

In this tutorial, we will learn *more* about definitions.

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

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

Run 

error

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

`f(2) + y` is evaluated in the top-level block, where `y` is not defined. So, this program errors.

Click [here](#) to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
x = 1  
def f():  
    y = 2  
    def g():  
        return x + y  
    return g()  
print(f())
```

Run 

3

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

`f()` returns the value of `g()`. The value of `g()` is the value of `x + y`, which is `1 + 2`, which is 3.

Click [here](#) to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
x = 1  
def f(y):  
    x = 2
```

Run 

```

    ^      ^
    return x + y
print(f(0) + x)

```

4

Syntax: Lispy JS PY Scala 3

The answer is 3. `x = 2` binds `x` to 2. You might think the binding applies everywhere. However, it only applies to the body of `f`. The `x` in `f(0) + x` appears in the top-level block, so it refers to the `x` defined in the top-level block. In SMoL, variable references follow the hierarchical structure of blocks.

Click [here](#) to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```

cat = 7
def k(dog):
    cat = 4
    return cat + dog
print(k(1) + cat)

```

Run ▶

12

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```

x = 1
def f(y):
    def g():
        z = 2
        return x + y + z
    return g()
print(f(3) + 4)

```

Run ▶

10

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3


This program binds `x` to 1 and `f` to a function and then evaluates `f(3) + 4`. The value of `f(3) + 4` is the value of `g()` + 4, which is the value of `(x + y + z) + 4`, which is the value of `(1 + 3 + 2) + 4`, which is 10.

Click [here](#) to run this program in the Stacker.

What is the result of running this program?

Syntax: Lispy JS PY Scala 3

```
x = 1
def f():
    return x
def g():
    x = 2
    return f()
print(g())
```

Run 

1

Syntax: Lispy JS PY Scala 3

You got it right! 🎉🎉🎉

Syntax: Lispy JS PY Scala 3

`g()` evaluates to `f()`, which evaluates to `1` because `f` and `x` are both defined in the same block, and `f` does *not* get the `x` defined inside `g`.

Click [here](#) to run this program in the Stacker.

When we see a variable reference (e.g., the `x` in `x + 1`), how do we find its value, if any?

Syntax: Lispy JS PY Scala 3

what?

Syntax: Lispy JS PY Scala 3

Variable references follow the hierarchical structure of blocks.

Syntax: Lispy JS PY Scala 3

If the variable is defined in the current block, we use that declaration.

Otherwise, we look up the block in which the current block appears, and so on recursively. (Specifically, if the current block is a function body, the next block will be the block in which the function definition is; if the current block is the top-level block, the next block will be the **primordial block**.)

If the current block is already the primordial block and we still haven't found a corresponding declaration, the variable reference errors.

The primordial block is a non-visible block enclosing the top-level block. This block defines values and functions that are provided by the language itself.

Any feedback regarding these statements? Feel free to skip this question.

Syntax: Lispy JS PY Scala 3

(You skipped the question.)

Syntax: Lispy JS PY Scala 3

Now please scroll back and select 1-3 programs that make the point that

Syntax: Lispy JS PY Scala 3

■ If the variable is declared in the current block, we use that declaration.

You don't need to select *all* such programs.

(You selected 1 programs)

Syntax: Lispy JS PY Scala 3

Okay. How does this program (7) support the point?

Syntax: Lispy JS PY Scala 3

When calling on function $f()$ in $g()$, it will not know the value of x from g , but instead from the global scope, since $f()$ is not defined inside $g()$

Syntax: Lispy JS PY Scala 3

Please select 1-3 programs that make the point that

Syntax: Lispy JS PY Scala 3

■ Otherwise, we look up the block in which the current block appears, and so on recursively.

You don't need to select *all* such programs.

(You selected 2 programs)

Syntax: Lispy JS PY Scala 3

Okay. How do these programs (4,13) support the point?

Syntax: Lispy JS PY Scala 3

They look up hierarchically, as the variable does not exist in the current block

Syntax: Lispy JS PY Scala 3

Please select 1-3 programs that make the point that

Syntax: Lispy JS PY Scala 3

■ If the current block is already the primordial block and we still haven't found a corresponding declaration, the variable reference errors.

You don't need to select *all* such programs.

