Scopes
Global and Block

Lecture 9

Go to poll.unc.edu

Sign-in via this website then go to pollev.com/compunc

VSCode: Open Project -> View Terminal -> npm run pull -> npm start

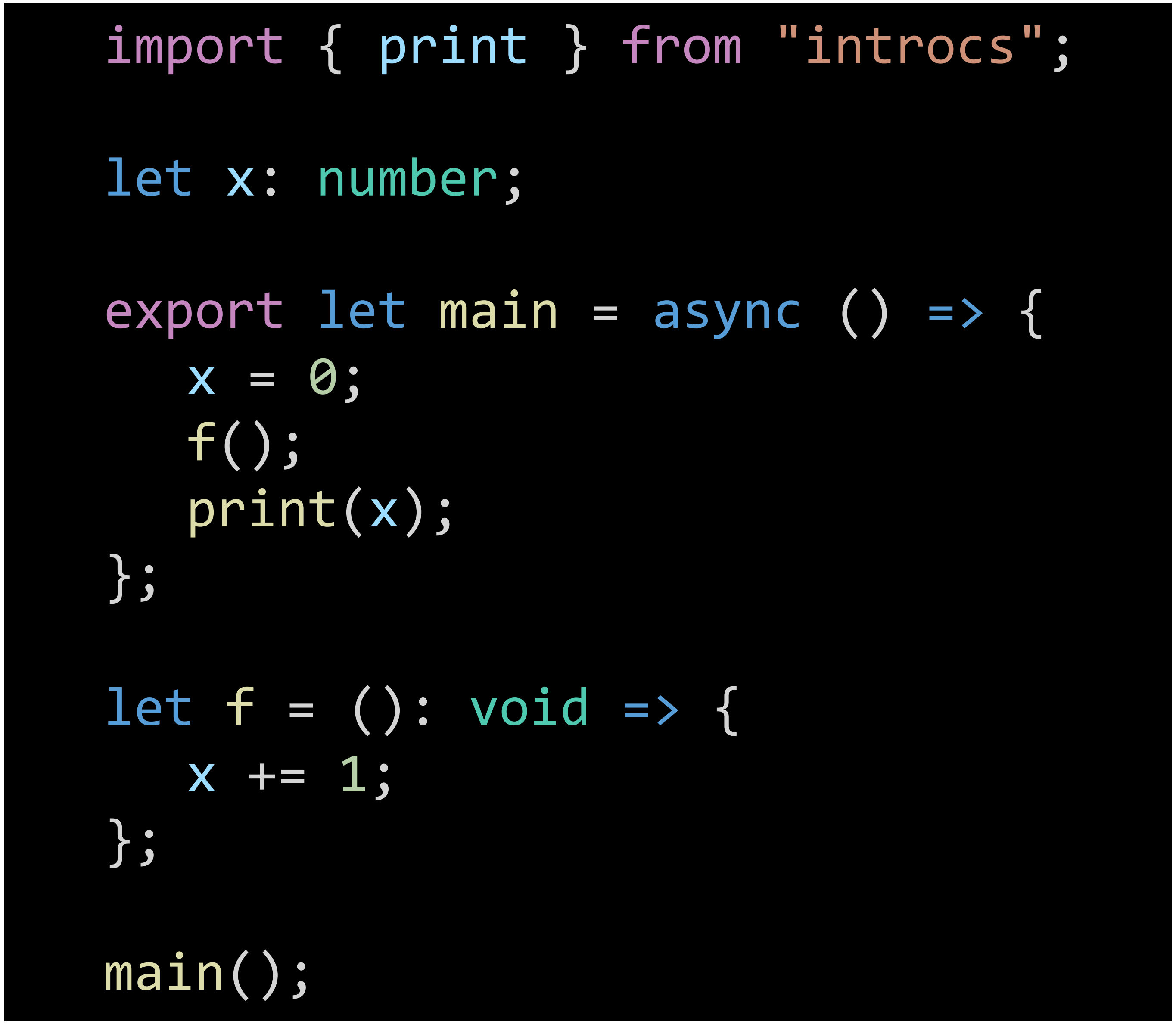
Videos for Thursday

- V15 - Classes and Objects: Conceptual Introduction
- V16 - Classes and Objects: Syntax
- Warm-up Quiz on Thursday - Take 1-page of Notes

Next Worksheet

- Will post tonight and focus on environment diagramming concepts
  - All concepts needed for first 3 diagrams are introduced today
  - Final diagram involves a concept you'll get exposure to in videos and we'll diagram on Thursday
- Submission Warning – Half Credit for Improperly Scanned Worksheets
  - 50% penalty for worksheets scanned out of order, in the wrong orientation, or significantly out of focus
  - Open your PDF on your computer before submitting to check it

Challenge Question #0 - What is printed?



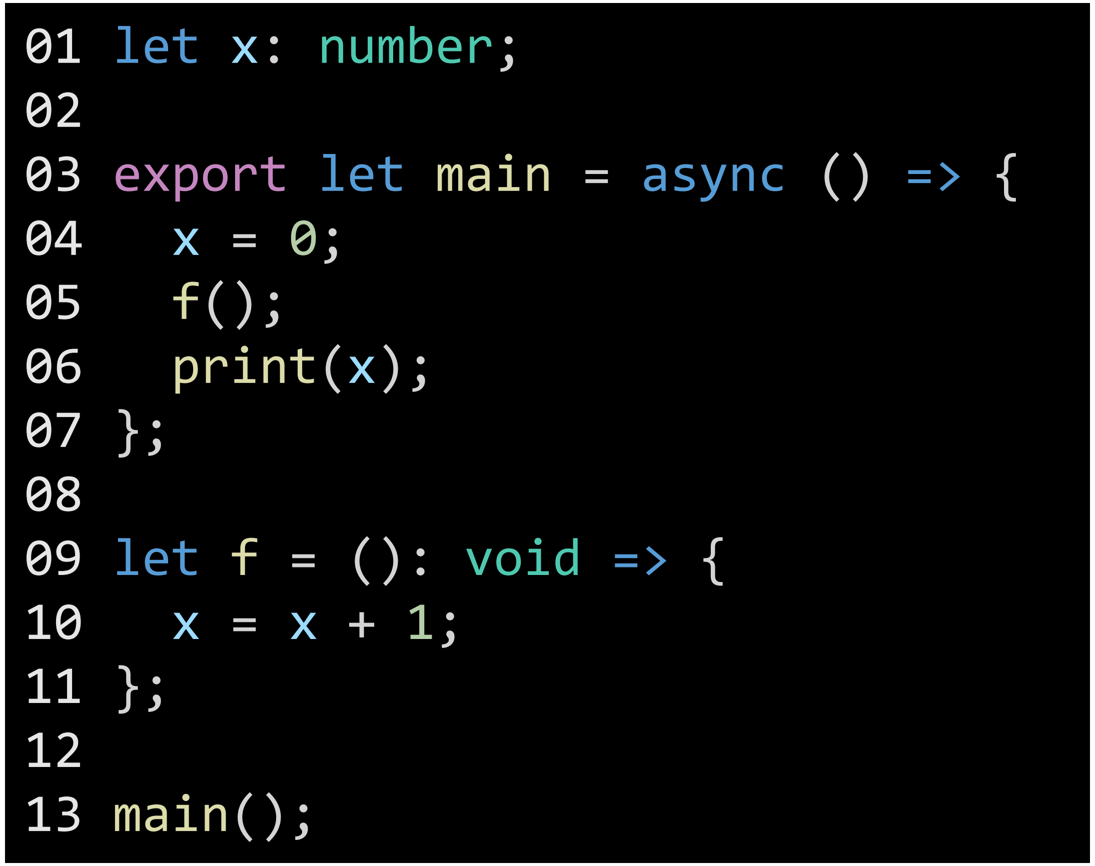
```
import { print } from "introcs";

let x: number;

export let main = async () => {
    x = 0;
    f();
    print(x);
};

let f = (): void => {
    x += 1;
};

main();
```

Notes on the Globals Frame

- The final segment of memory to add to our environment diagrams is the Globals frame.
- Your function definitions, when defined as we have so far, as well as global variables (introduced today) are bound in the Globals frame.
- The Globals frame is established before any function call frames on the call stack and behaves like any other frame when definitions are reached.

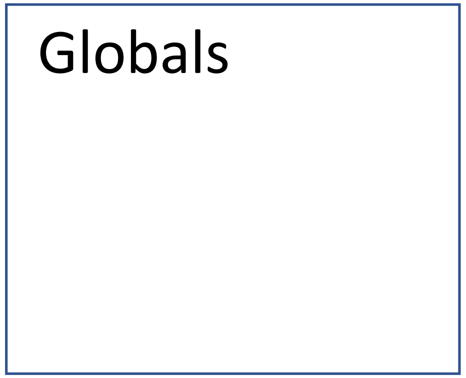
Globals - Starting Point

When a program* is loaded by an interpreter, it begins with an empty** stack and heap. The top-most frame of our stack is called the **Globals frame**.



```
let x: number;

export let main = async () => {
  x = 0;
  f();
  print(x);
};

let f = (): void => {
  x = x + 1;
};

main();
```

The Stack



The Heap

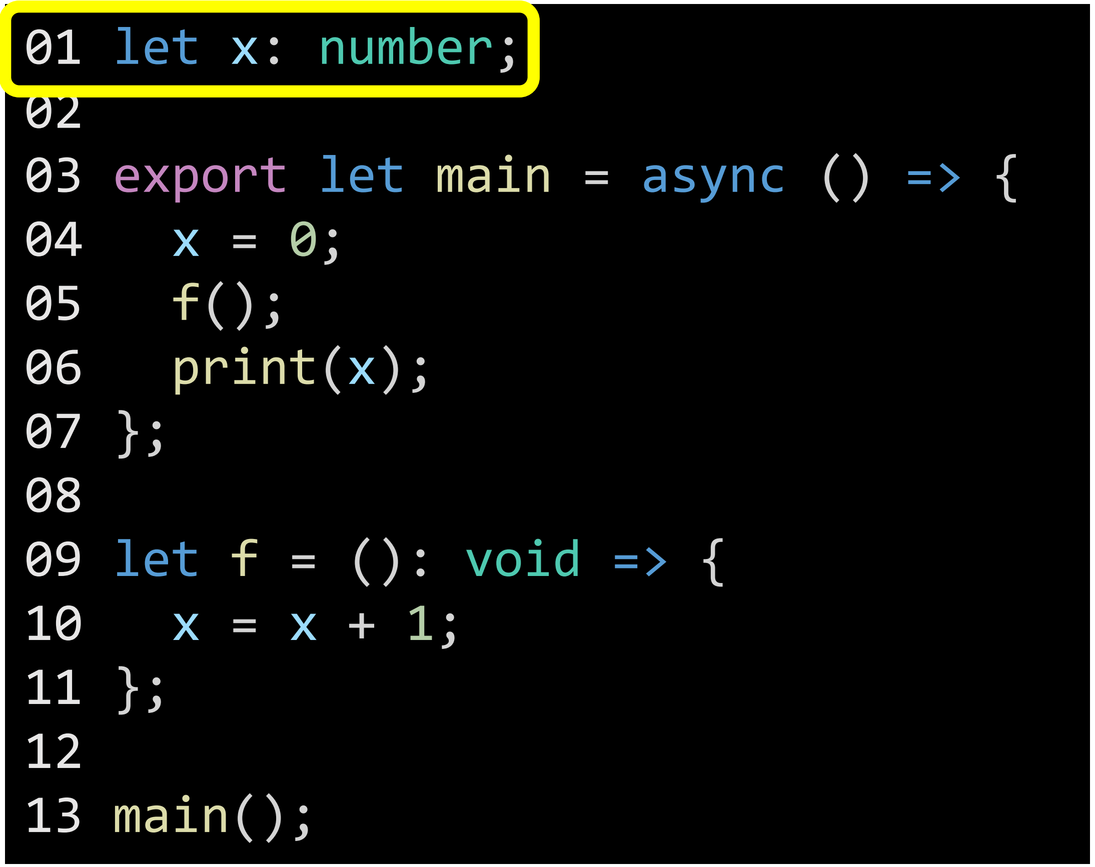
* The import `{ print }` statement is hidden in this example for illustration, but it should be there.

** This is a simplification for modeling purposes. In reality there are a lot of commonly used names established globally by the programming language (like Math)

Variable Declaration

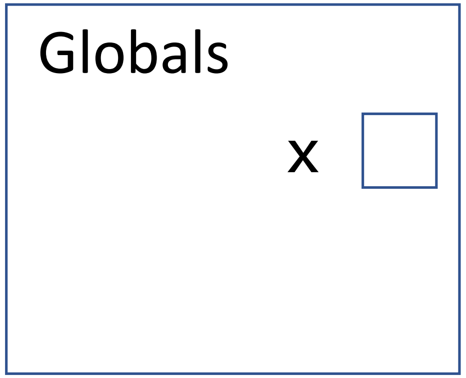
When a **variable** is declared, add its name to the current frame on the stack.

Since **x** is declared in the global frame it's called a **global variable**.



```
let x: number;

export let main = async () => {
  x = 0;
  f();
  print(x);
};

let f = (): void => {
  x = x + 1;
};

main();
```

