Lecture 2: Functions

CS 61A - Summer 2024

Raymond Tan

$$20 + 24 \qquad 2^{100} \qquad \sin \pi$$

$$\lim_{x \to \infty} \frac{1}{x} \qquad f(x) \qquad \frac{20}{24} \qquad \sum_{i=1}^{n} i$$

$$\log x \qquad \sqrt{2024} \qquad \binom{n}{x} \qquad -2024$$

An **expression** describes a computation and evaluates to a value.

Types of Expressions

Primitive Expressions:

`2` `3.5` `'hello!'` `True`

numbers | strings | booleans

Call Expressions:

`add` (`2` , `3`)

Operator | Operand | Operand

Call Expressions

Evaluation procedure for **call expressions**

1. Evaluate the **operator**
2. Evaluate the **operands** from left to right
3. **Apply** the operator (a **function**) to the evaluated operands (**arguments**)

Operators and operands are also expressions

So they also *evaluate to values*

```
add(add(6, mul(4, 6)), mul(3, 5))
```

Operator

Operand

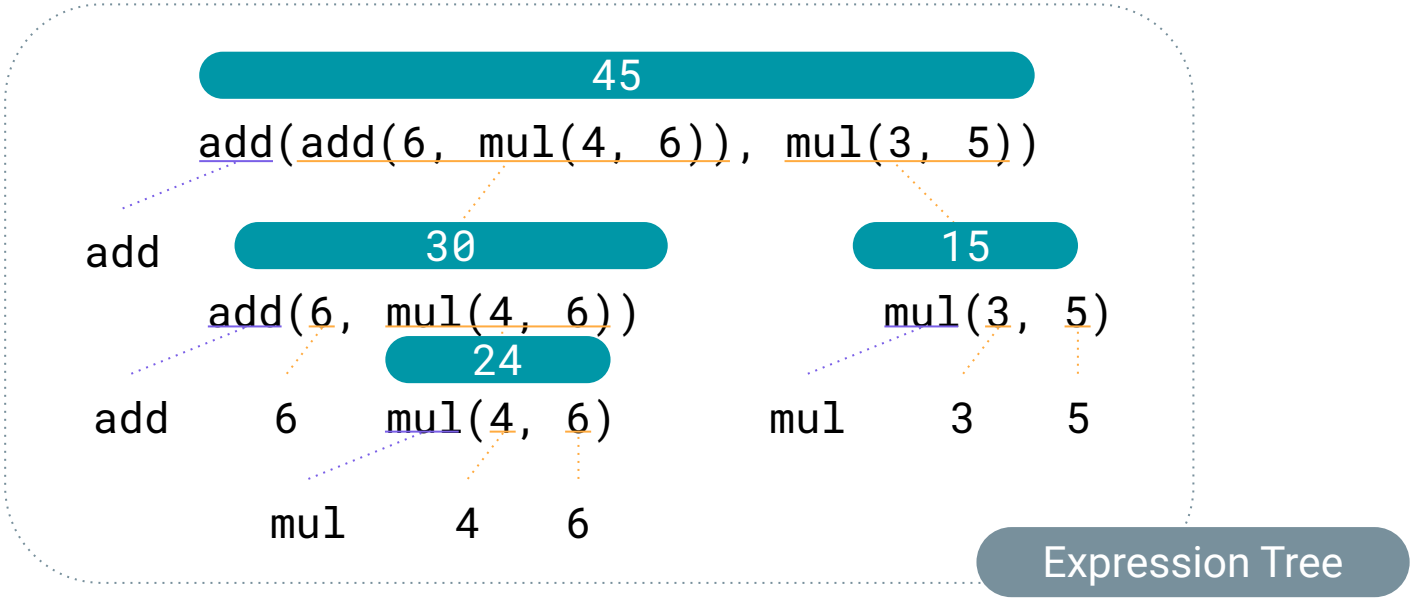
Operand

Nested Call Expressions

1 Evaluate operator

2 Evaluate operands

3 Apply!



Assignment Statements

In Python, we can bind expressions to variables.

