Final Examples
Class outline:

- Trees
- Recursive accumulation
- Regular expressions
- Interpreters
Trees
Tree abstractions

Using functions:

```python
def tree(label, branches=[]):
    return [label] + list(branches)
def label(t):
    return t[0]
def branches(t):
    return t[1:]
def is_leaf(t):
    return not branches(t)
```

Using a class:

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)
    
def is_leaf(self):
        return not self.branches
```
Tree-structured data

A tree is a recursive structure, where each branch may itself be a tree.

[5, [6, 7], 8, [[9], 10]]

(+ 5 (- 6 7) 8 (* (- 9) 10))

(S
   (NP (JJ Short) (NNS cuts))
   (VP (VBP make)
      (NP (JJ long) (NNS delays)))
   (.
   .))

<ul>
   <li>Midterm <strong>1</strong></li>
   <li>Midterm <strong>2</strong></li>
</ul>

Tree processing often involves recursive calls on subtrees.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

1. Understand the question and function signature.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    ""
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving tree problems

Implement \texttt{bigs}, which takes a \texttt{Tree} instance \texttt{t} containing integer labels. It returns the number of nodes in \texttt{t} whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
    """
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving tree problems

Implement `bigs`, which takes a `Tree` instance `t` containing integer labels. It returns the number of nodes in `t` whose labels are larger than all labels of their ancestor nodes.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
4. Consider what you expect to see in the solution.
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""

4. Consider what you expect to see in the solution.

Typical tree processing structure?

```python
if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])
```
Solving bigs #2

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""
```

4. Consider what you expect to see in the solution.

Typical tree processing structure?

```python
if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])
```

✗ That won't work, since we need to know about ancestors.
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""
Solving bigs #3

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
    """
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

4. Consider what you expect to see in the solution.

Some code that increments the total count

```
1 + _____
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
    """

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """

4. Consider what you expect to see in the solution.

Some code that increments the total count

1 + _____

Some way of tracking ancestor labels or max of ancestors seen so far.

if node.label > max(ancestors):

if node.label > max_ancestor:
Solving bigs #4

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."
    
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

5. Check out the provided template.

```python
def f(a, x):
    if ________________:
        return 1 + __________
    else:
        return __________
return ________________
```
Solving bigs #4

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
    ""

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```python
def f(a, x):
    if ________________:
        return 1 + __________
    else:
        return _____________
    return _____________________
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

```python
def f(a, x):
    if ________________:  # Track the largest ancestor
        return 1 + ________  # Increment total
    else:
        return ____________
return ___________________
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

def f(a, x):
    if _________________:  # Track the largest ancestor
        return 1 + __________  # Increment total
    else:
        return _______________

    return _________________
Solving bigs #4

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

>>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
>>> bigs(a)
4
"""
```

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

```python
# a is the current subtree, x is the largest ancestor
def f(a, x):
    if _____________________:  # Track the largest ancestor
        return 1 + __________  # Increment total
    else:
        return __________
return ________________
```
Solving bigs #5

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```

8. Finish filling in the skeleton.

```python
def f(a, x):
    if a.label > x:
        return 1 + sum([f(b, a.label) for b in a.branches])
    else:
        return sum([f(b, x) for b in a.branches])
def f(t, t.label - 1)
```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """

def f(a, x):
    if a.label > x:
        return 1 + sum([f(b, a.label) for b in a.branches])
    else:
        return sum([f(b, x) for b in a.branches])

return f(t, t.label - 1)
Recursive accumulation
Alternative bigs approach

Initialize some data structure to an empty/zero value, and populate it as you go.

```python
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = [0]
    def f(a, x):
        if ________________________:
            ________________________
            ________________________:
            f(_______________________)
            ________________________
    return n[0]
```
Alternative bigs solution

Initialize some data structure to an empty/zero value, and populate it as you go.

def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = [0]
    def f(a, x):
        if a.label > x:
            n[0] += 1
        for b in a.branches:
            f(b, max(a.label, x))
    f(t, t.label - 1)
    return n[0]
Solving smalls

Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."
    
    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
```

1. Understand the question and function signature.
Solving smalls

Implement smalls, which takes a Tree instance \( t \) containing integer labels. It returns the non-leaf nodes in \( t \) whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in \( t \) that are smaller than all their descendants."

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
"""
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
Solving smalls

Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
**Solving smalls**

Implement `small` function which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving smalls

Implement `smalls`, which takes a `Tree` instance `t` containing integer labels. It returns the non-leaf nodes in `t` whose labels are smaller than any labels of their descendant nodes.

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
"""
```

1. Understand the question and function signature.
2. Make any diagrams that may be helpful.
3. Work through the examples and make observations.
Solving smalls #2

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."
```

```python
>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
"""
```

4. Consider what you expect to see in the solution.
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    """

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

min(____)
Solving smalls #2

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    """
    return [t.label for t in smalls(t) if t.label < smallest]

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])

>>> sorted([t.label for t in smalls(a)])

[0, 2]
```

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

```python
min(__)
```

Something which compares smallest to current

```python
t.label < smallest
```
Solving smalls #2

```python
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    """

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
```

4. Consider what you expect to see in the solution.

Something which finds the smallest value in a subtree