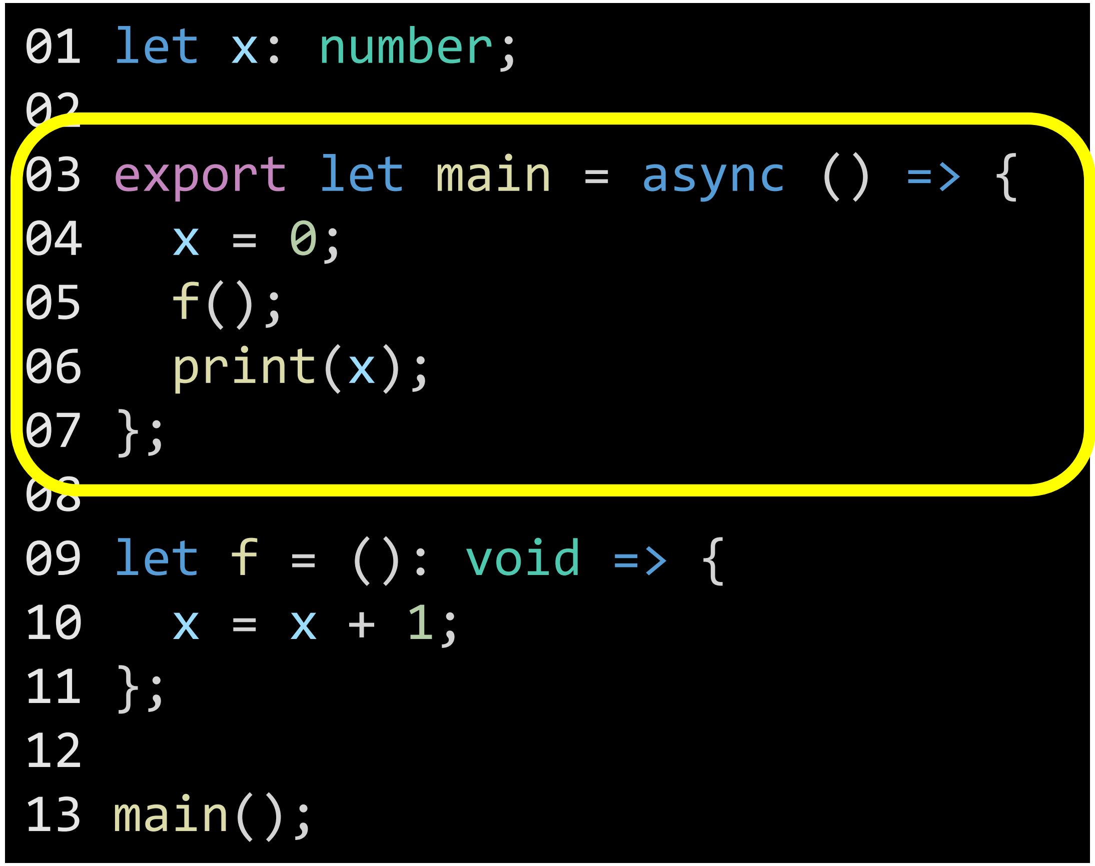
The Stack



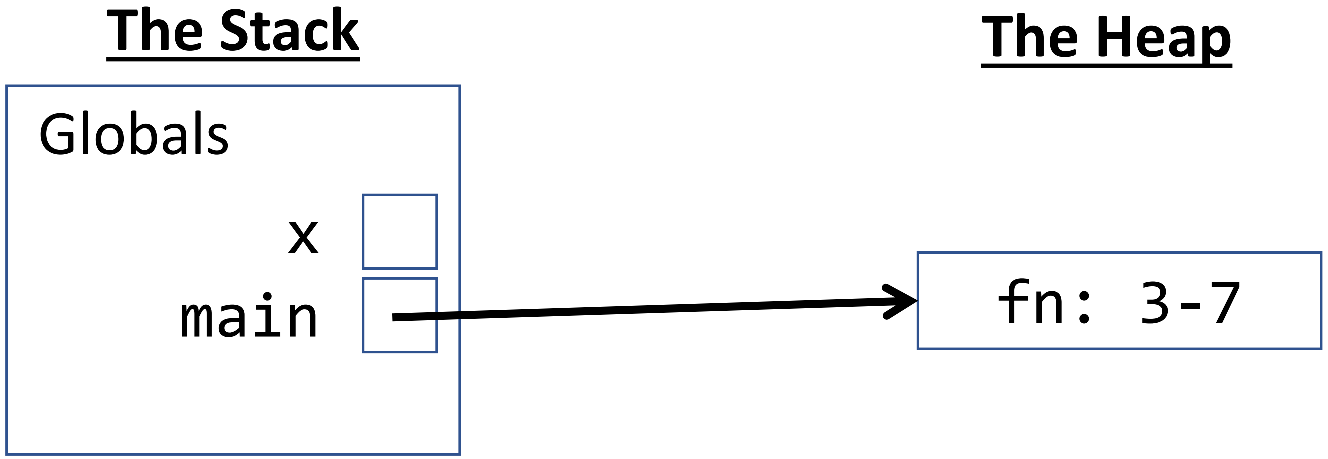
The Heap

Variable Declarations - Function

When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' on the heap with start-end lines.



```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



The Stack

Globals

x

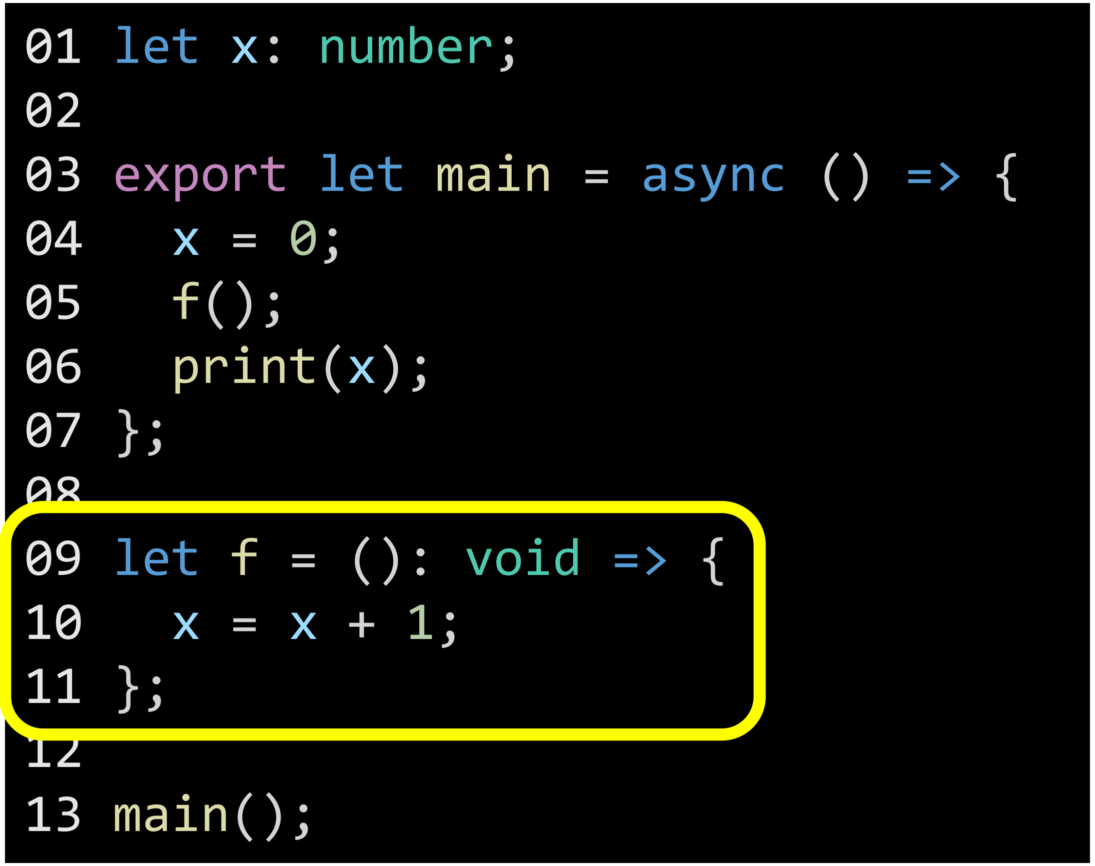
main

The Heap

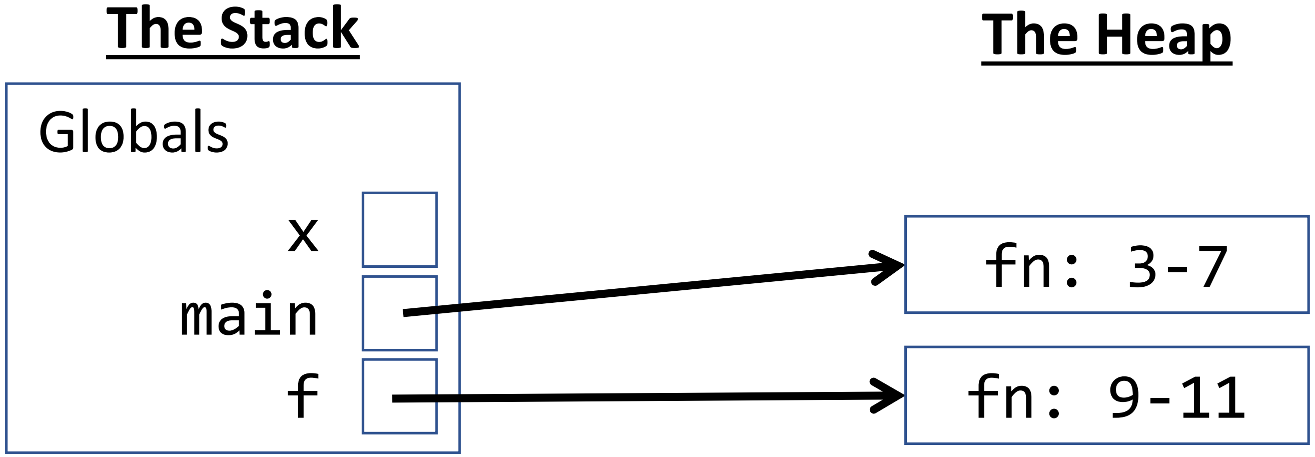
fn: 3-7

Variable Declarations - Function

When a function definition is encountered, you will add its name to the current stack frame connected to a shorthand 'fn' on the heap with start-end lines.



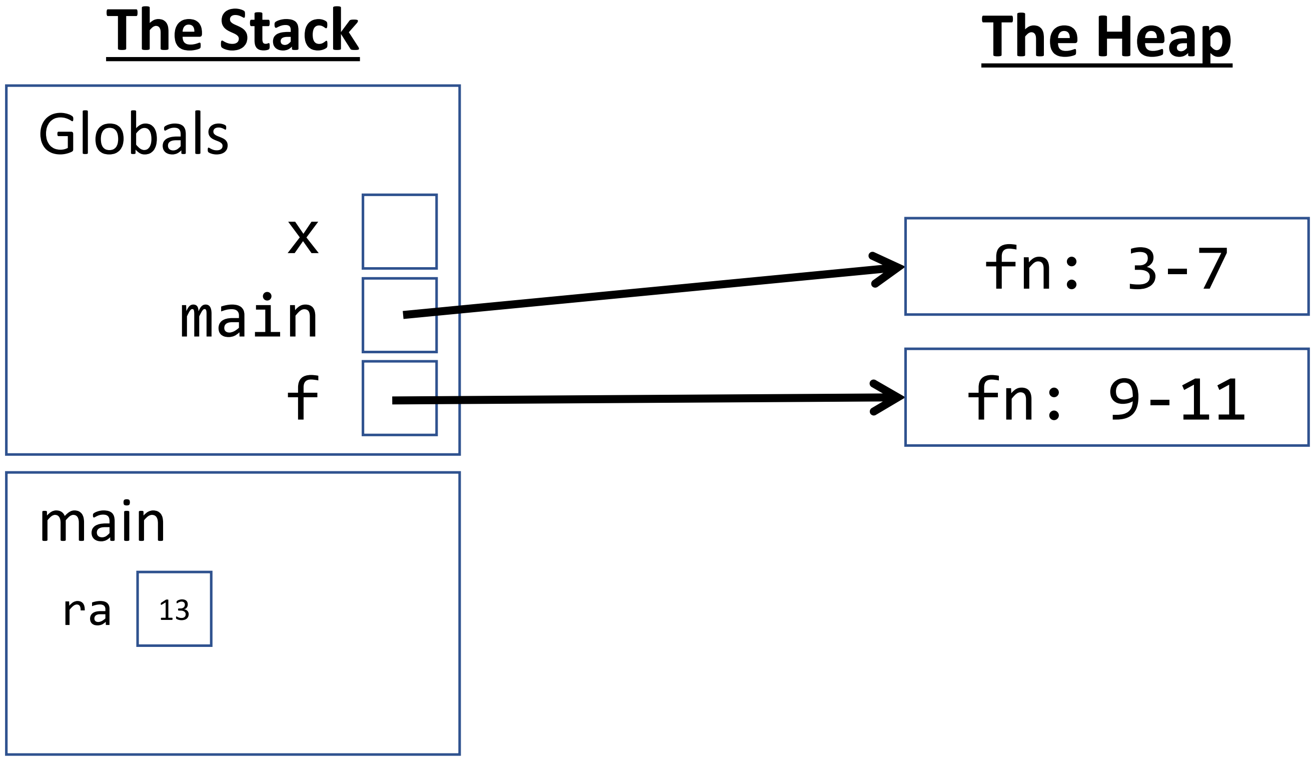
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Function Call

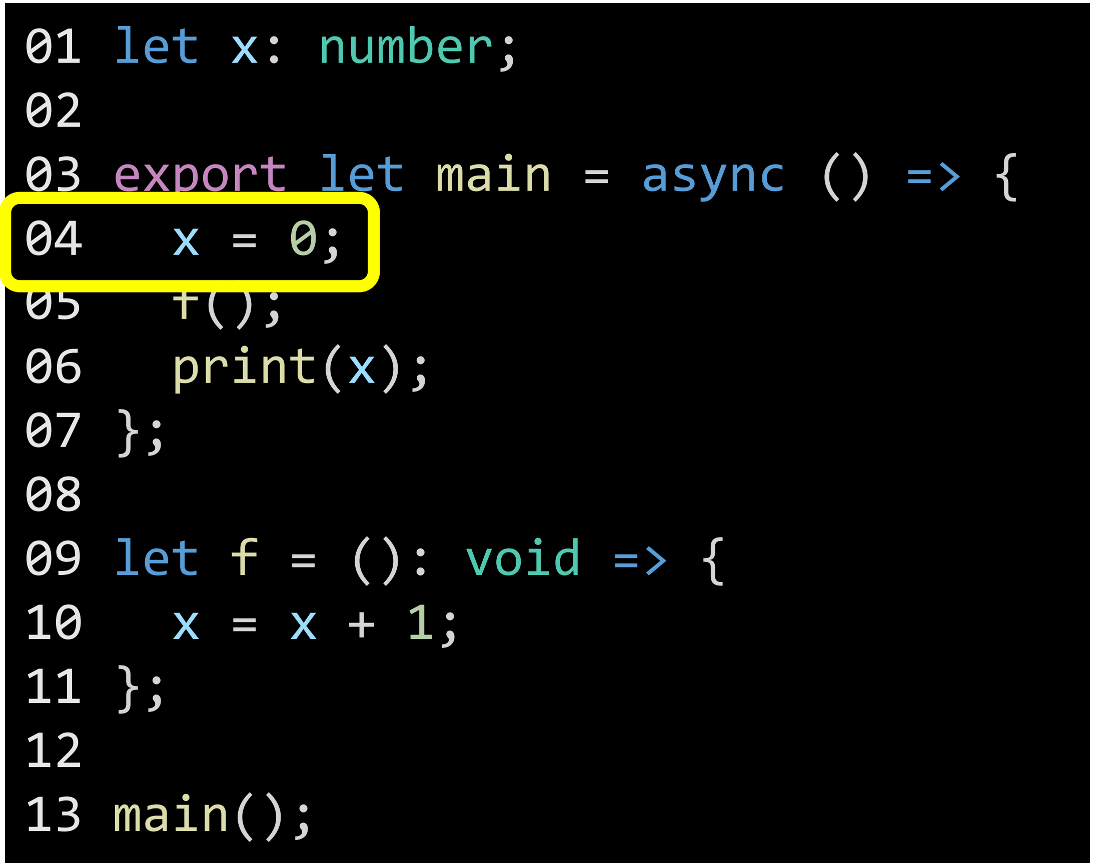
When a function call is encountered, a new **frame** is added to your stack. Label it with the function name. When it has parameters, you'll need to add them, too.

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

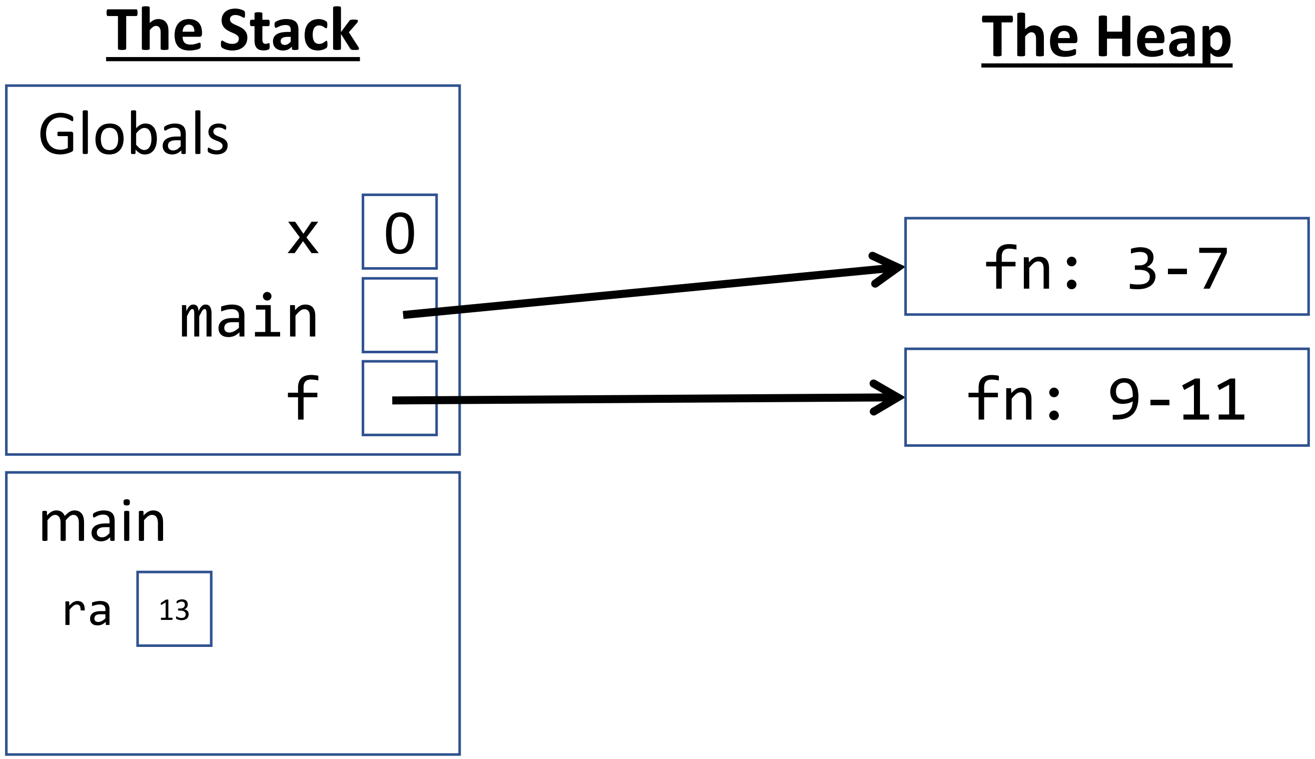


Name Resolution - Variable Assignment

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals. Primitive? Assign value in stack.

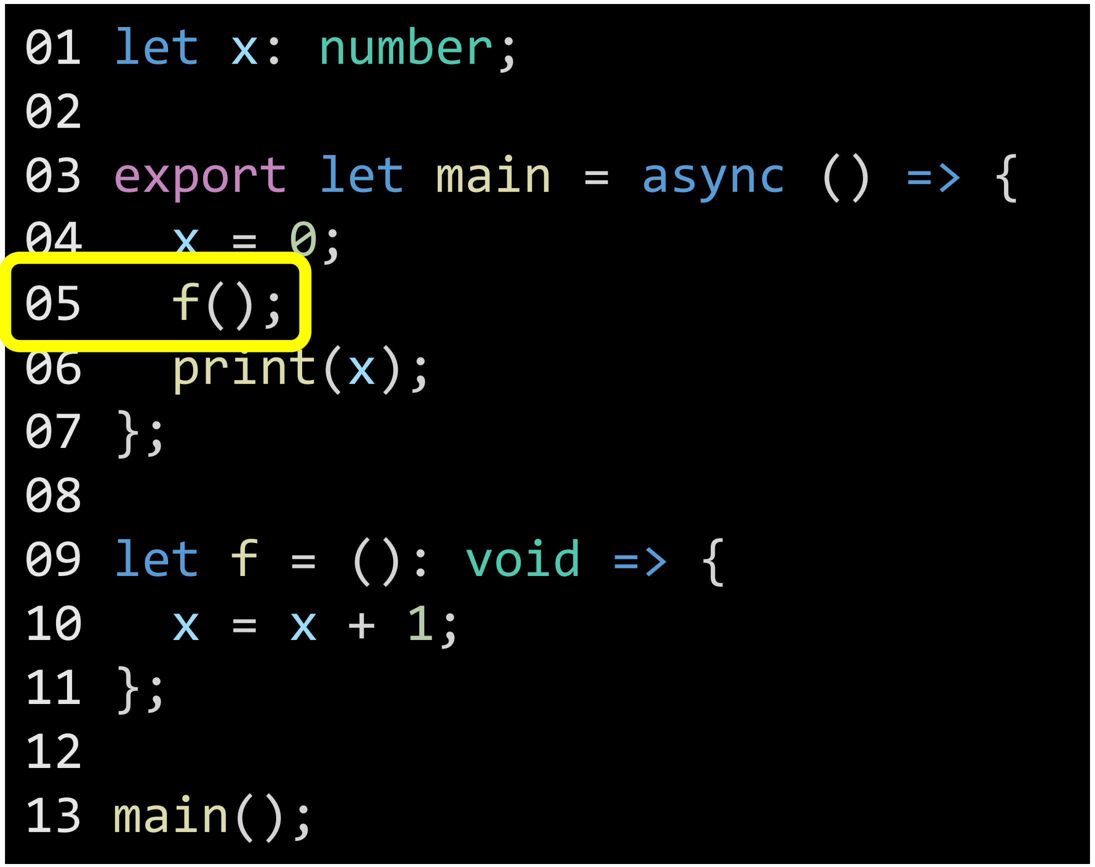


```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

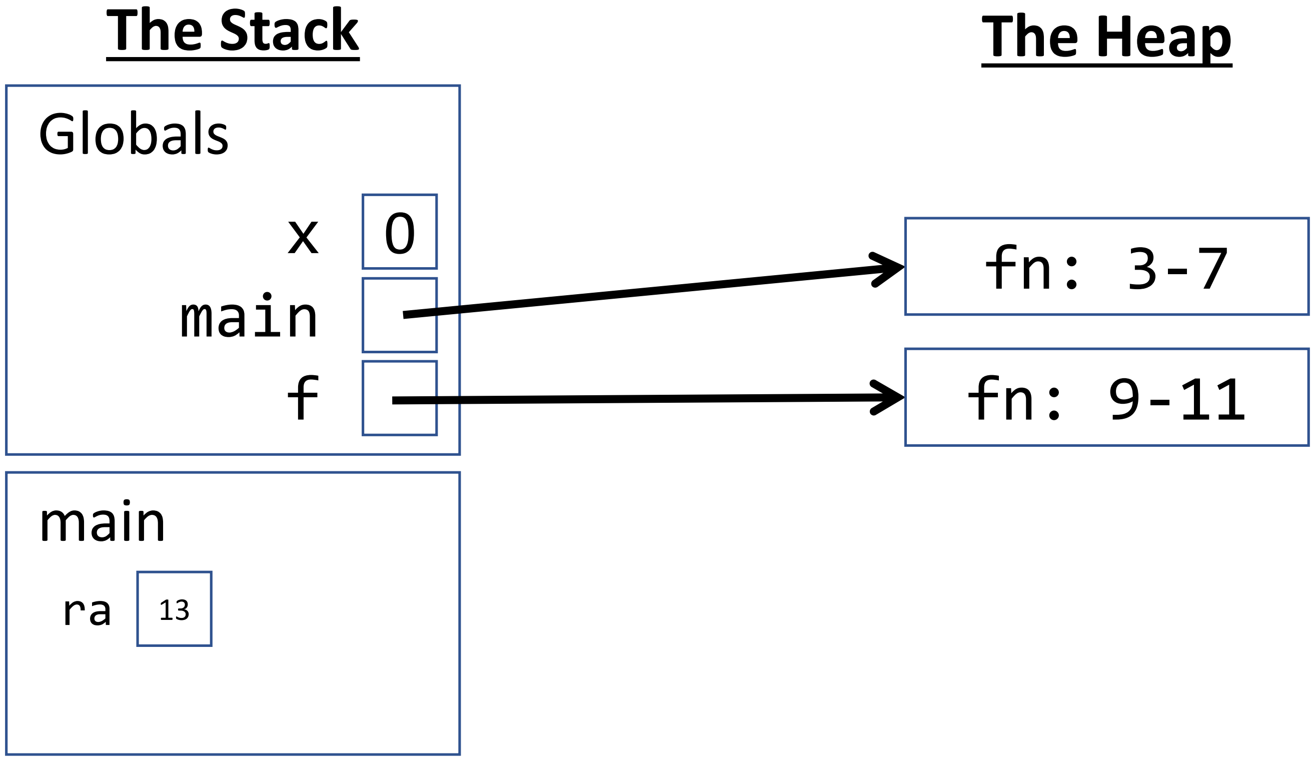


Function Call - Name Resolution

How do you know what the name **f** is? First look for a name in the current stack frame. If not there, then look in the globals. Notice **f** is defined in Globals.