Assignment Statements

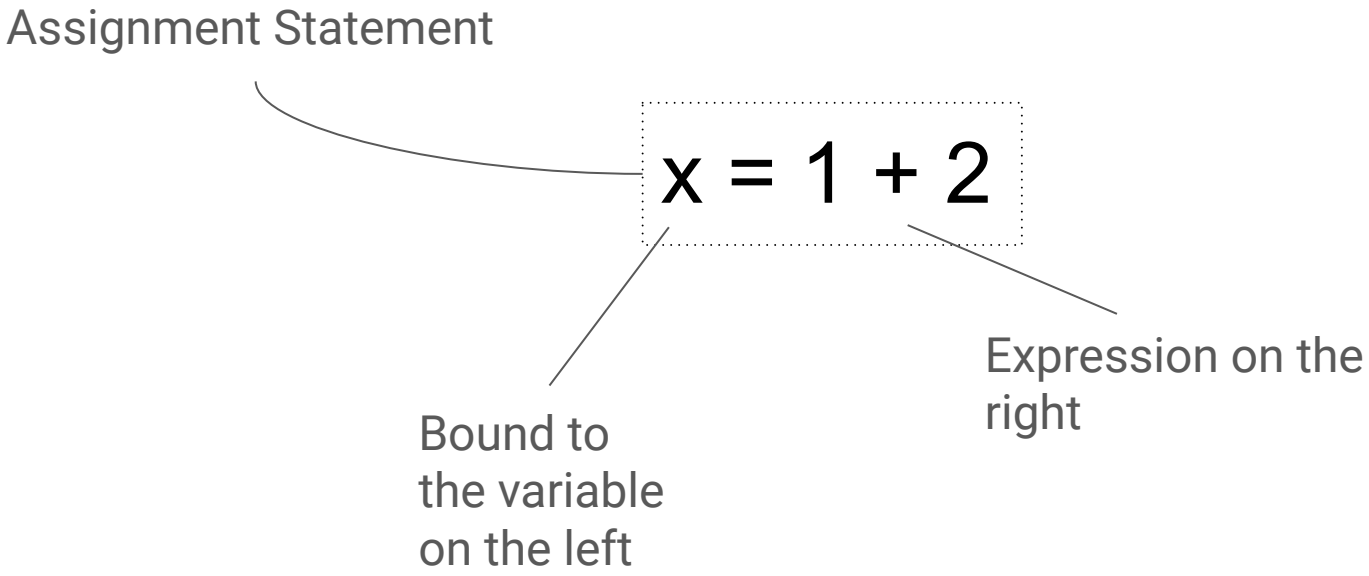
In Python, we can bind expressions to variables.

Assignment Statement

```
x = 1 + 2
```

Assignment Statements

In Python, we can bind expressions to variables.



Assignment Statements

In Python, we can bind expressions to variables.

Incorrect!

~~1 + 2 = x~~

Assignment Statement

x = 1 + 2

Bound to the variable on the left

Expression on the right

Assignment Statements – Example

```
>>> x = 2
>>> y = x + 1
>>> y
3
>>> x = 5
>>> y
3
```

Assignment Statements – Example

```
>>> x = 2
>>> y = x + 1
>>> y
3
>>> x = 5
>>> y
3
```

y still retains its original value after x has been reassigned!

Demo: Assignment of Python builtin functions

User Defined Functions

We can define our own functions using def statements!

User Defined Functions

We can define our own functions using def statements!

```
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)
```

User Defined Functions

We can define our own functions using def statements!

def tells Python we're defining a function

```
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)
```

User Defined Functions

We can define our own functions using def statements!

def tells Python we're defining a function

Doctests provide a description of our function and the expected behavior

```
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)
```

User Defined Functions

We can define our own functions using def statements!

def tells Python we're defining a function

Doctests provide a description of our function and the expected behavior

return statement denotes what the function is outputting

```
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)
```

User Defined Functions

We can define our own functions using def statements!

def tells Python we're defining a function

Doctests provide a description of our function and the expected behavior

return statement denotes what the function is outputting

```
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)
```

A function's ***signature*** is denoted as the name of the function and the arguments it takes in.