```
min(__)
```

Something which compares smallest to current

```
t.label < smallest
```

Something which adds a subtree to a list

```
__.append(t)
```
def smalls(t):
    
    
    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    
5. Check out the provided template.

result = []
def process(t):
    if t.is_leaf():
        return ______________________
    else:
        smallest = ______________________
        if ______________________:
            ______________________
            return min(smallest, t.label)
        process(t)
    return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

result = []
def process(t):
    if t.is_leaf():
        return ____________________________
    else:
        smallest = ____________________________
        if ____________________________:
            ____________________________
            return min(smallest, t.label)
        process(t)
    return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."""

    a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

result = []
def process(t):
    if t.is_leaf():
        return ________________
    else:
        smallest = ________________  # Finds smallest
        if ________________:
            ________________
            return min(smallest, t.label)
        process(t)
    return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

result = []
def process(t):
    if t.is_leaf():
        return ____________________
    else:
        smallest = ____________________ # Finds smallest
        if ________________________: # Compares smallest
            ________________________
            return min(smallest, t.label)

    process(t)
    return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

>>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
>>> sorted([t.label for t in smalls(a)])
[0, 2]
""

5. Check out the provided template.
6. Figure out where what you expected fits into the template.

result = [] # The result list
def process(t):
    if t.is_leaf():
        return ______________________
    else:
        smallest = ____________________ # Finds smallest
        if ___________________________:
            ___________________________
        return min(smallest, t.label)
process(t)
return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

5. Check out the provided template.
6. Figure out where what you expected fits into the template.
7. Label any ambiguously named variables if its helpful.

result = [] # The result list
def process(t):
    if t.is_leaf():
        return _________________
    else:
        smallest = _________________ # Finds smallest
        if _________________:
            _________________ # Compares smallest
        _________________ # Appends subtree to list
        return min(smallest, t.label)
    process(t)
return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    """

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)]]))])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

result = []  # The result list
def process(t):  # t is a Tree
    if t.is_leaf():
        return ______________________
    else:
        smallest = _________________  # Finds smallest
        if __________________________:  # Compares smallest
            __________________________  # Appends subtree to list
            return min(smallest, t.label)
    process(t)
return result
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants."

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """

8. Finish filling in the skeleton.

```python
def process(t):
    if t.is_leaf():
        return t.label
    else:
        smallest = min([process(b) for b in t.branches])
        if t.label < smallest:
            result.append(t)
            return min(smallest, t.label)

process(t)
return result
```

def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.
    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):
        if t.is_leaf():
            return t.label
        else:
            smallest = min([process(b) for b in t.branches])
            if t.label < smallest:
                result.append(t)
            return min(smallest, t.label)
    process(t)
    return result

8. Check your work!
Regular expressions
# Matching patterns

Which strings are matched by each regular expression?

<table>
<thead>
<tr>
<th>Expressions:</th>
<th>abc</th>
<th>cab</th>
<th>bac</th>
<th>baba</th>
<th>ababca</th>
<th>aabcc</th>
<th>abba</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a<em>b</em>c*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ab</td>
<td>[bc]*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a[bc]+)+a?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ab</td>
<td>ba)+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ab</td>
<td>[bc])?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Matching patterns

Which strings are matched by each regular expression?

<table>
<thead>
<tr>
<th>Expressions:</th>
<th>abc</th>
<th>cab</th>
<th>bac</th>
<th>baba</th>
<th>ababca</th>
<th>aabcc</th>
<th>abba</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]*</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>a<em>b</em>c*</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>❌</td>
</tr>
<tr>
<td>ab</td>
<td>[bc]/*</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>(a[bc]+)+a?</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
</tr>
<tr>
<td>(ab</td>
<td>ba)+</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>(ab</td>
<td>[bc])?</td>
<td>✔</td>
<td>❌</td>
<td>✔</td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>
Interpreters
Interpreter analysis

What expressions are passed to \texttt{scheme_eval} when evaluating the following expressions?

\begin{verbatim}
(define x (+ 1 2))

(define (f y) (+ x y))

(f (if (> 3 2) 4 5))
\end{verbatim}
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

\[
\text{(define } x \text{ ( + 1 2 ))}
\]

\[
\text{(define (f y) ( + x y ))}
\]

\[
\text{(f (if (> 3 2) 4 5))}
\]
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))

(define (f y) (+ x y))

(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to \texttt{scheme\_eval} when evaluating the following expressions?

\begin{verbatim}
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
\end{verbatim}
Interpreter analysis

What expressions are passed to \texttt{scheme_eval} when evaluating the following expressions?

\begin{verbatim}
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
\end{verbatim}
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x ((+ 1 2)))
(define (f y) ((+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpretation analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

1. `(define x (+ 1 2))`
2. `(define (f y) (+ x y))`
3. `(f (if (> 3 2) 4 5))`
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```scheme
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```
Interpreter analysis

What expressions are passed to `scheme_eval` when evaluating the following expressions?

\[(\text{define } x \ (\ + \ 1 \ 2))\]

\[(\text{define } (f \ y) \ (\ + \ x \ y))\]

\[(f \ (\text{if} \ (\ > \ 3 \ 2) \ 4 \ 5))\]
Interpreter analysis

What expressions are passed to \texttt{scheme\_eval} when evaluating the following expressions?

\begin{itemize}
  \item \texttt{(define x (+ 1 2))}
  \item \texttt{(define (f y) (+ x y))}
  \item \texttt{(f (if (> 3 2) 4 5))}
\end{itemize}
**Interpreter analysis**

What expressions are passed to `scheme_eval` when evaluating the following expressions?

```
(define x (+ 1 2))
(define (f y) (+ x y))
(f (if (> 3 2) 4 5))
```