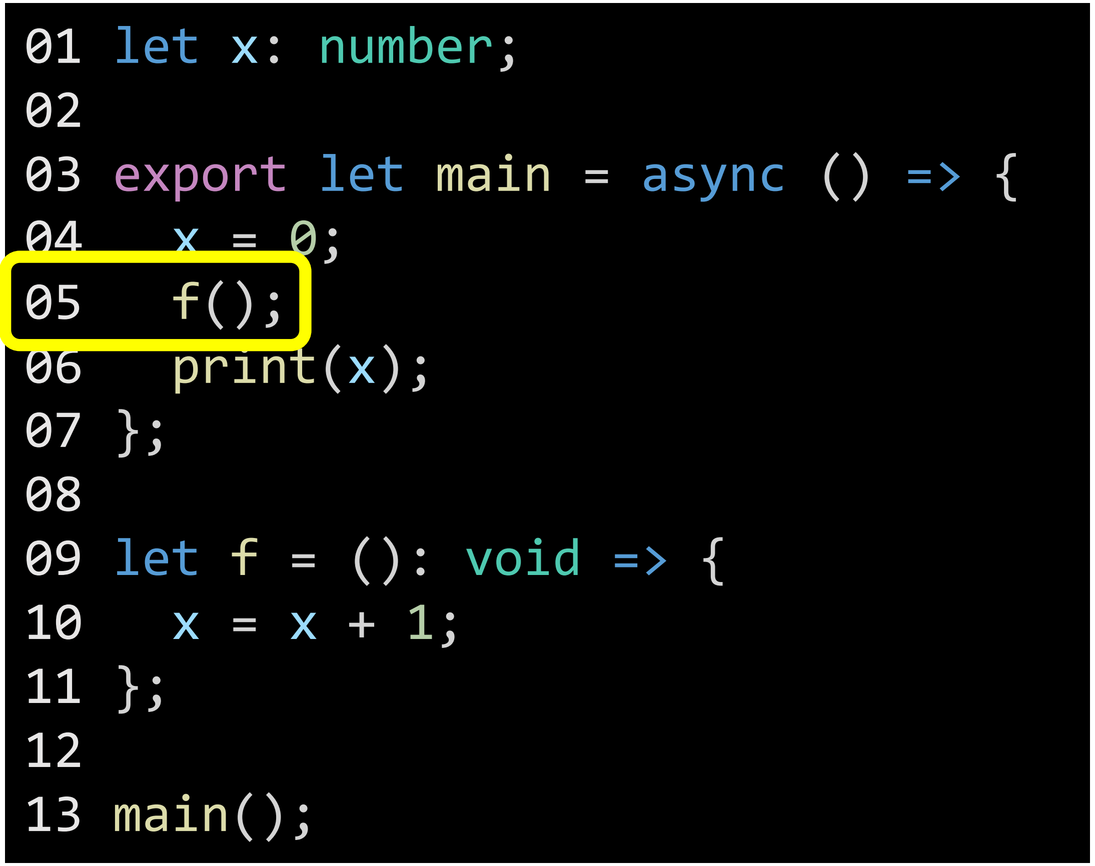


```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

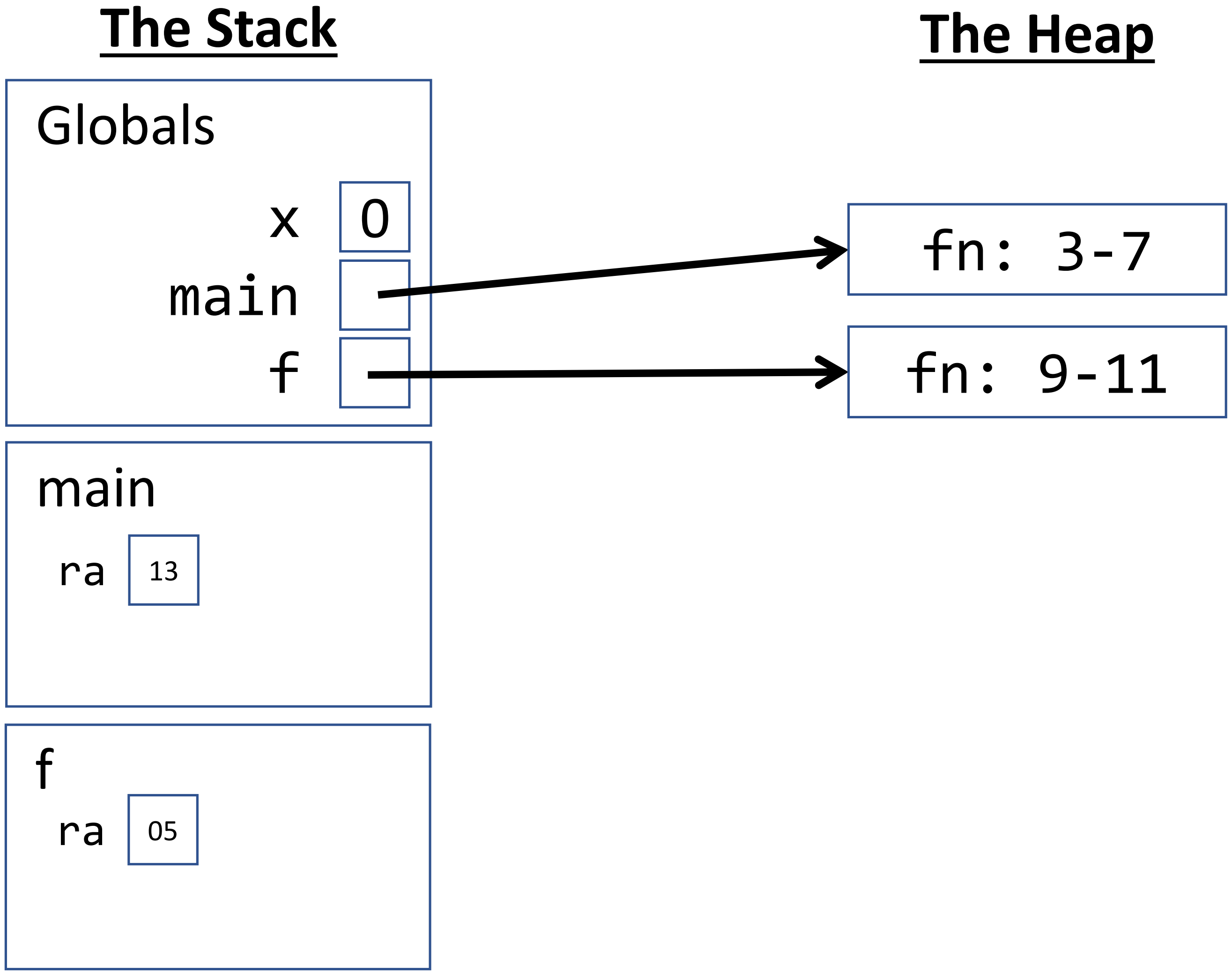


Function Call

When a valid function call is encountered, a new **frame** is added to your stack. Label it with the name, add return address line, and establish parameters.

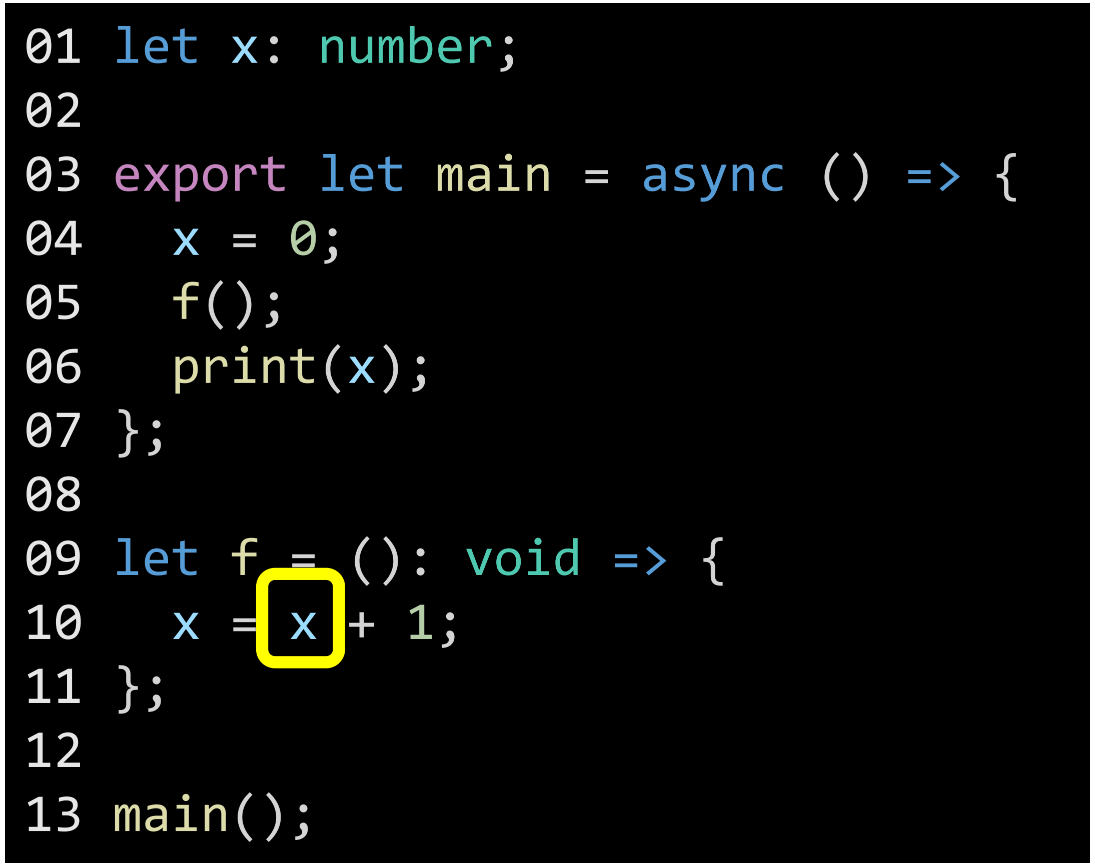


```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```

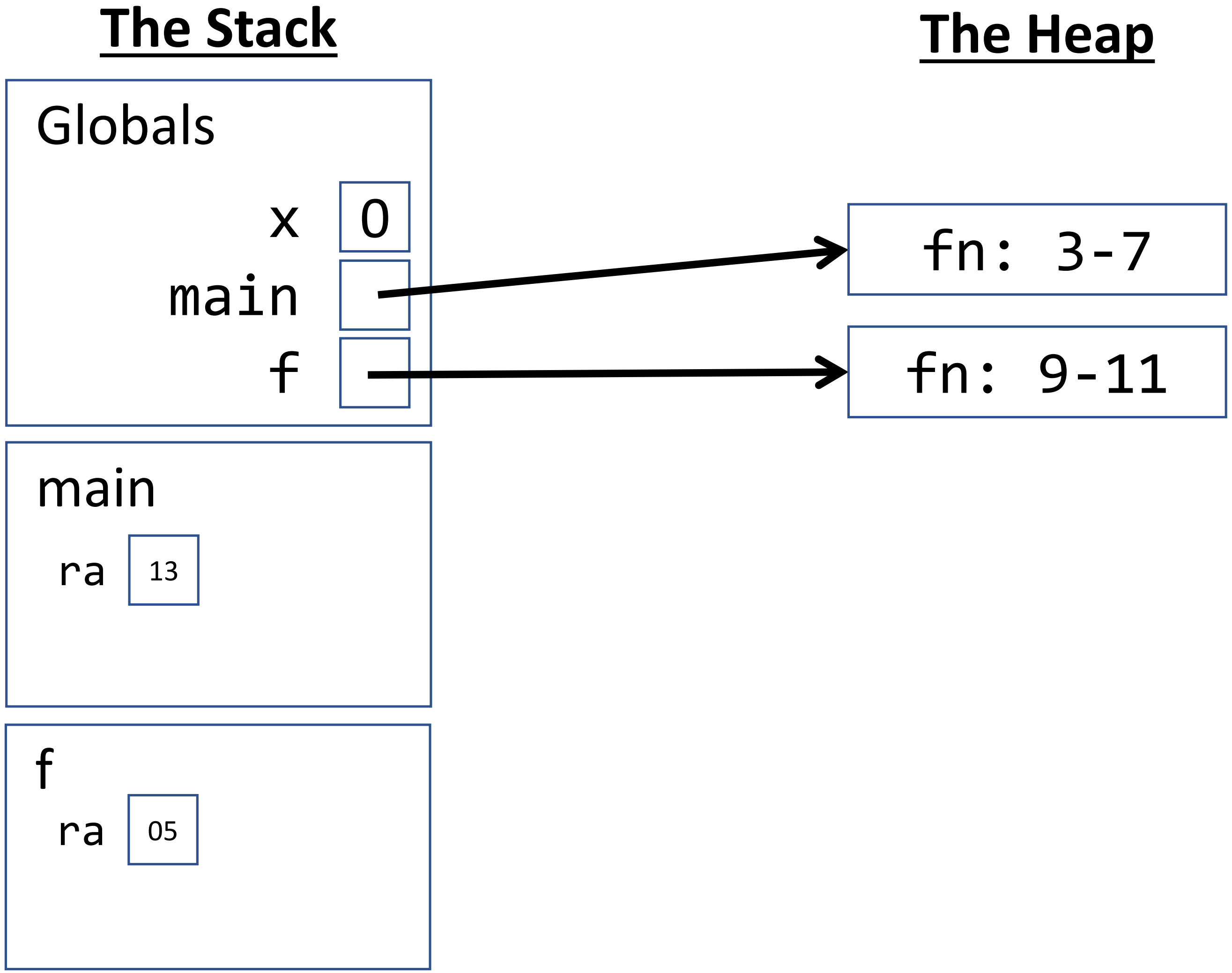


Name Resolution - Variable Access

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. In this case, it's in globals!



```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



The Stack

The Heap

Globals

x 0

main

f

fn: 3-7

fn: 9-11

main

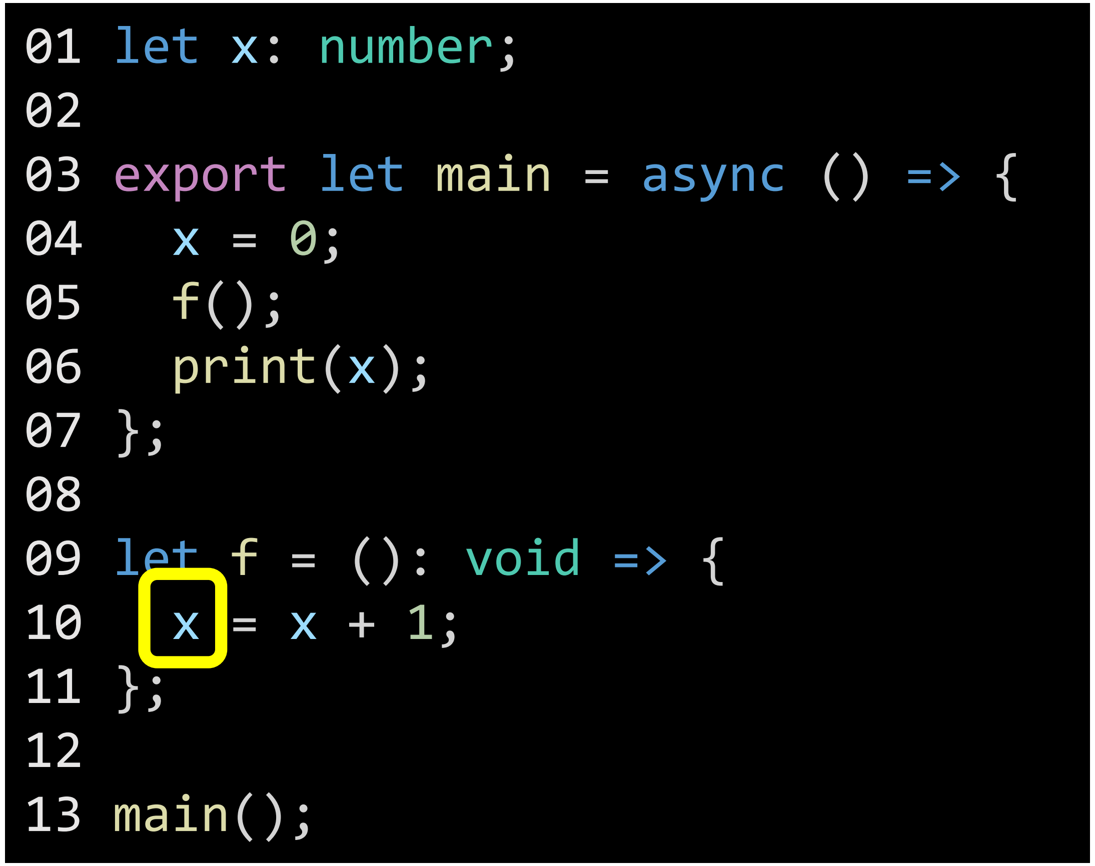
ra 13

f

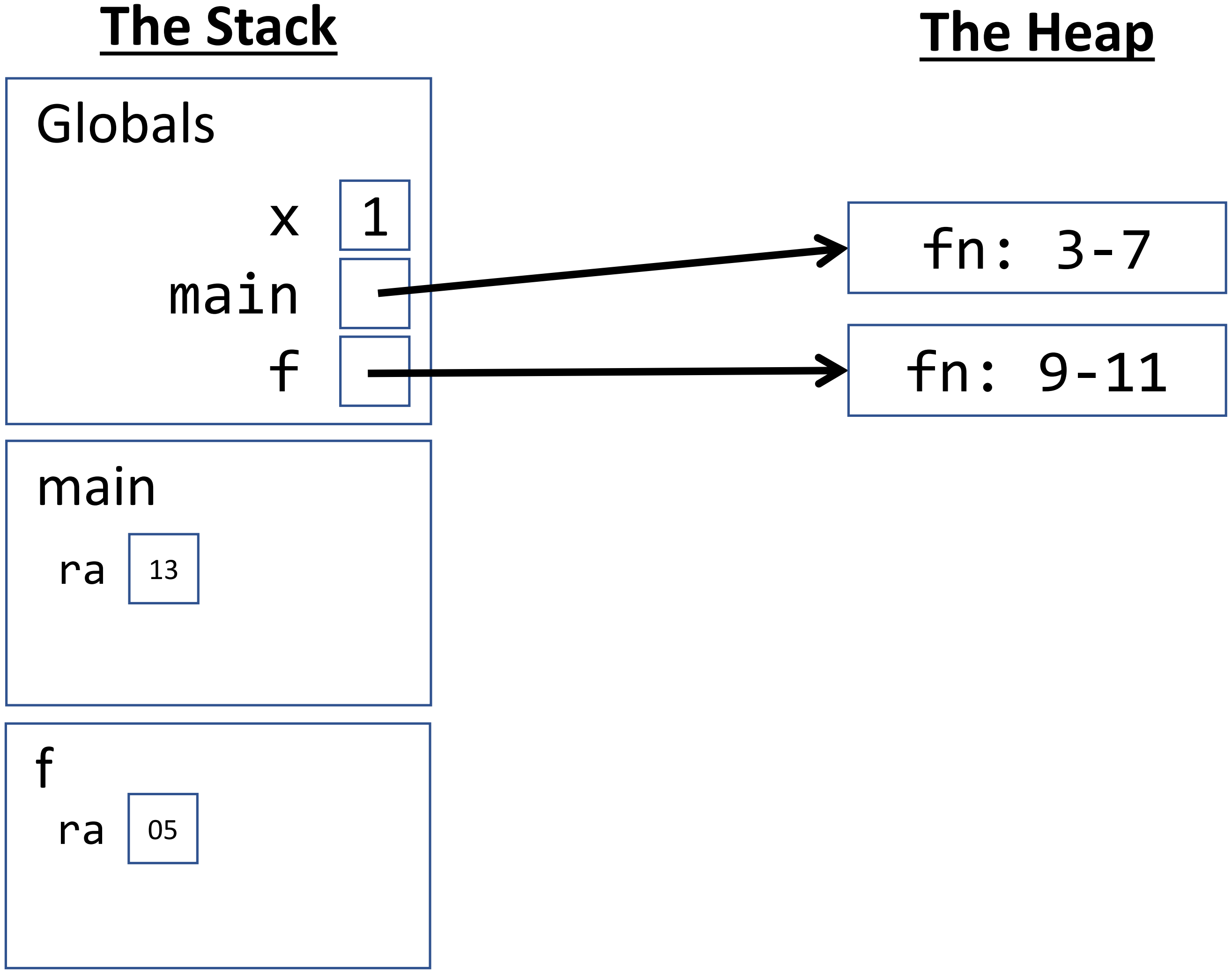
ra 05

Name Resolution - Variable Assignment

When a **primitive variable** is assigned a value, first resolve its frame location by name, then update its value.