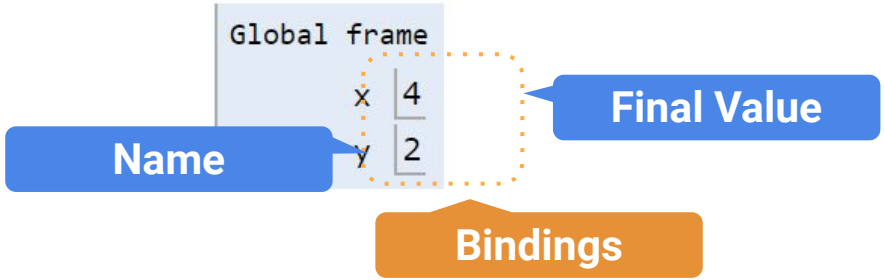
Environment Diagrams

Demo

Visualizing Assignment

Names are bound to **values** in an environment

```
1  x = 1
2  y = 2
3  x = y * 2
```

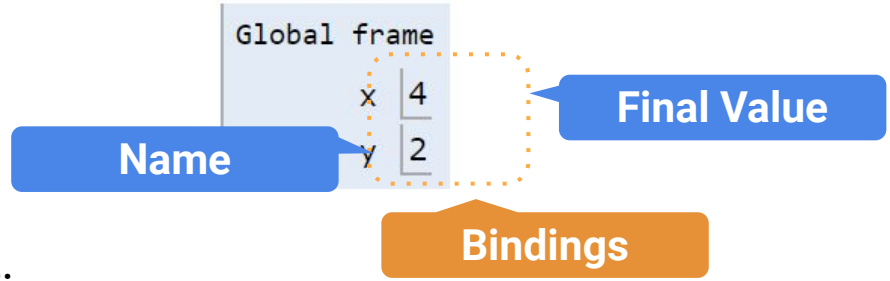


Demo

Visualizing Assignment

Names are bound to **values** in an environment

```
1  x = 1
2  y = 2
3  x = y * 2
```



To execute an assignment statement:

1. **Evaluate** the expression to the right of **=**.
2. **Bind** the value of the expression to the name to the left of **=** in the current environment.

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function’s formal parameters to its arguments in that frame

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

Write code in Python 3.6 and display frames of exited functions.

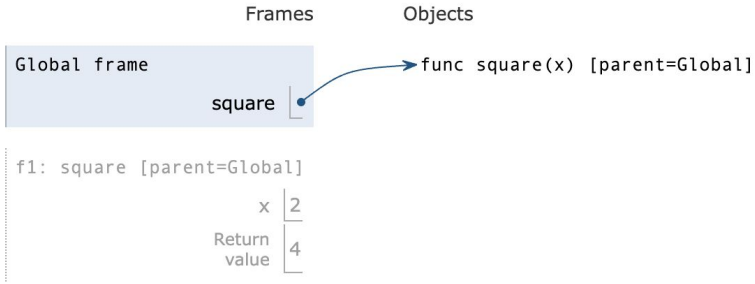
```python
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)

square(2)
```

tutor.cs61a.org

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment



Write code in Python 3.6 and display frames of exited functions.

