Evaluation rule for call expressions:
1. Evaluate the operand 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:
Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true 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 <exp>; The value is True if the result is a false value, 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.

Execution rule for if statements:
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.
3. Otherwise, the expression evaluates to the value of the first remaining clause.

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.

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 first remaining clause.

Evaluation rule for or expressions:
1. Evaluate the subexpression <left>.
2. If the result is a true 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 <exp>; The value is True if the result is a false value, 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.
3. Otherwise, the expression evaluates to the value of the first remaining clause.
CS 61A Midterm 1 Study Guide – Page 2

Facts about print
- Non-pure function
- Multiple arguments are printed with a space between them

>>> print(1, 2)
1 2

• Must be a single expression

A function is recursive if the body calls the function itself, either directly or indirectly. Recursive functions have two important components:
1. Base case(s), where the function directly computes an answer without calling itself
2. Recursive case(s), where the function calls itself as part of the computation

1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: x - f(x) / f'(x)

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess."
    return newton_update(f, guess)

def newton_update(f):
    """Return an update function for f using Newton's method."
    return lambda x, delta:
        x - f(x) / approx_derivative(f, x)

def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."
    return f(x + delta) - f(x) / delta

def find_zero(f, guess):
    """Find a zero of the function f, near guess."
    return newton_update(f, guess)

# Example usage
f = lambda x: x**2 - 2
print(find_zero(f, 1))
1.4142135623730951