```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



The Stack

The Heap

Globals

x 1

main

f

fn: 3-7

fn: 9-11

main

ra 13

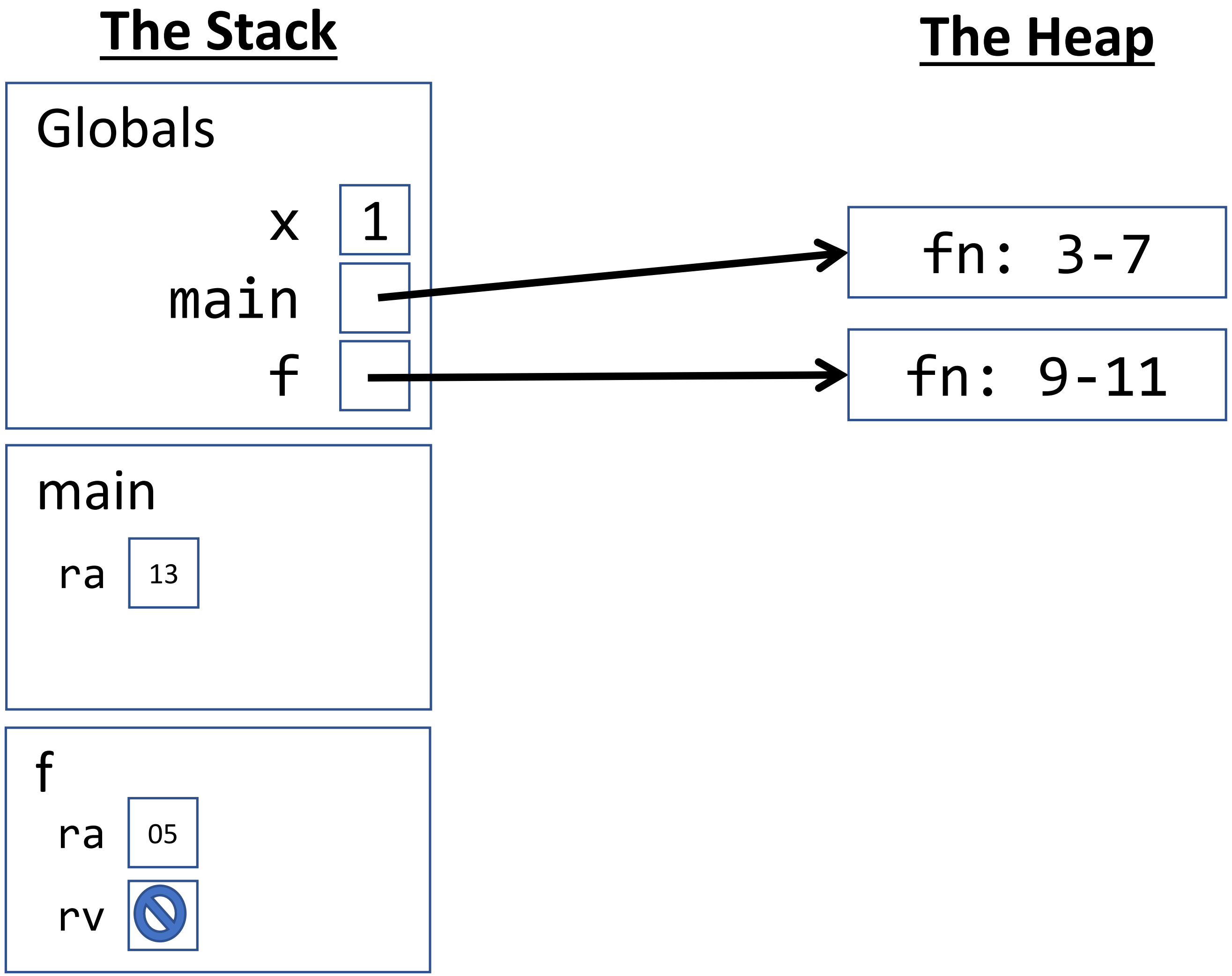
f

ra 05

Function Return - **void** Functions

When a **void function** completes, it returns nothing and control jumps back to the return address.

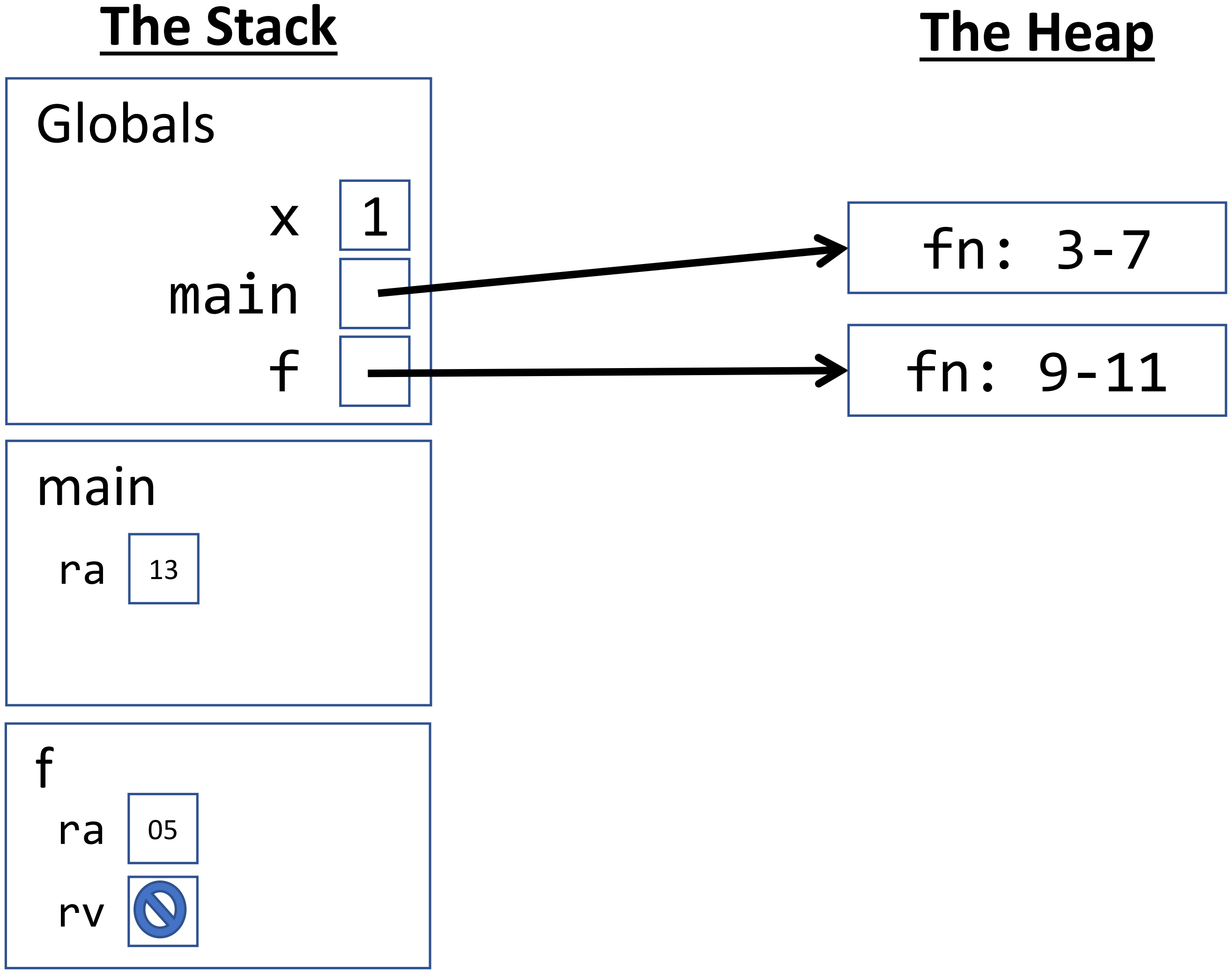
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Name Resolution - Variable Access

How do you know what the name **x** is? First look for a name in the current stack frame. If not there, then look in the globals frame. **In this case, it's in globals**!

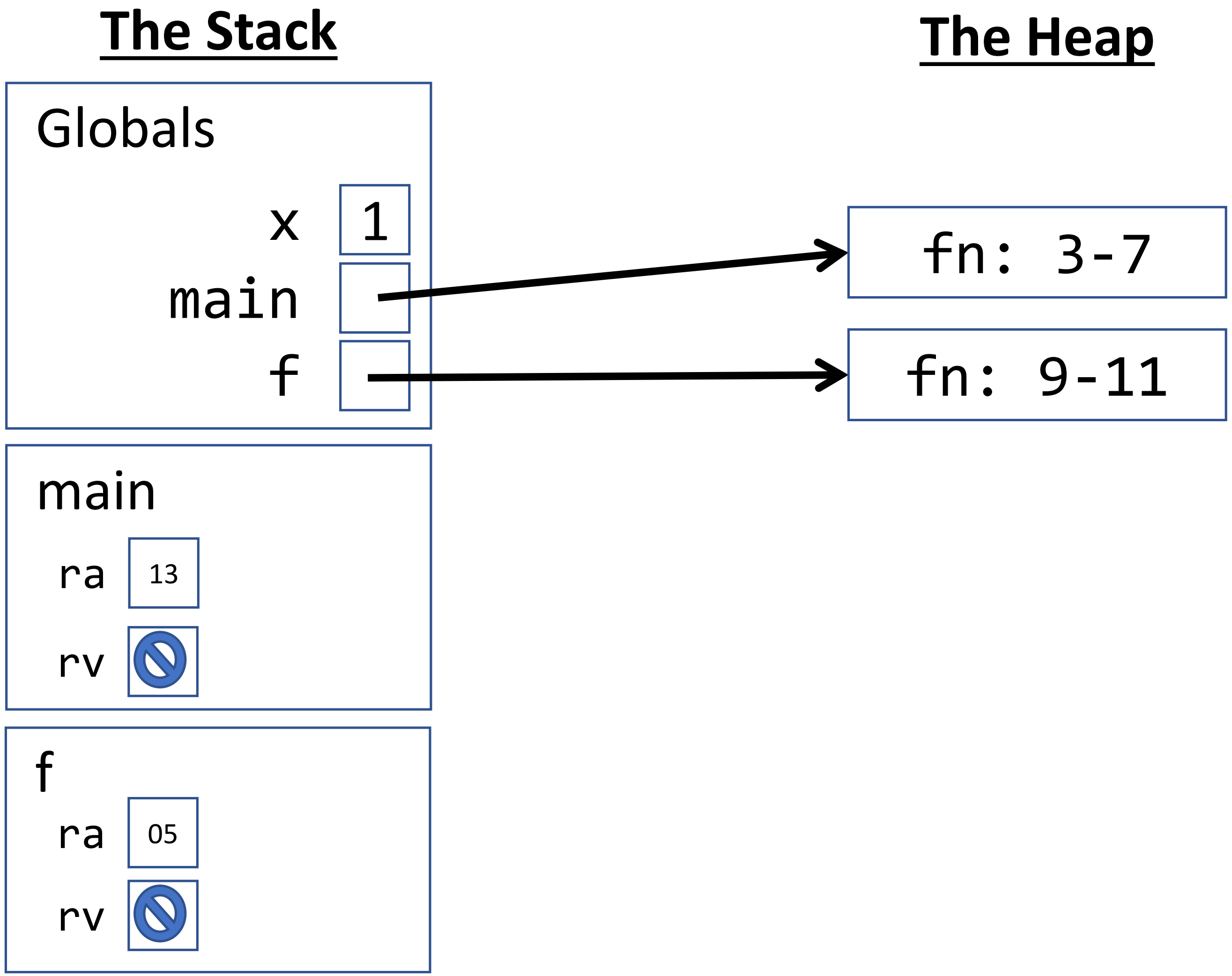
```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



Function Return - **void** Functions

When a **void function** completes, it returns nothing and control jumps back to the return address.

```
01 let x: number;
02
03 export let main = async () => {
04   x = 0;
05   f();
06   print(x);
07 };
08
09 let f = (): void => {
10   x = x + 1;
11 };
12
13 main();
```



The Stack

Globals

x 1

main

f

main

ra 13

rv ⊘

f

ra 05

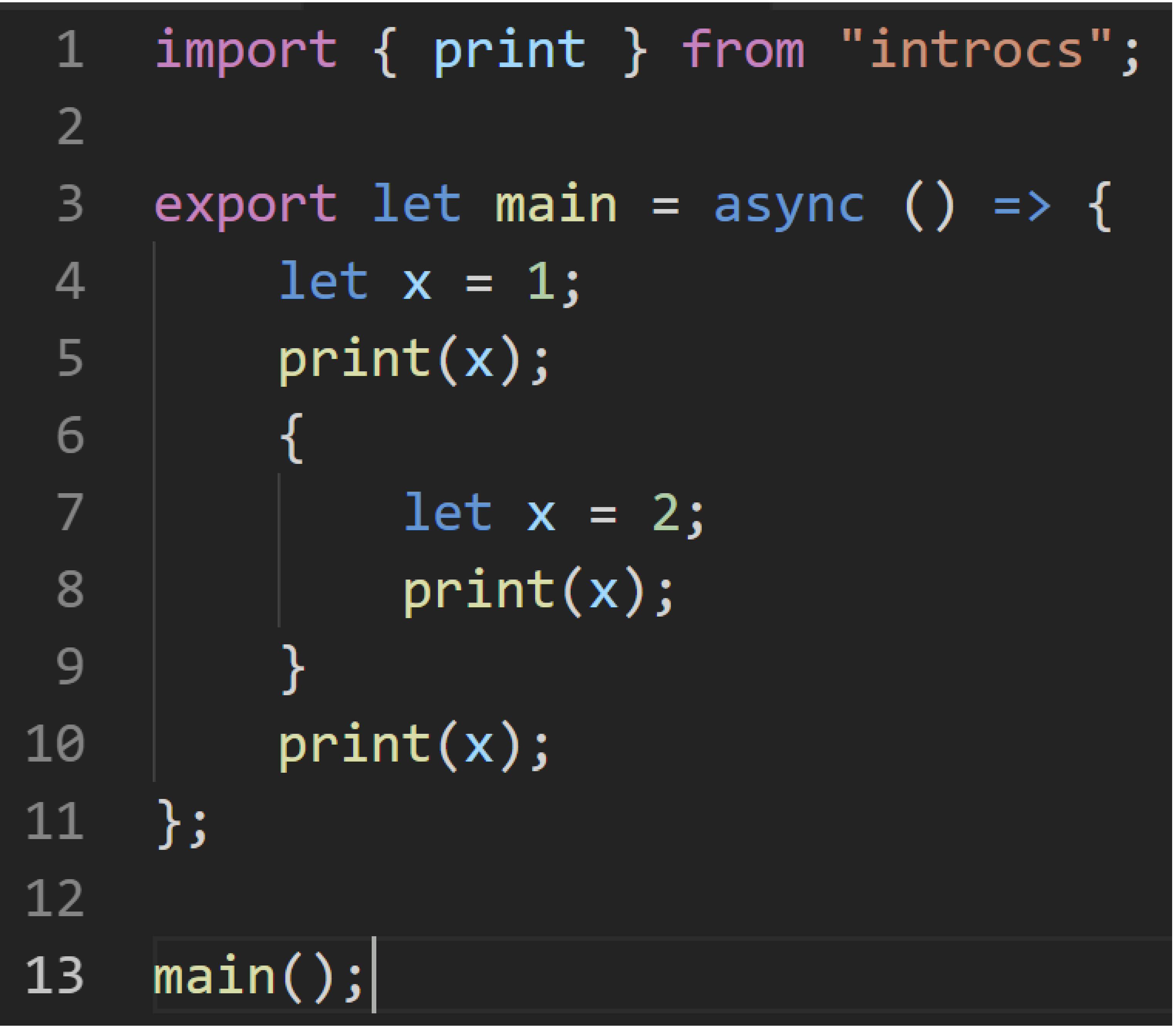
rv ⊘

The Heap

fn: 3-7

fn: 9-11

Challenge Question 1: What is the output?



```
import { print } from "introcs";

export let main = async () => {
    let x = 1;
    print(x);
    {
        let x = 2;
        print(x);
    }
    print(x);
};

main();
```

Block Statements, Scope, and Shadowing

Block Statements (1 / 2)

- A **block** is a special kind of statement that groups multiple, related statements.

- Blocks are **enclosing curly braces** that "contain" its statements.

```
{
   // This is a block statement
   print("Statement one");
   print("Statement two");
}
```

- Blocks do not end with a semicolon after the closing curly brace. The closing curly signals the end of the block.

- The *then-block* and *else-block* of *if* statements, as well as the *repeat-block* of loops, are all *just blocks* like the one shown above.

Block Statements (2 / 2)

- Anywhere you can write a statement, you can also write a block statement.
  - Thus, you can nest inner blocks inside of outer blocks.