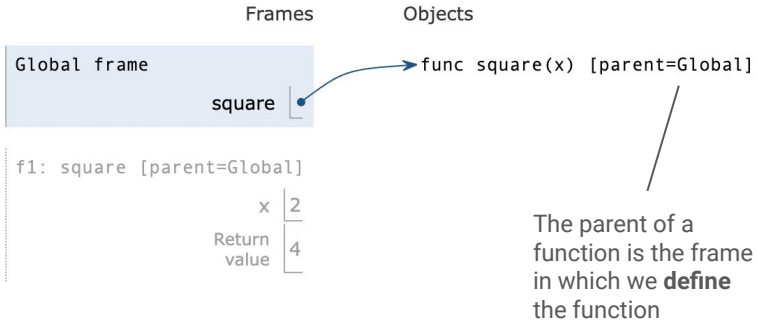
```python
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)

square(2)
```

tutor.cs61a.org

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment



Write code in Python 3.6 and display frames of exited functions.

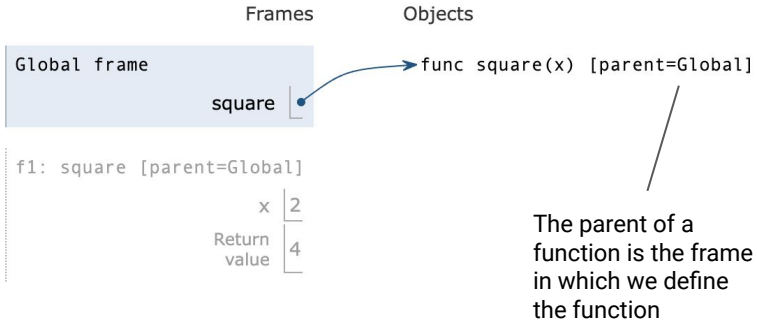
```python
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)

square(2)
```

tutor.cs61a.org

Calling User Defined Functions

Procedure for calling/applying user-defined functions:

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment



Write code in Python 3.6 and display frames of exited functions.

```python
def square(x):
    """
    Takes an input x and squares it.
    >>> square(3)
    9
    """
    return pow(x, 2)

square(2)
```

tutor.cs61a.org

*A function's signature has all the information needed to create a local frame

*Except argument values

Break time!

What if we have variables with the same name in different parts of our program?

```
x = 1
def my_func():
    x = 100
    print(x)

my_func() # What value gets displayed here?
```

What if we have variables with the same name in different parts of our program?

```
x = 1
def my_func():
    x = 100
    print(x)

my_func() # What value gets displayed here?
```

Answer: 100

Looking Up Names in Environments

Every expression is evaluated in the context of an environment.

Looking Up Names in Environments

Every expression is evaluated in the context of an environment.

Our programs always begin with a *global* frame, and then new *local* frames are created when we make a function call.

Looking Up Names in Environments

Every expression is evaluated in the context of an environment.

Our programs always begin with a *global* frame, and then new *local* frames are created when we make a function call.

Most important rules for environment diagrams:

Looking Up Names in Environments

Every expression is evaluated in the context of an environment.

Our programs always begin with a *global* frame, and then new *local* frames are created when we make a function call.

Most important rules for environment diagrams:

1. ***An environment is a sequence of frames.***

Looking Up Names in Environments

Every expression is evaluated in the context of an environment.

Our programs always begin with a *global* frame, and then new *local* frames are created when we make a function call.

Most important rules for environment diagrams:

1. ***An environment is a sequence of frames.***
2. ***A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found***

