword!

\[
\text{square} = \lambda x: x \times x
\]

A function with formal parameters \( x \) and \( y \) that returns the value of "\( x \times y \)"

\[
\text{square} = \lambda x: x \times x
\]

Must be a single expression

---

The parent of a function

- Every user-defined function has a parent frame.
- The parent of a function is the frame in which it was defined.
- Every local function has a parent frame.
- The parent of a local function is the parent of the function called.

---

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

\[
\text{def} \quad \text{curry2}(f): \\
\quad \text{"""} \text{Returns a function } g \text{ such that } g(x)(y) \text{ returns } f(x, y)."""
\]

\[
\text{def} \ g(x): \quad \text{return } h
\]

\[
\text{def} \ h(y): \quad \text{return } f(x, y)
\]

---

Anatomy of a recursive function:
- The def statement header is similar to other functions.
- Conditional statements check for base cases.
- Base cases are evaluated without recursive calls.
- Recursive cases are evaluated with recursive calls.

---

Recursive decomposition:

\[
\text{def} \quad \text{count_partitions}(n, m):
\quad \text{if } n = 0:
\quad \quad \text{return } 1
\]

E.g., \( \text{count_partitions}(6, 4) \)

Explore two possibilities:
- Use at least one 4
- Don't use any 4

Solve two simpler problems:
- \( \text{count_partitions}(2, 4) \)
- \( \text{count_partitions}(6, 3) \)

Tree recursion often involves exploring different choices.

---

\[
\text{square} = \lambda x: x \times x
\]

\[
\text{def} \quad \text{square}(x):
\quad \text{return } x \times x
\]

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

---

When a function is defined:
1. Create a function value: \( \text{func } <\text{name}> <\text{formal parameters}> \)
2. Its parent is the current frame.
3. Bind \(<\text{name}>\) 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 the local frame.

---

Program output:

```
12
123
```

\[
\text{def} \quad \text{inverse_cascade}(n):
\quad \text{grow}(n)
\quad \text{print}(n)
\quad \text{shrink}(n)
\quad \text{fib}(n): \quad \text{if } m:
\quad \quad \text{print}(n)
\quad \quad \text{fib}(n)
\quad \quad \text{g}(n)
\quad \quad \text{if } n:
\quad \quad \quad \text{return } 0
\quad \quad \text{elif } n = 1:
\quad \quad \quad \text{return } 1
\quad \quad \text{else:}
\quad \quad \quad \text{return } \text{fib}(n-2) + \text{fib}(n-1)
\]

\[
\text{def} \quad \text{fib}(n)
\quad \text{return } \text{fib}(n-2) + \text{fib}(n-1)
\]

\[
\text{def} \quad \text{Fact}(n)
\quad \text{if } n = 0:
\quad \quad \text{return } 1
\quad \quad \text{else:}
\quad \quad \quad \text{return } n * \text{Fact}(n-1)
\quad \text{fact}(3)
\]

---

Multiple assignment:

```
\[
\text{from} \ \text{operator} \ \text{import} \ \text{floordiv}, \ \text{mod}
\quad \text{def} \ \text{def} \text{divide_exact}(n, d):
\quad \text{"""} \text{Return the quotient and remainder of dividing N by D.}
\quad \text{"""
\quad \text{if } n = 0:
\quad \quad \text{return } 0
\quad \quad \text{elif } n < 0:
\quad \quad \quad \text{return } 0
\quad \quad \text{elif } n = 0:
\quad \quad \quad \text{return } 0
\quad \quad \text{else:}
\quad \quad \quad \text{with } m = \text{count_partitions}(n, m)
\quad \quad \quad \text{without } m = \text{count_partitions}(n, m-1)
\quad \quad \quad \text{return } m + \text{without } m
\]
\[
\text{print}(\text{floordiv}(d, m), \text{mod}(n, d))
\]
```

---

Multiple return values, separated by commas.