```
{
  print("Statement one");
  {
       print("Statement two");
  }
  print("Statement three");
}
```

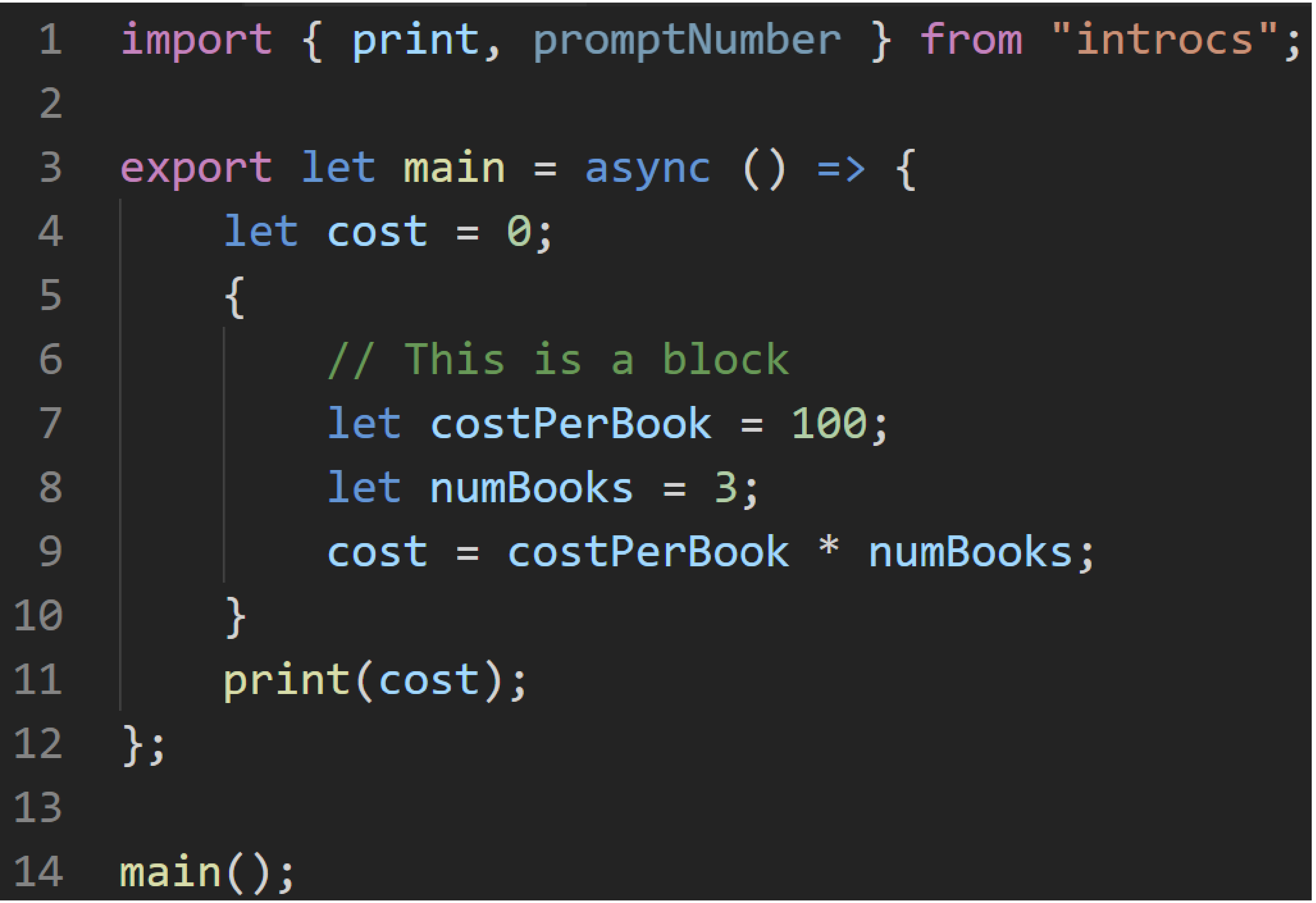
- **Important** formatting rule: **each statement inside of a block is indented one additional level**!

Variable's **Block Scope** Rule

- A variable is only accessible after it is declared in the same block or in an outer, containing block.

```
{
   let x = 0;
   print(x); // OK! x declared in same block
   {
      print(x); // OK! x declared in outer block
   }
}

print(x); // ERROR! x declared in different block
```

Blocks in Environment Diagrams

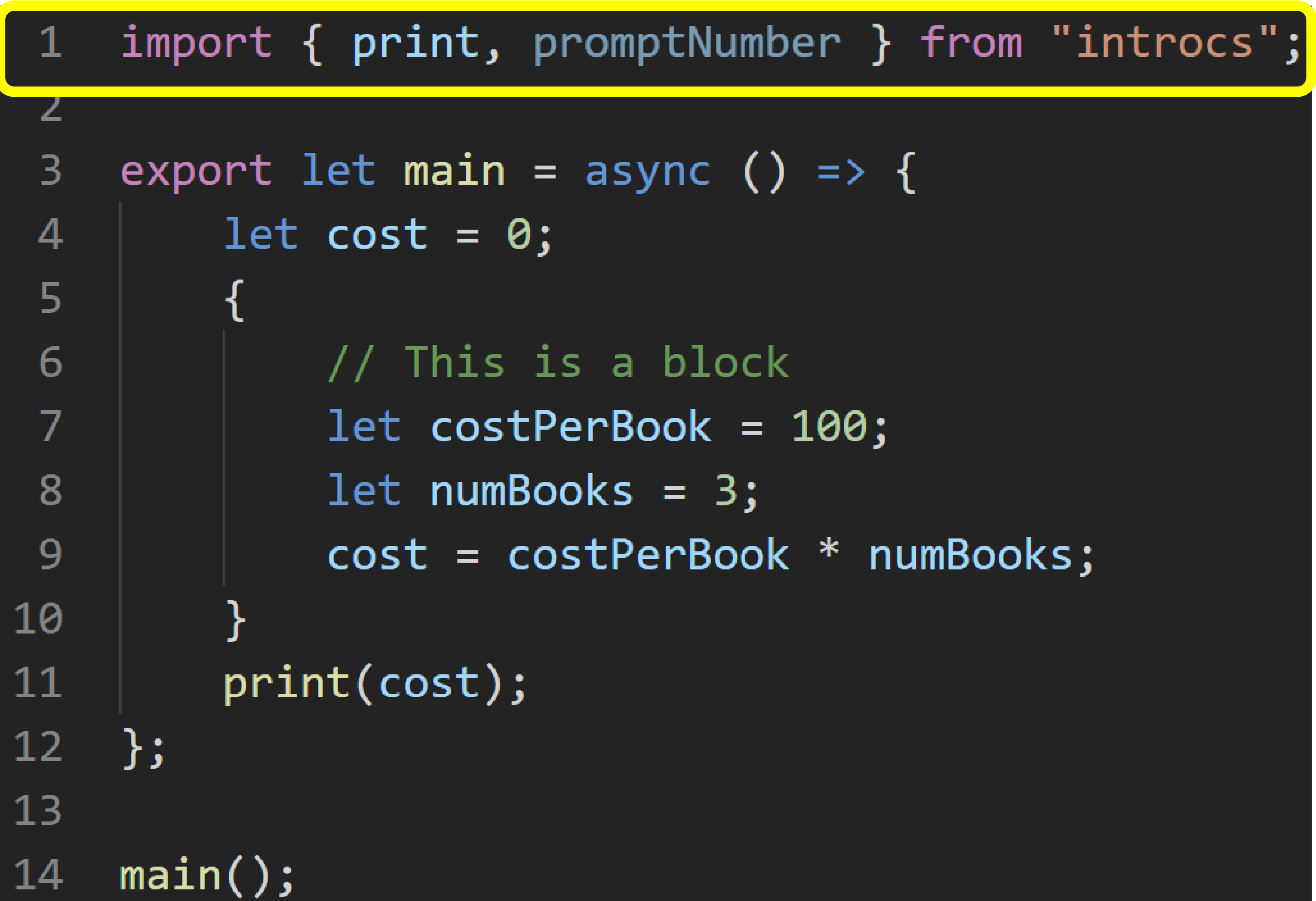


```
import { print, promptNumber } from "introcs";

export let main = async () => {
    let cost = 0;
    {
        // This is a block
        let costPerBook = 100;
        let numBooks = 3;
        cost = costPerBook * numBooks;
    }
    print(cost);
};

main();
```

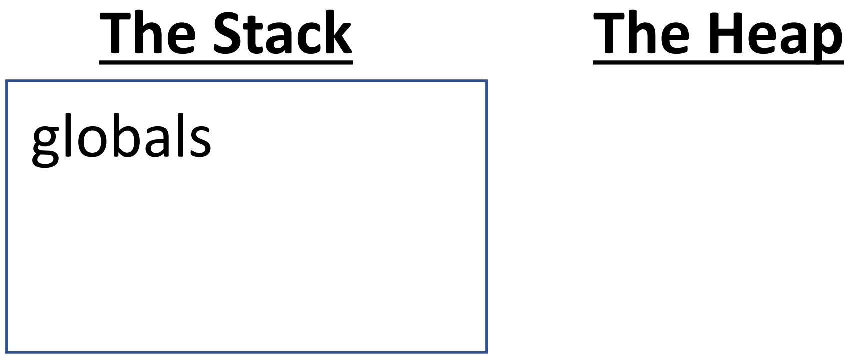
- Let's use the program to the left to illustrate how block scopes are represented in environment diagrams.

Imports

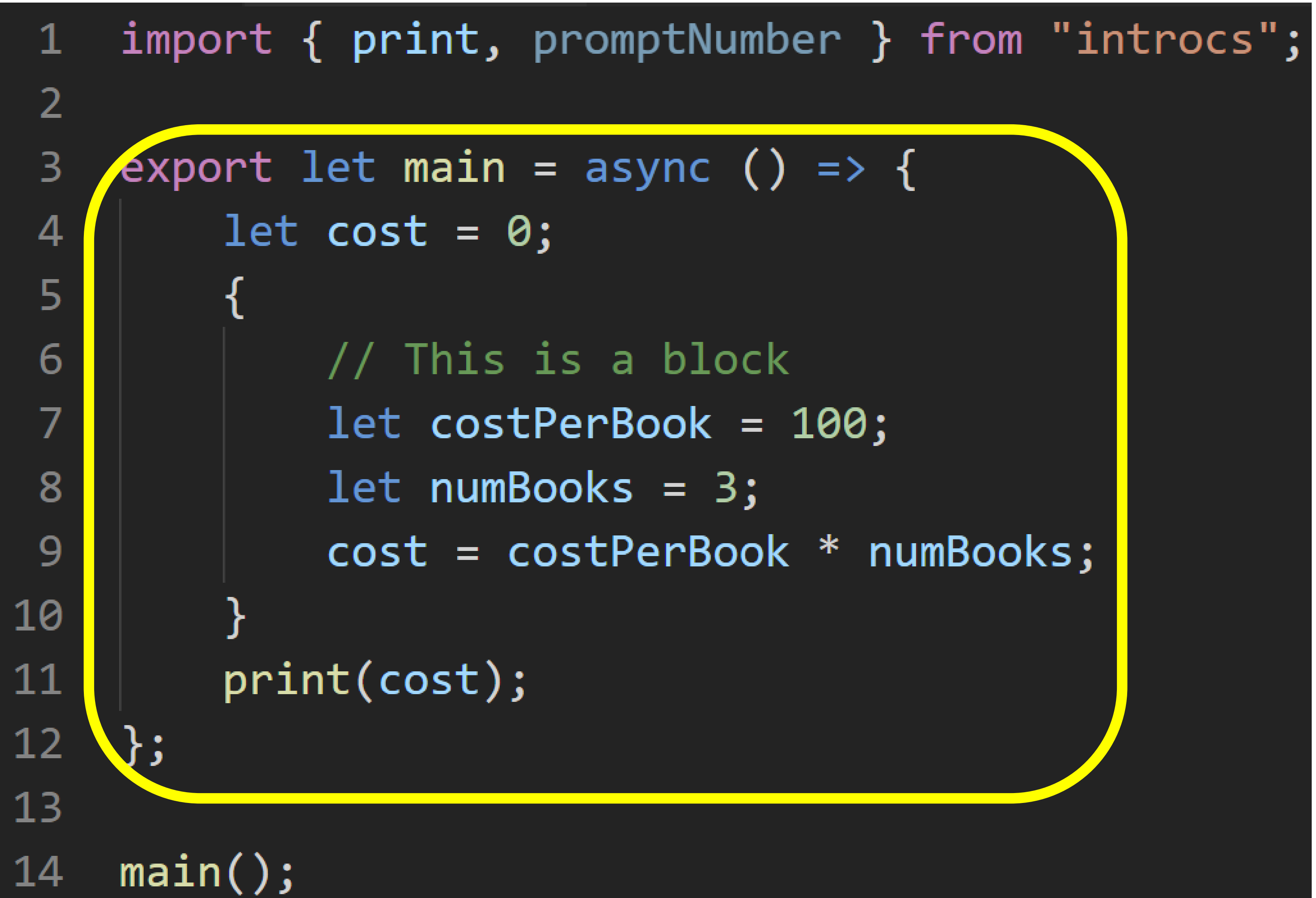
We will not represent imported names (like functions) in our diagrams. Technically, these names would be entered into the globals frame and bound to their definitions.



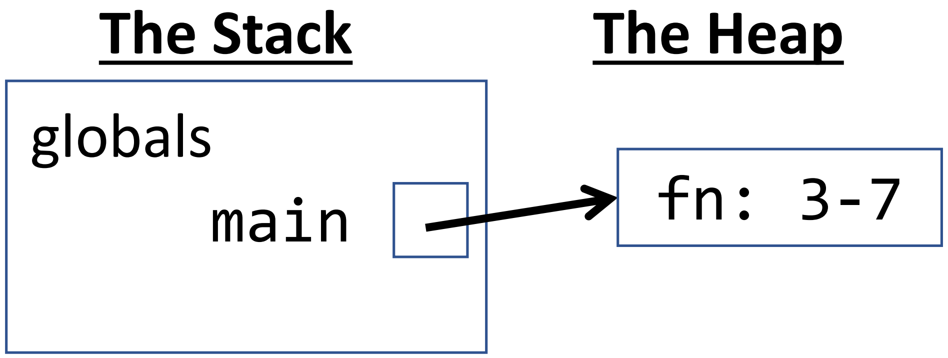
```
import { print, promptNumber } from "introcs";

export let main = async () => {
    let cost = 0;
    {
        // This is a block
        let costPerBook = 100;
        let numBooks = 3;
        cost = costPerBook * numBooks;
    }
    print(cost);
};

main();
```



Function Definition



```
import { print, promptNumber } from "introcs";

export let main = async () => {
    let cost = 0;
    {
        // This is a block
        let costPerBook = 100;
        let numBooks = 3;
        cost = costPerBook * numBooks;
    }
    print(cost);
};