A sequence of frames

Latest

Global

f1

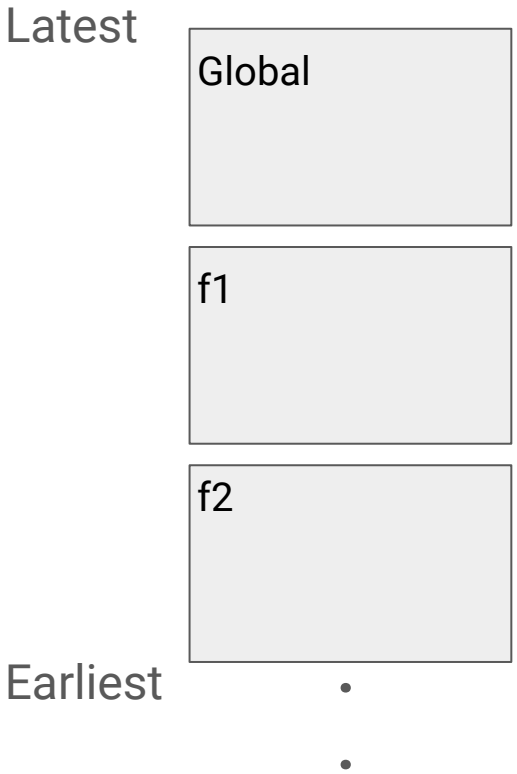
f2

Earliest

•

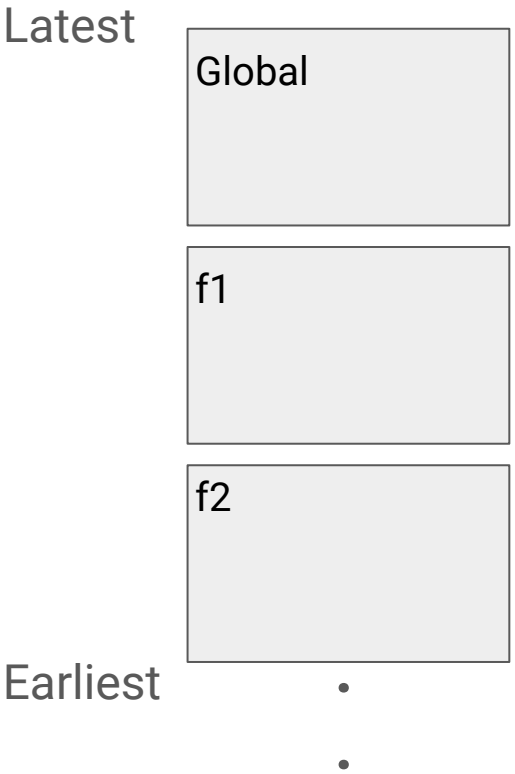
•

A sequence of frames



A sequence is a first frame, and then the rest of the sequence

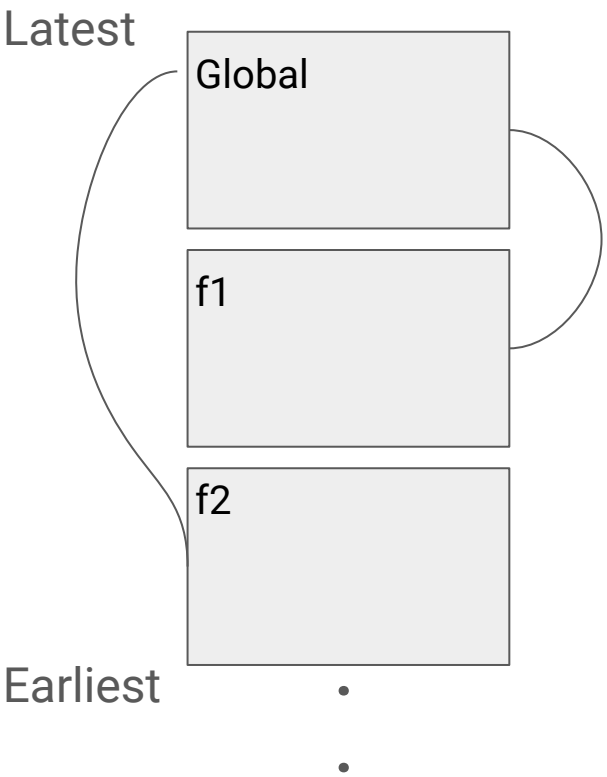
A sequence of frames



A sequence is a first frame, and then the rest of the sequence

Not every frame is part of the same environment, though each frame on the left is part of the **environment diagram**