(You selected 1 programs)

Syntax: Lispy JS PY Scala 3

Okay. How does this program (1) support the point?

Syntax: Lispy JS PY Scala 3

y is not defined globally

Syntax: Lispy JS PY Scala 3

Syntax: Lispy JS PY Scala 3

The **scope** of a variable is the region of a program where we can refer to the variable.

Technically, it includes the block in which the variable is defined (including sub-blocks) except the sub-blocks where the same name is re-defined. When the exception happens, that is, when the same name is defined in a sub-block, we say that the variable in the sub-block **shadows** the variable in the outer block.

We say a variable reference (e.g., "the `x` in `x + 1`") is **in the scope of** a declaration (e.g., "the `x` in `x = 23`") if and only if the former refers to the latter.

Syntax: Lispy JS PY Scala 3

Any feedback regarding these statements? Feel free to skip this question.

Syntax: Lispy JS PY Scala 3

(You skipped the question.)

Syntax: Lispy JS PY Scala 3

Please select all statements that apply to the following program:

```
x = 45
def f():
    def g():
        return x + 1
    x = 67
    return x + 2
print(f())
print(x + 3)
```

Run 

Syntax: Lispy JS PY Scala 3

- ☐ The scope of the top-level `x` includes the whole program.
- ☒ The scope of the top-level `x` includes `x + 3`.
- ☐ The top-level `x` shadows the `x` defined in `f`.
- ☐ The `x` in `x + 1` is in the scope of the `x` defined in `f`.
- ☒ The `x` in `x + 2` is in the scope of the `x` defined in `f`.
- ☐ The `x` in `x + 3` is in the scope of the `x` defined in `f`.

Syntax: Lispy JS PY Scala 3

You should have chosen "The `x` in `x + 1` is in the scope of the `x` defined in `f`". The scope of the top-level `x` includes the whole program except the body of `f`. The scope of the `x` defined in `f` is the body of `f`.

The `x` defined in `f` shadows the top-level `x` rather than the other way around.

Syntax: Lispy JS PY Scala 3



Here is a program that confused many students

```
x = 1
def foo():
    return x
```

Run 

```
def bar():  
    x = 2  
    return foo()  
print(bar())
```

Please

1. Run this program in the stacker by clicking the green run button above;
2. The stacker would show how this program produces its result(s);
3. Keep clicking  until you reach a configuration that you find particularly helpful;
4. Click  to get a link to your configuration;
5. Submit your link below;

<https://smol-tutor.xyz/stacker/?syntax=Python&randomSeed=smol-tutor&nNext=5&program=%28defvar+x+1%29%0A%28deffun+%28foo%29%0A++x%29%0A%28deffun+%28bar%29%0A++%28defvar+x+2%29%0A++%28foo%29%29%0A%0A%28bar%29%0A&readOnlyMode=>

Syntax: Lispy JS PY Scala 3

Please write a couple of sentences to explain how your configuration explains the result(s) of the program.

Syntax: Lispy JS PY Scala 3

It shows the final result, you can figure everything out based on that.

Syntax: Lispy JS PY Scala 3

Let's review what we have learned in this tutorial.

Syntax: Lispy JS PY Scala 3

Variable references follow the hierarchical structure of blocks.

If the variable is defined in the current block, we use that declaration.

Otherwise, we look up the block in which the current block appears, and so on recursively. (Specifically, if the current block is a function body, the next block will be the block in which the function definition is; if the current block is the top-level block, the next block will be the **primordial block**.)

If the current block is already the primordial block and we still haven't found a corresponding declaration, the variable reference errors.

The primordial block is a non-visible block enclosing the top-level block. This block defines values and functions that are provided by the language itself.

The **scope** of a variable is the region of a program where we can refer to the variable. Technically, it includes the block in which the variable is defined (including sub-

blocks) except the sub-blocks where the same name is re-defined. When the exception happens, that is, when the same name is defined in a sub-block, we say that the variable in the sub-block **shadows** the variable in the outer block.

We say a variable reference (e.g., "the x in $x + 1$ ") is **in the scope of** a declaration (e.g., "the x in $x = 23$ ") if and only if the former refers to the latter.

You have finished this tutorial 🎉🎉🎉

Please the finished tutorial to a PDF file so you can review the content in the future. **Your instructor (if any) might require you to submit the PDF.**

Start time: 1711095634504