main();
```



The Stack

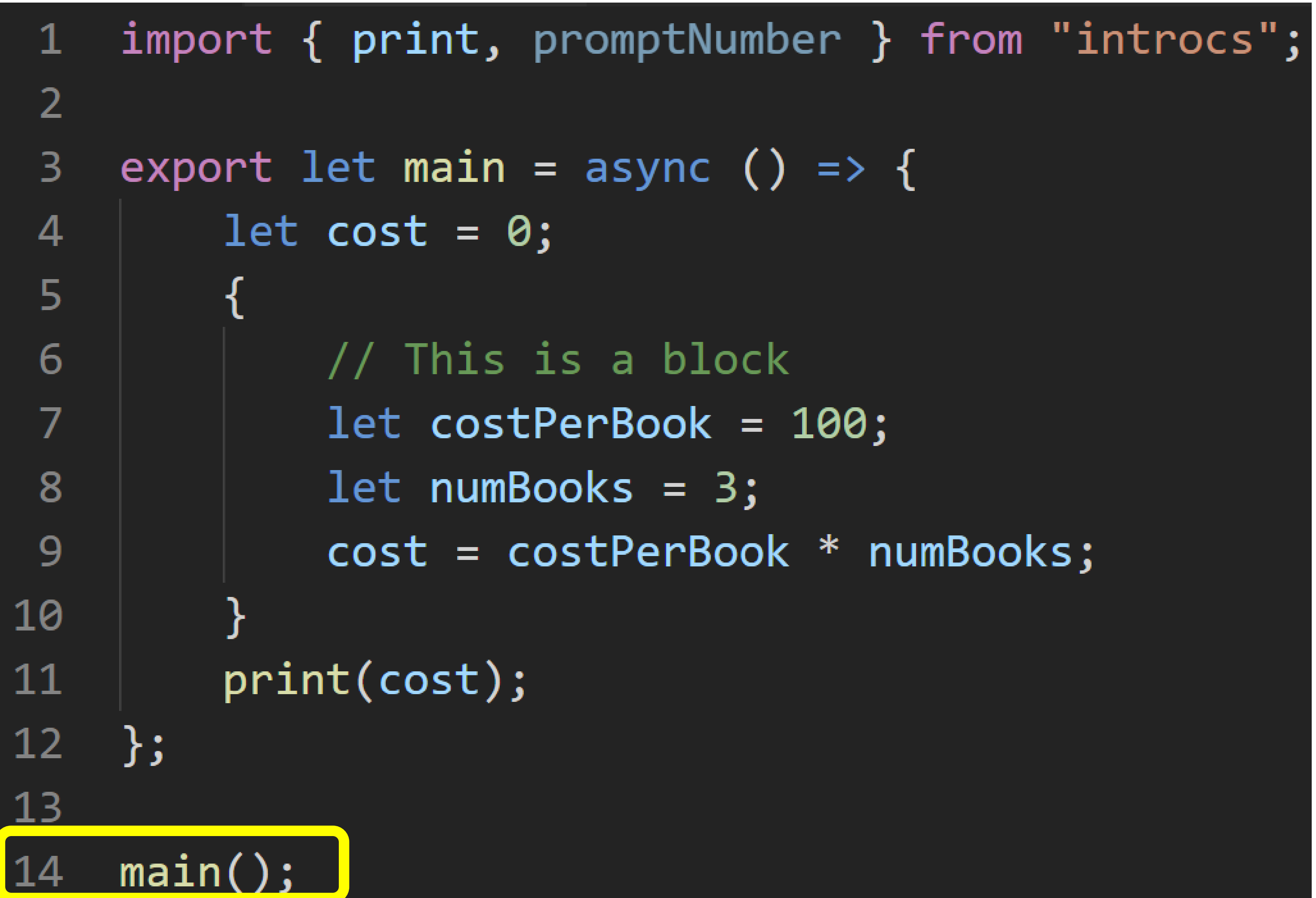
The Heap

globals

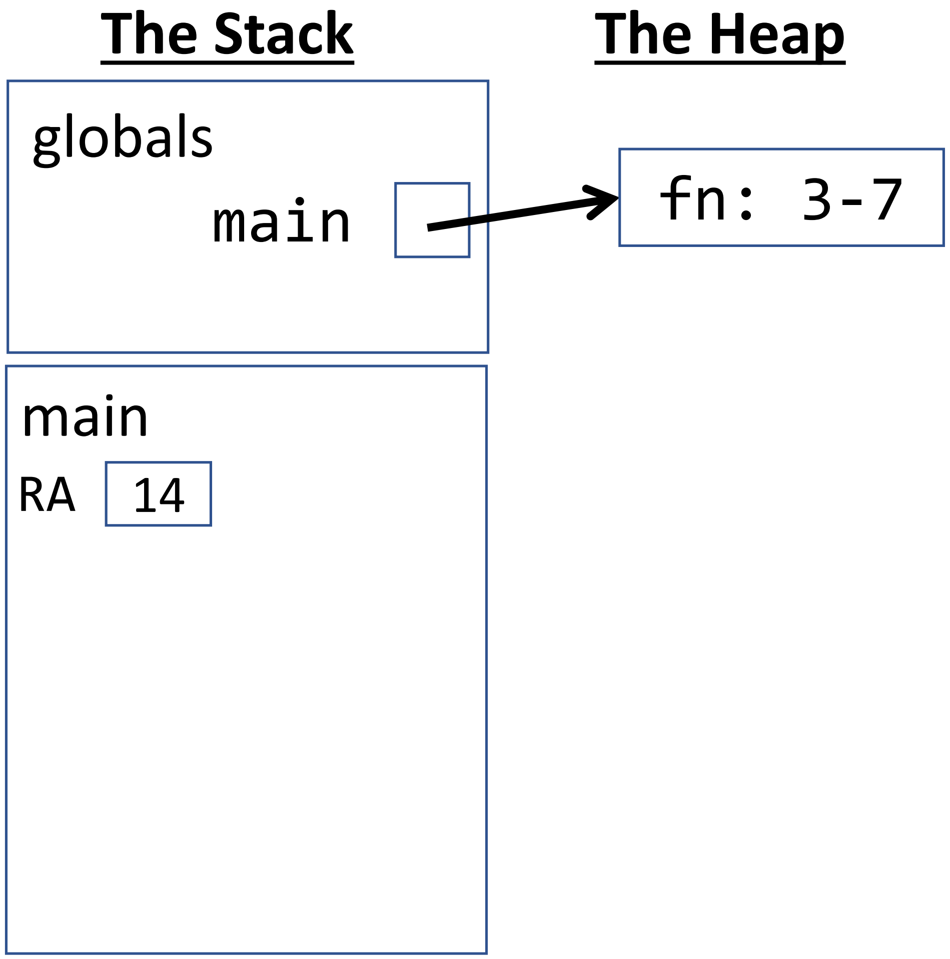
main

fn: 3-7

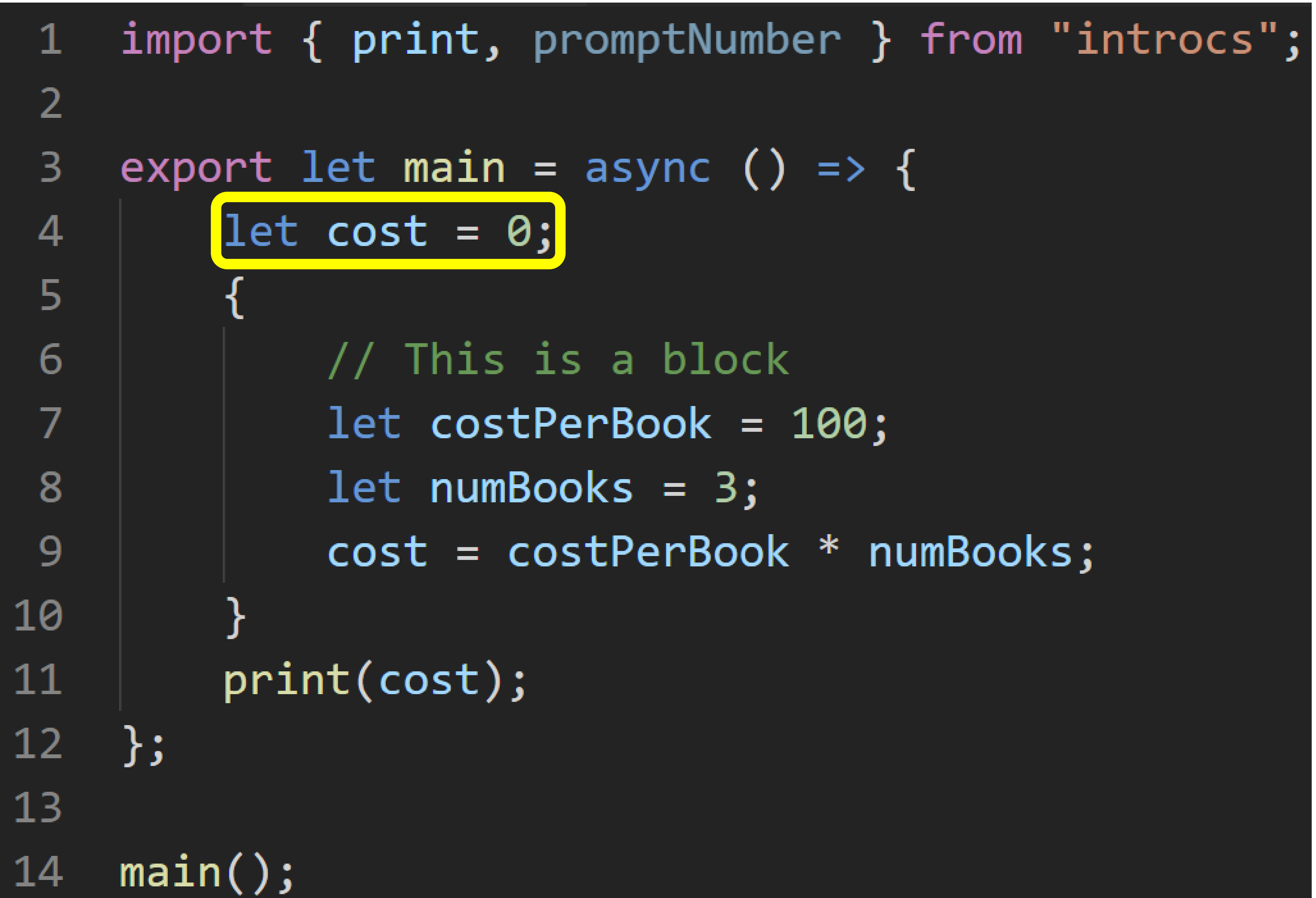
Function Call



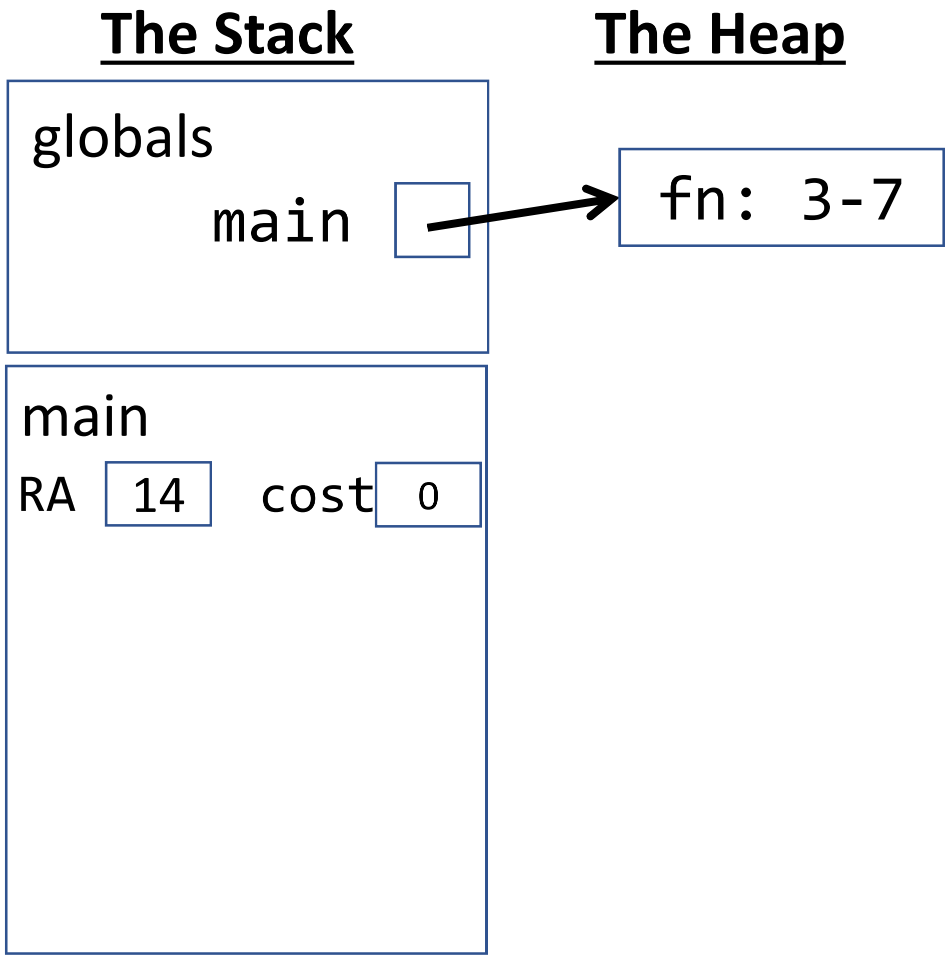
```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



Variable Declaration and Initialization



```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

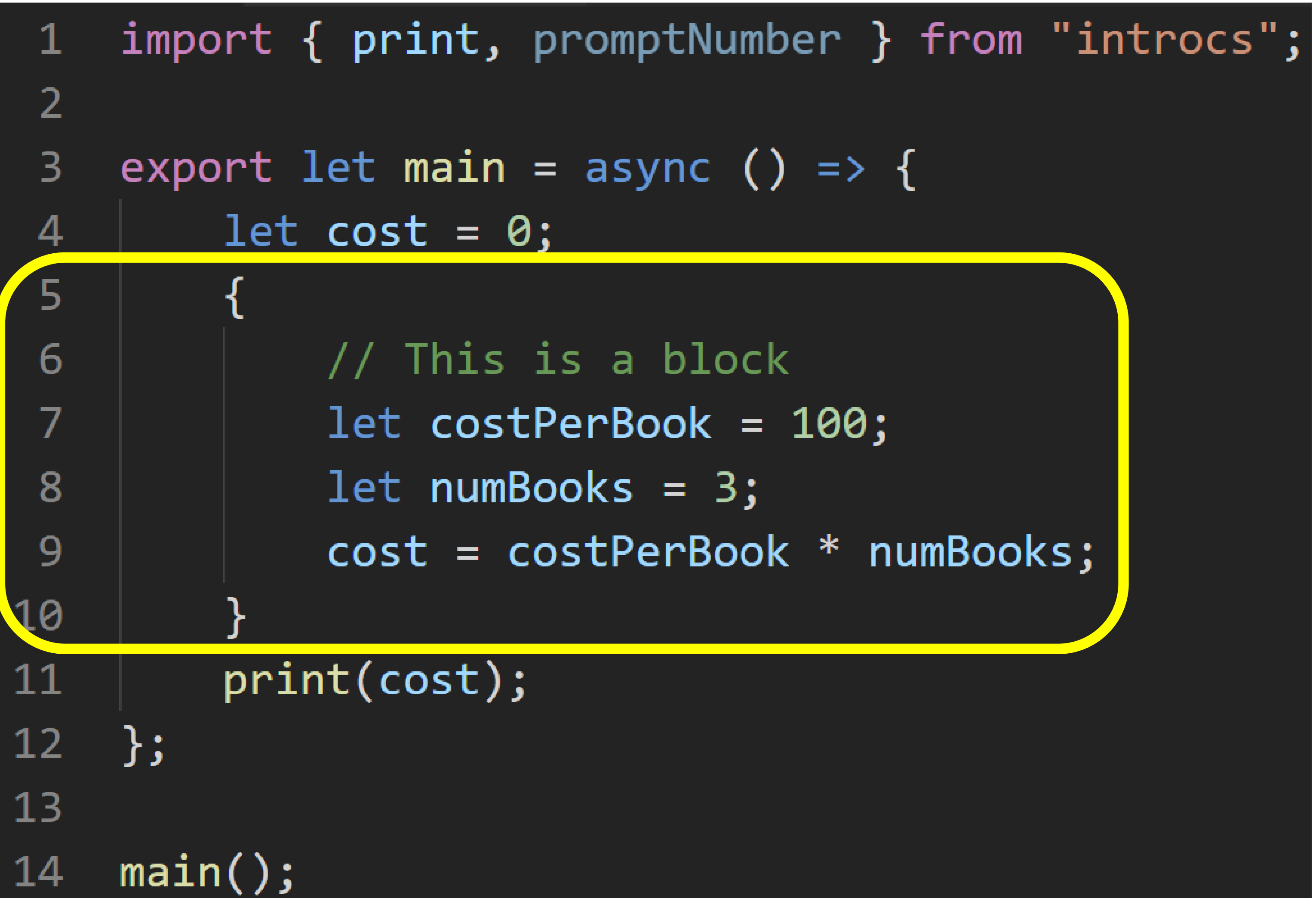
main

RA 14

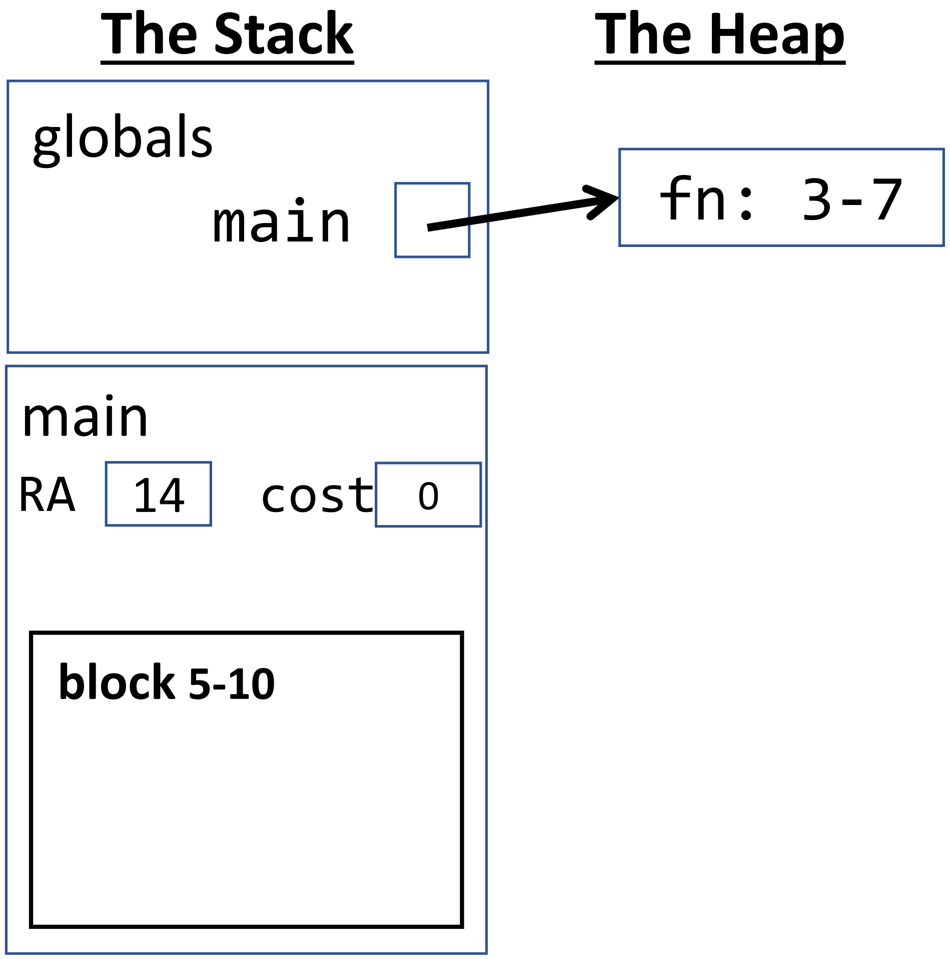
cost 0

Block

When a block with variables declared inside of it is encountered, add a block entry with the start/end lines onto the current frame of the stack.