A sequence of frames

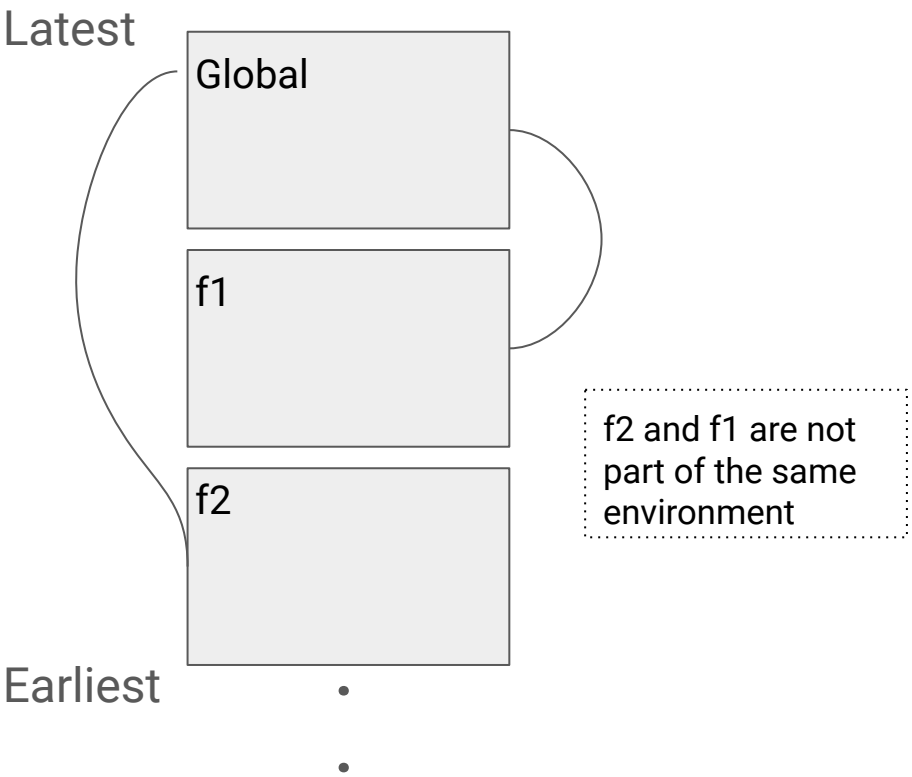


A sequence is a first frame, and then the rest of the sequence

Not every frame is part of the same environment, though each frame on the left is part of the **environment diagram**

f2 has a parent frame of Global, and f1 has a parent frame of Global

A sequence of frames



A sequence is a first frame, and then the rest of the sequence

Not every frame is part of the same environment, though each frame on the left is part of the **environment diagram**

f2 has a parent frame of Global, and f1 has a parent frame of Global

Example #1

```
x = 4
def f():
    x = 2
    return x + 1

def g():
    y = 2
    return x + y

f()
print(g()) # What value gets displayed here?
```

Example #1

```
x = 4
def f():
    x = 2
    return x + 1

def g():
    y = 2
    return x + y

f()
print(g()) # What value gets displayed here?
```

Answer: 6

Example #2

```
from operator import mul

def square(square):
    return mul(square, square)

print(square(3))
```

Reminder: Rules for evaluating call expressions

Evaluation procedure for **call expressions**

1. Evaluate the **operator**
2. Evaluate the **operands** from left to right
3. **Apply** the operator (a **function**) to the evaluated operands (**arguments**)

It is not until we finish the first two steps and get to the third step that a new frame is opened, and operand values are bound to the argument names!

Example #2

```
from operator import mul

def square(square):
    return mul(square, square)

print(square(3))
```

Answer: 9

Summary

- An **expression** is anything that evaluates to a value in Python
  - Primitive and call expressions
- **Assignment statements** bind names to values
- Call expression evaluation follows a distinct set of rules
  - Evaluate the operator, evaluate the operand, and apply the operator onto the operands
- **Environment diagrams** allow us to visualize assignment
  - Use tutor.cs61a.org to try this out on your own programs!
- Each environment is a sequence of frames, and all frames in a program make up an environment diagram