```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

main

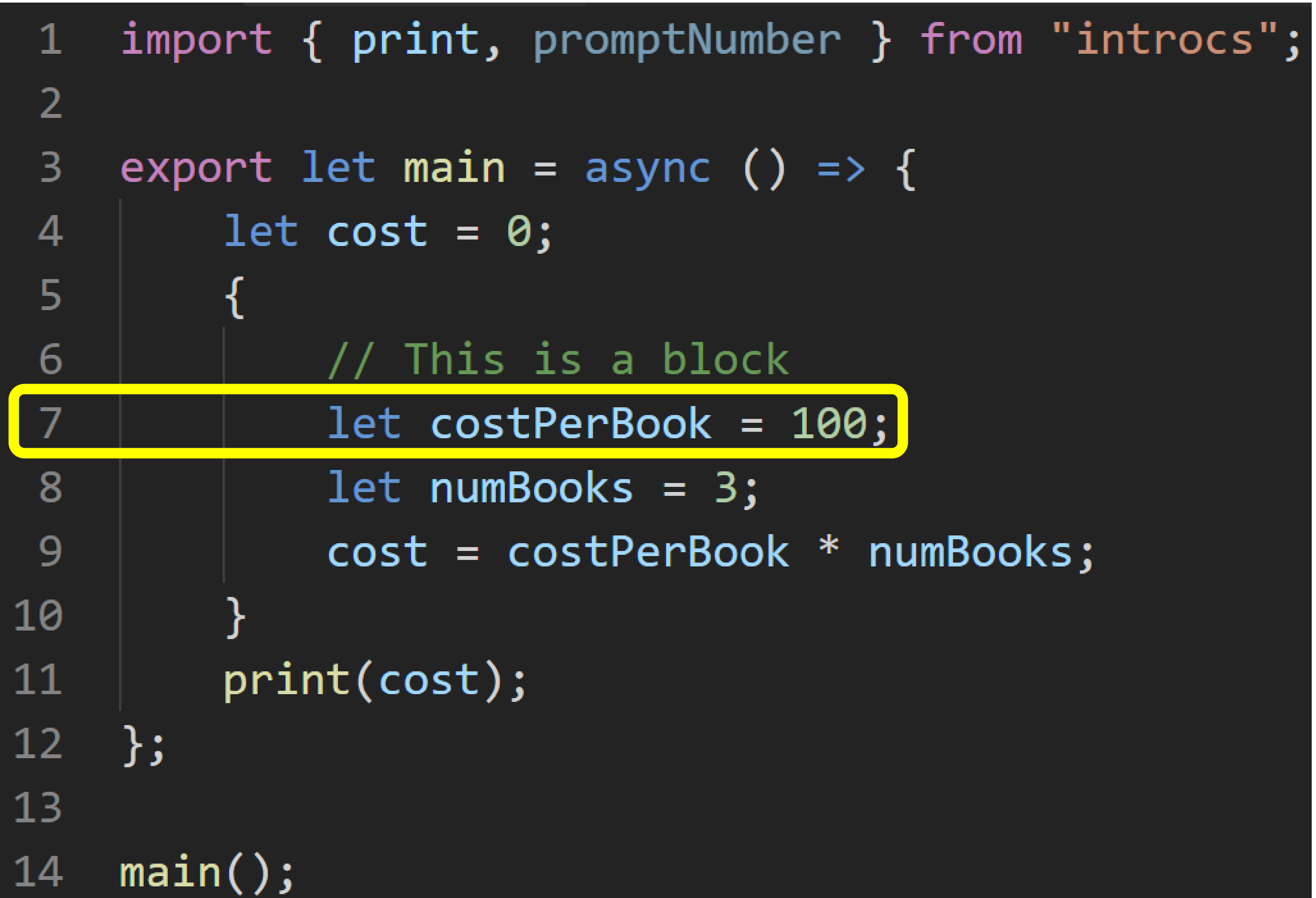
RA 14

cost 0

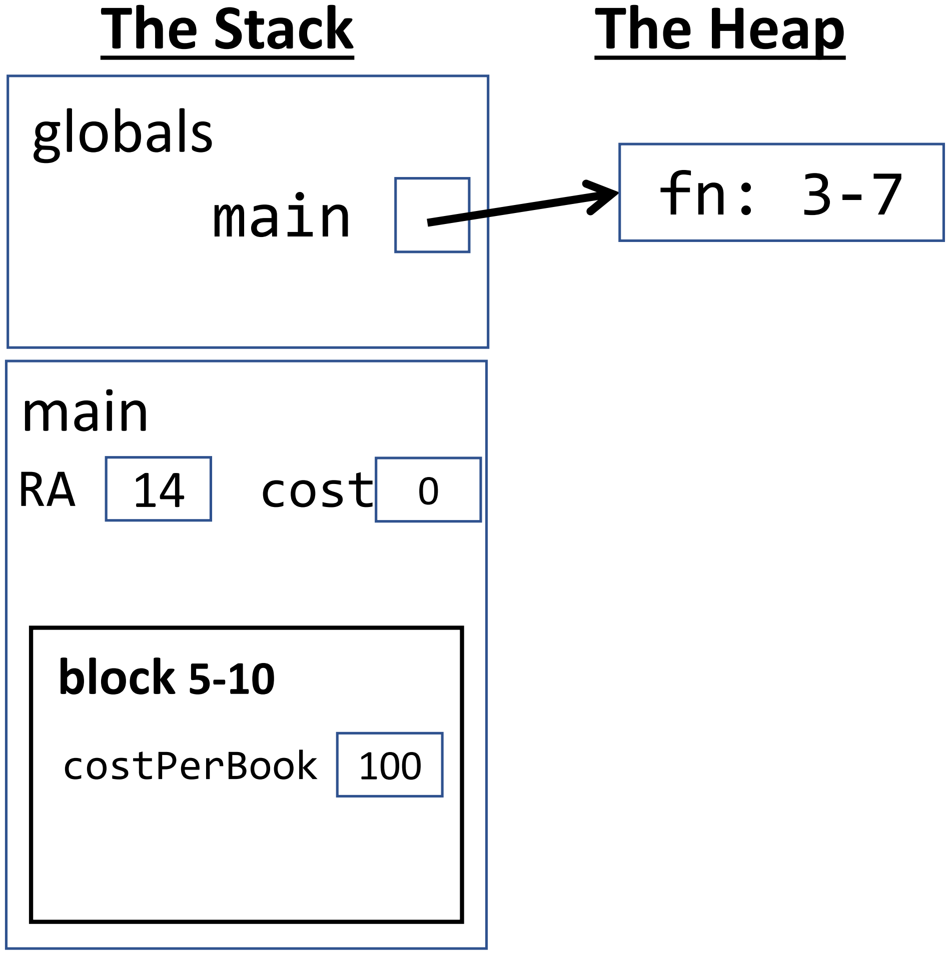
block 5-10

Variable Declaration

Notice the declared variable is established inside of the block's scope.



```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

main

RA 14

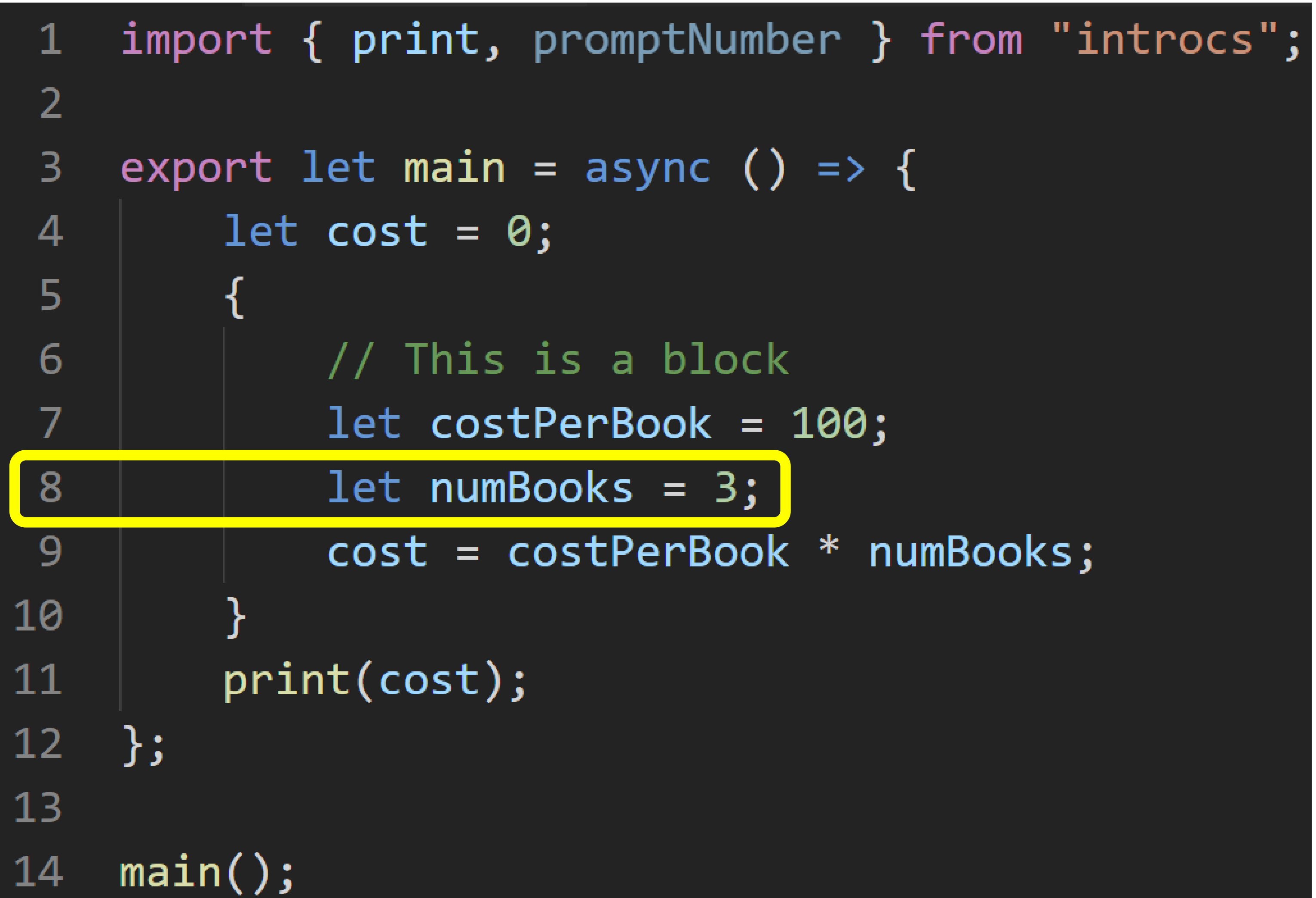
cost 0

block 5-10

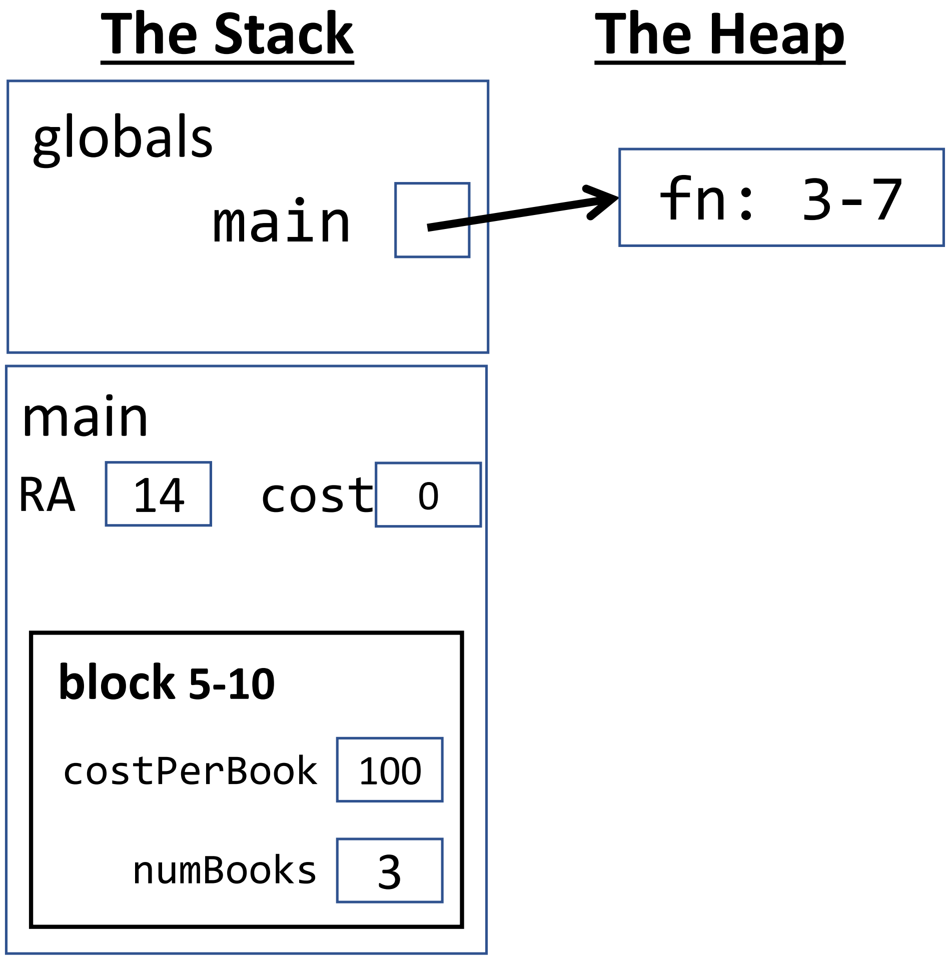
costPerBook 100

Variable Declaration

Notice the declared variable is established inside of the block's scope.



```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

main

RA 14

cost 0

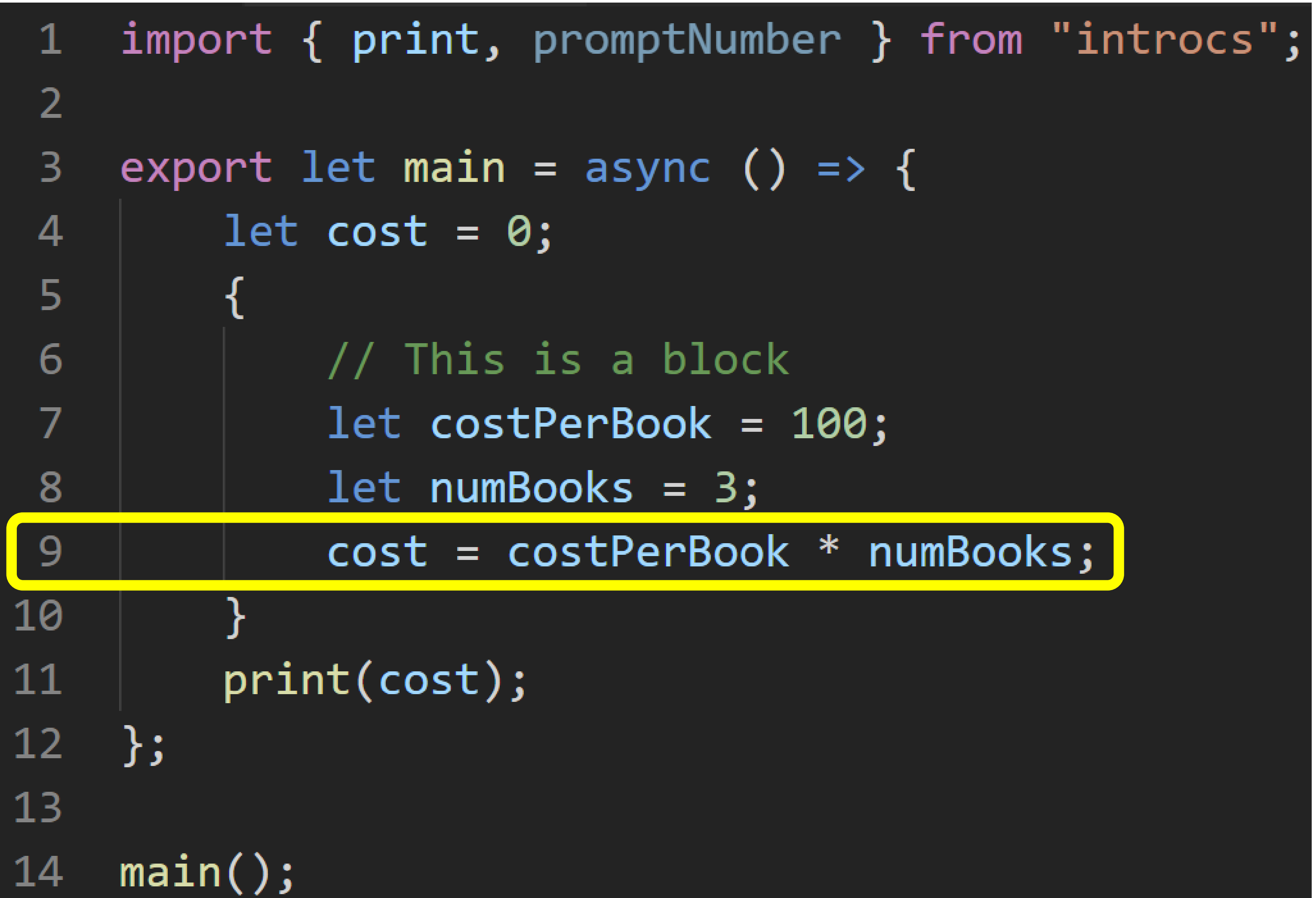
block 5-10

costPerBook 100

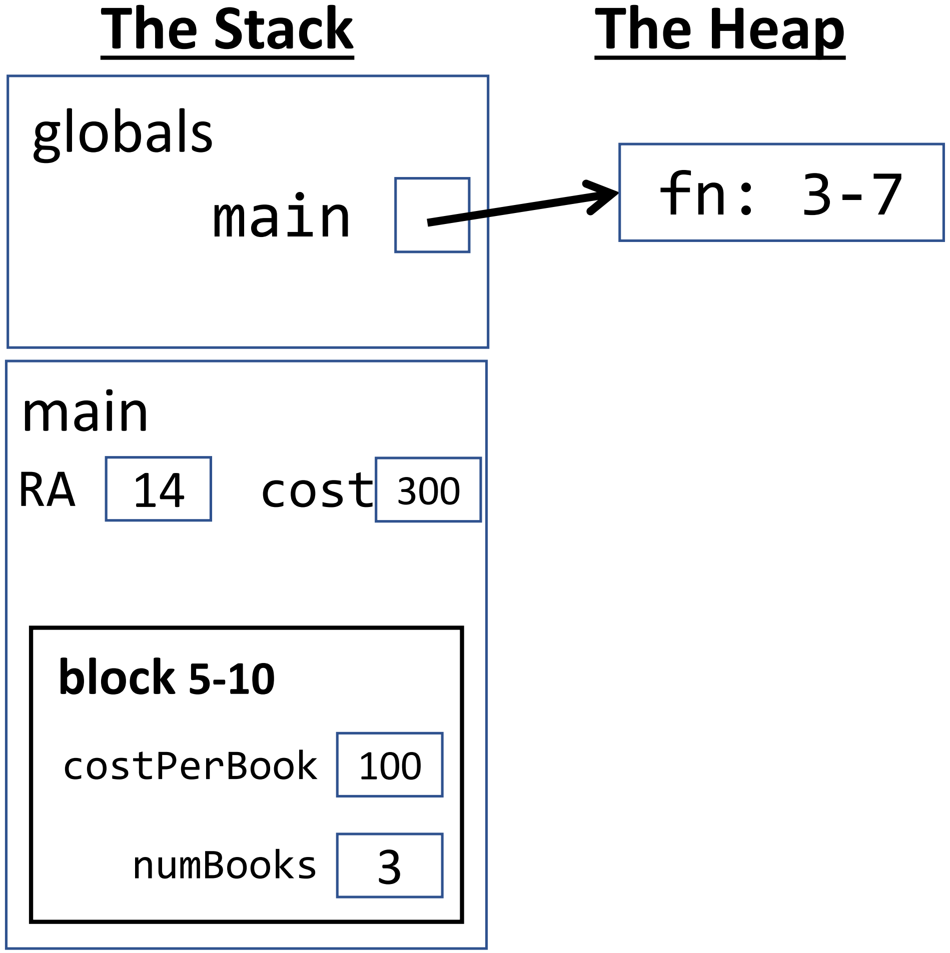
numBooks 3

Name Resolution

Notice the cost variable is declared outside the block. To find a name inside of a block, work your way out toward the surrounding frame, then check globals.



```
 1  import { print, promptNumber } from "introcs";
 2
 3  export let main = async () => {
 4      let cost = 0;
 5      {
 6          // This is a block
 7          let costPerBook = 100;
 8          let numBooks = 3;
 9          cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

main

RA 14

cost 300

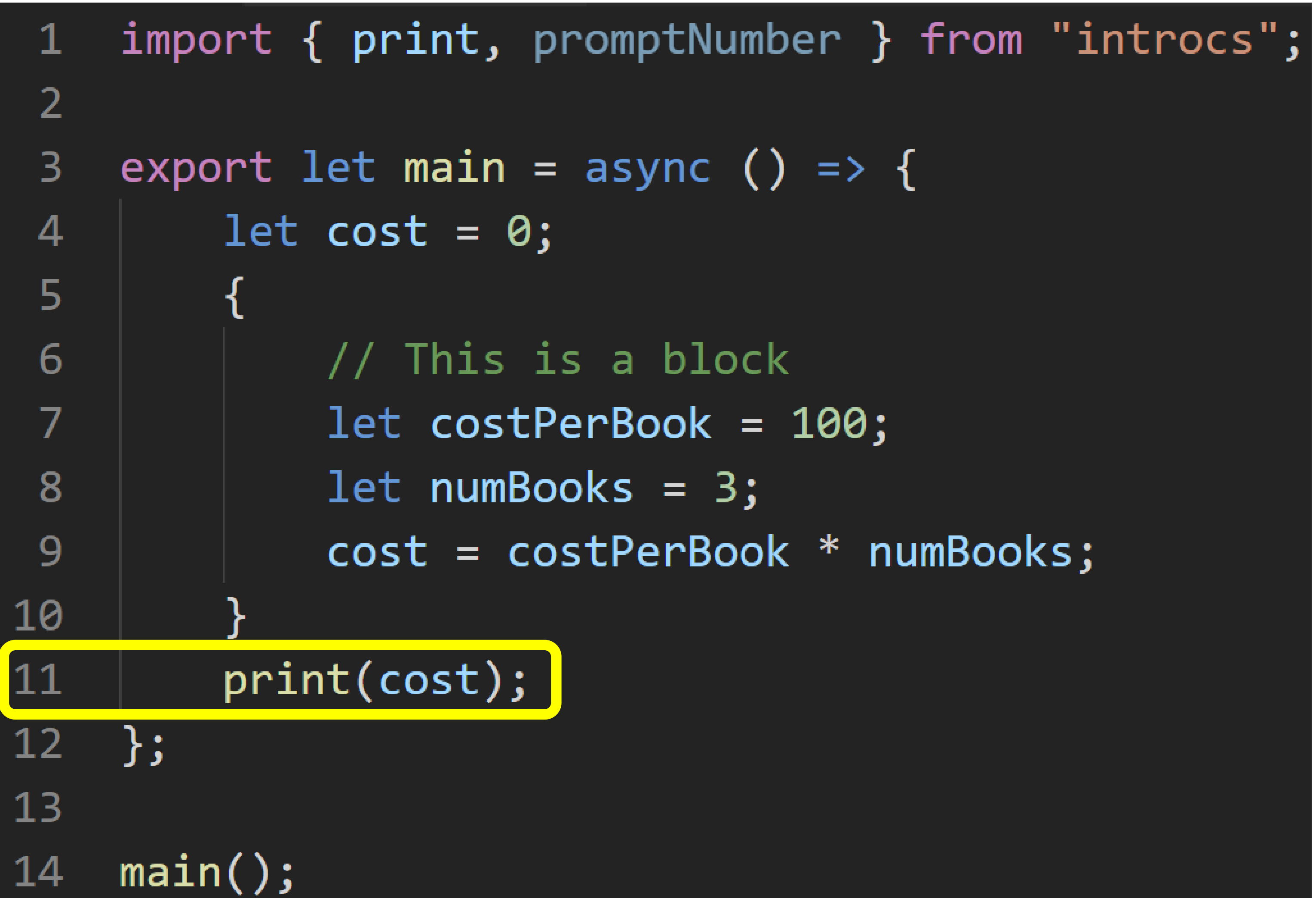
block 5-10

costPerBook 100

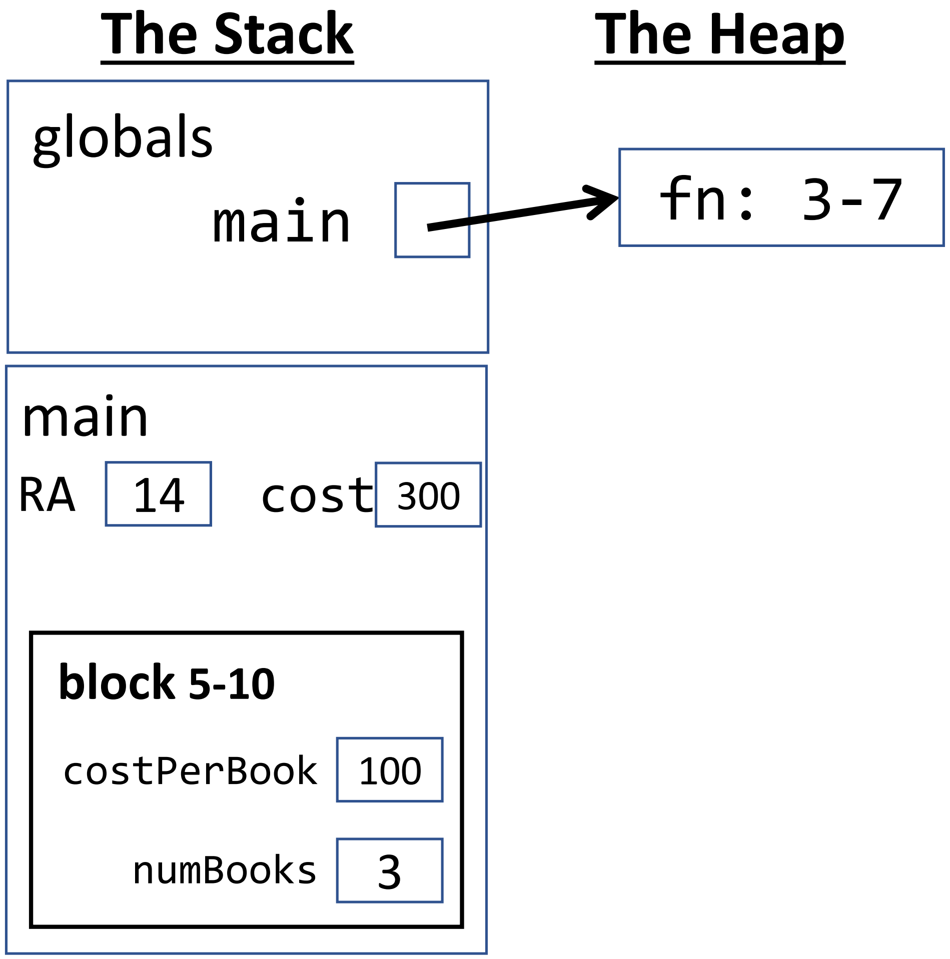
numBooks 3

Name Resolution

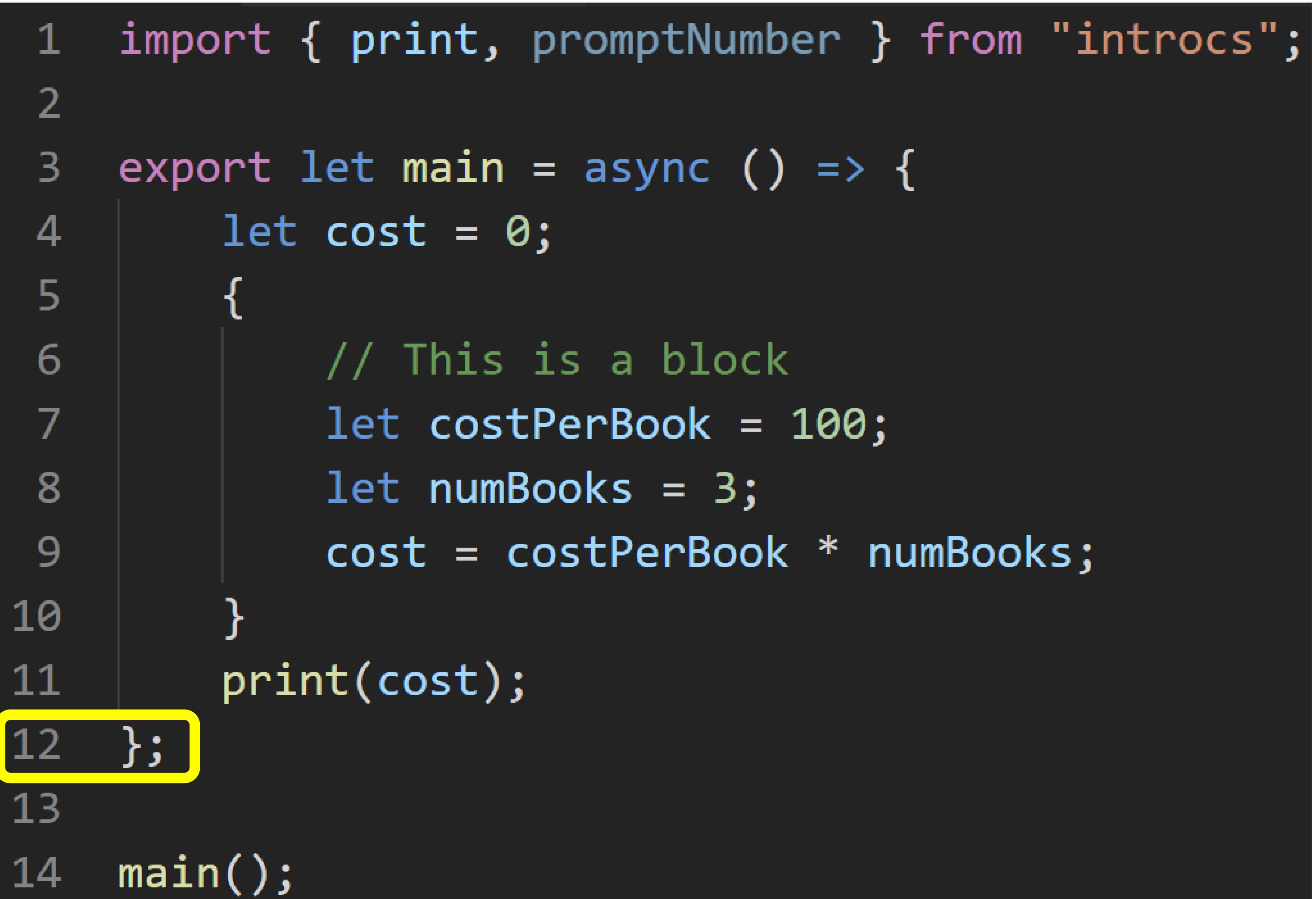
At this point in the program, only the cost variable is accessible in the main frame. Any attempt to print costPerBook or numBooks would error.



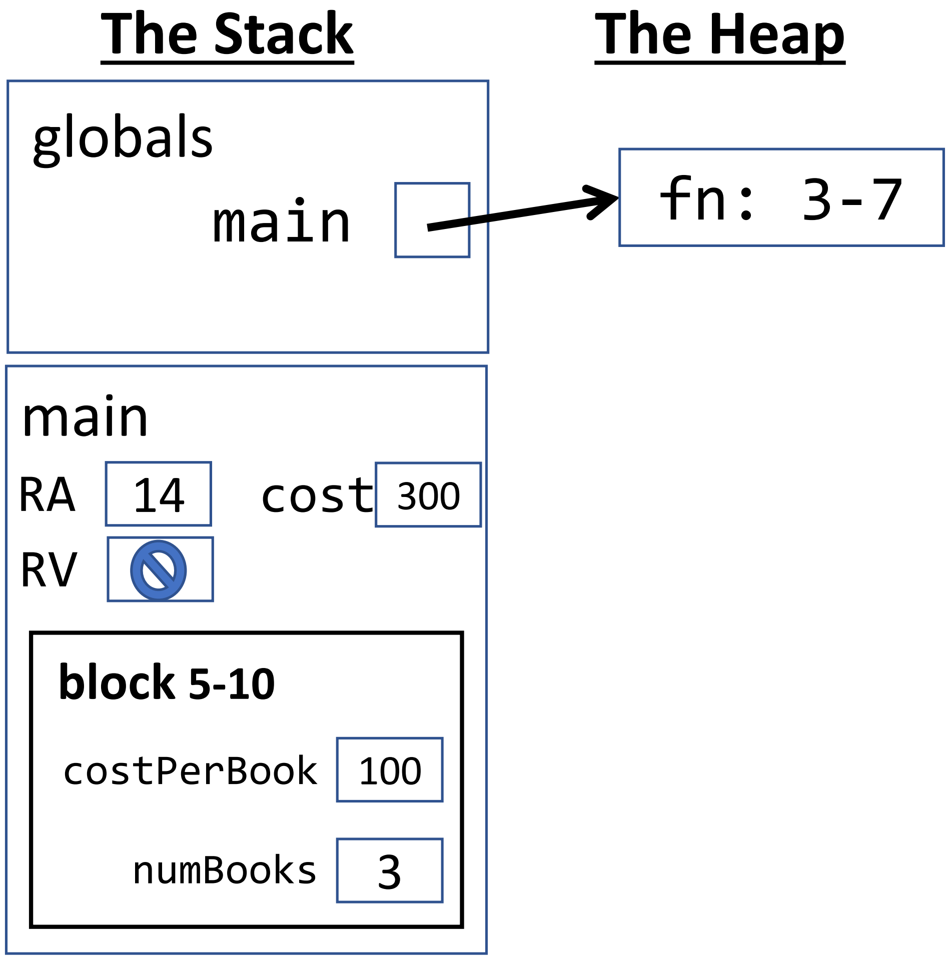
```
1   import { print, promptNumber } from "introcs";
2
3   export let main = async () => {
4       let cost = 0;
5       {
6           // This is a block
7           let costPerBook = 100;
8           let numBooks = 3;
9           cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



Return from **main** Function



```
 1  import { print, promptNumber } from "introcs";
 2
 3  export let main = async () => {
 4      let cost = 0;
 5      {
 6          // This is a block
 7          let costPerBook = 100;
 8          let numBooks = 3;
 9          cost = costPerBook * numBooks;
10      }
11      print(cost);
12  };
13
14  main();
```



The Stack

The Heap

globals

main

fn: 3-7

main

RA 14

cost 300

RV ⃠

block 5-10

costPerBook 100

numBooks 3

Variable Scoping Intuition

- Why have these specific rules?

- Block statements are like building blocks
  - Programs are constructed through many block statements

- Declaring a variable *in a block* prevents unrelated blocks from interference
  - Needing to worry about the existence of unrelated variables
  - Accidentally changing and mucking up another block's variables

- The rules help you avoid accidental logical errors

Warning: Variable **Shadowing (1 / 2)**

- You cannot declare two variables with the same name in the same block.

```
{
   let x = 0;
   let x = 1; // ERROR! x declared prev in same block
}
```

- Why not?
  1. It helps you avoid accidents in longer blocks of code.
     - i.e. you forgot you declared a variable of the same name and used it for another purpose
  2. The *name* of this variable is already reserved in the current stack frame

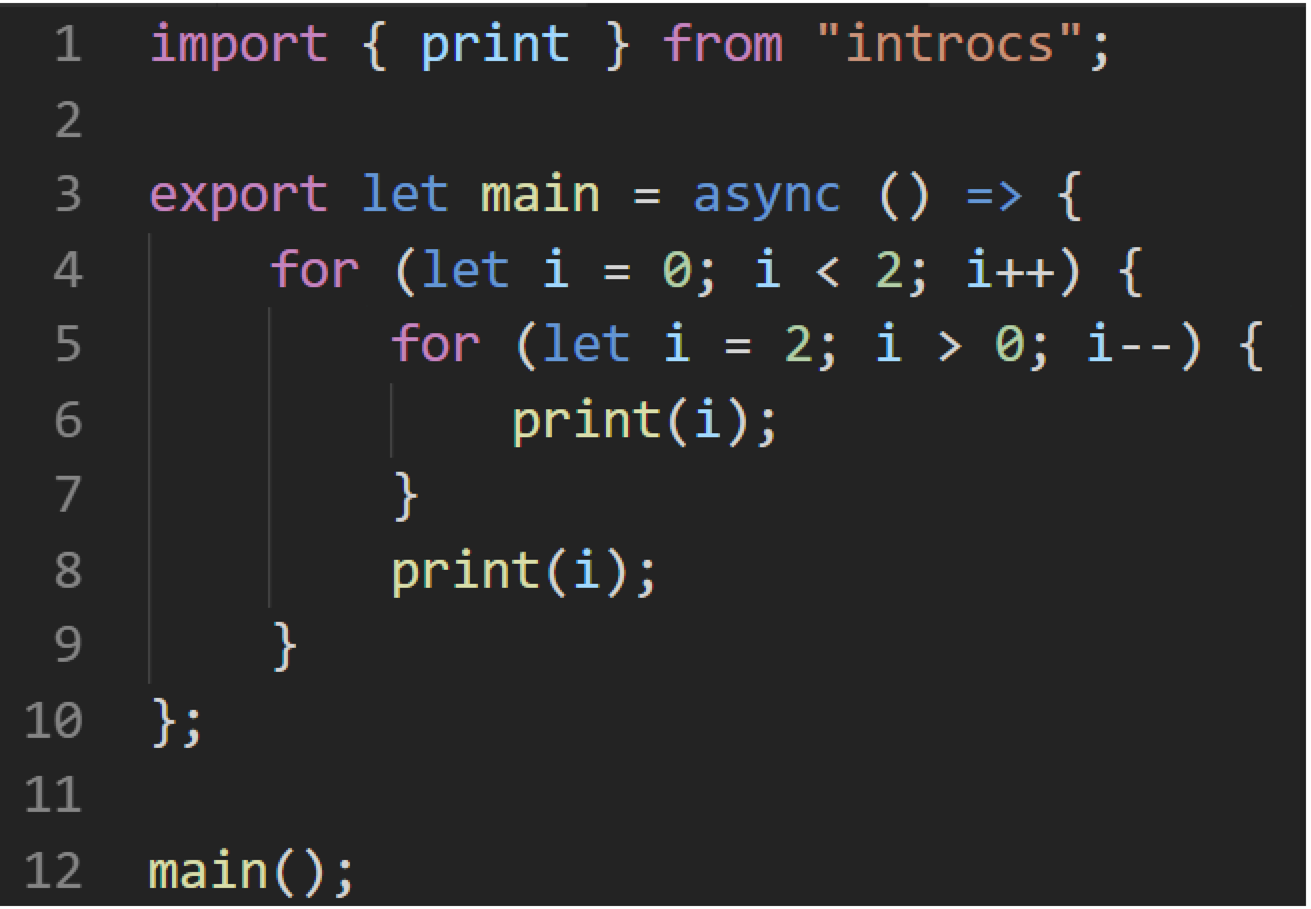
Warning: Variable **Shadowing** (2 / 2)

- You *can* declare a variable of the same name in a nested, inner block. This is called "**variable shadowing**".
- The inner variable is a completely separate variable from the outer variable.
- When the processor returns to the outer block, **x** refers to the original **x** variable and its contents are unchanged.

```
let x = 0;
print(x); // Prints 0
{
  let x = 10;
  print(x); // Prints 10
}
print(x); // Prints 0
```

Variable shadowing is confusing and can usually be avoided by choosing meaningful variable names.

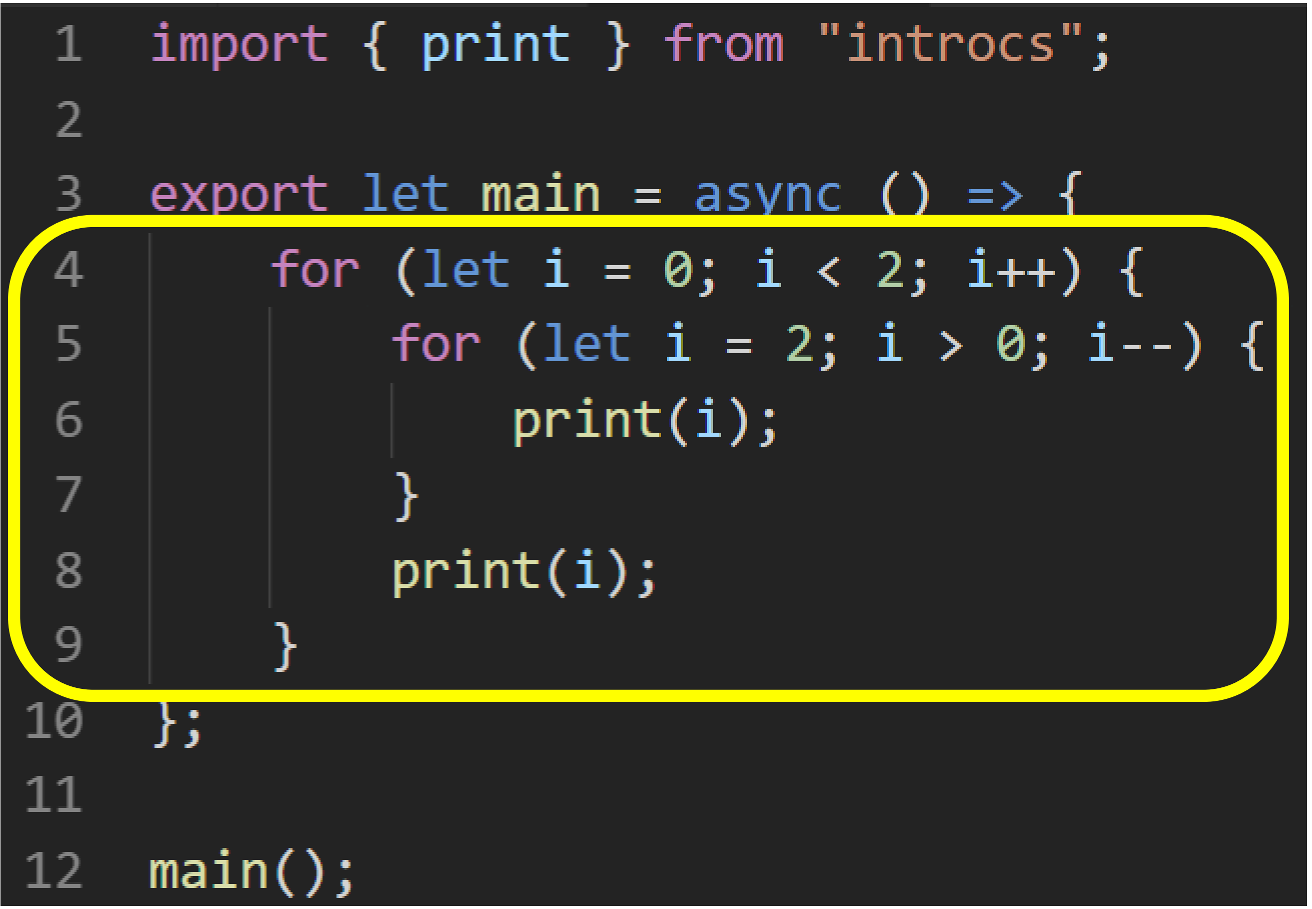
CQ2 – What is the output?



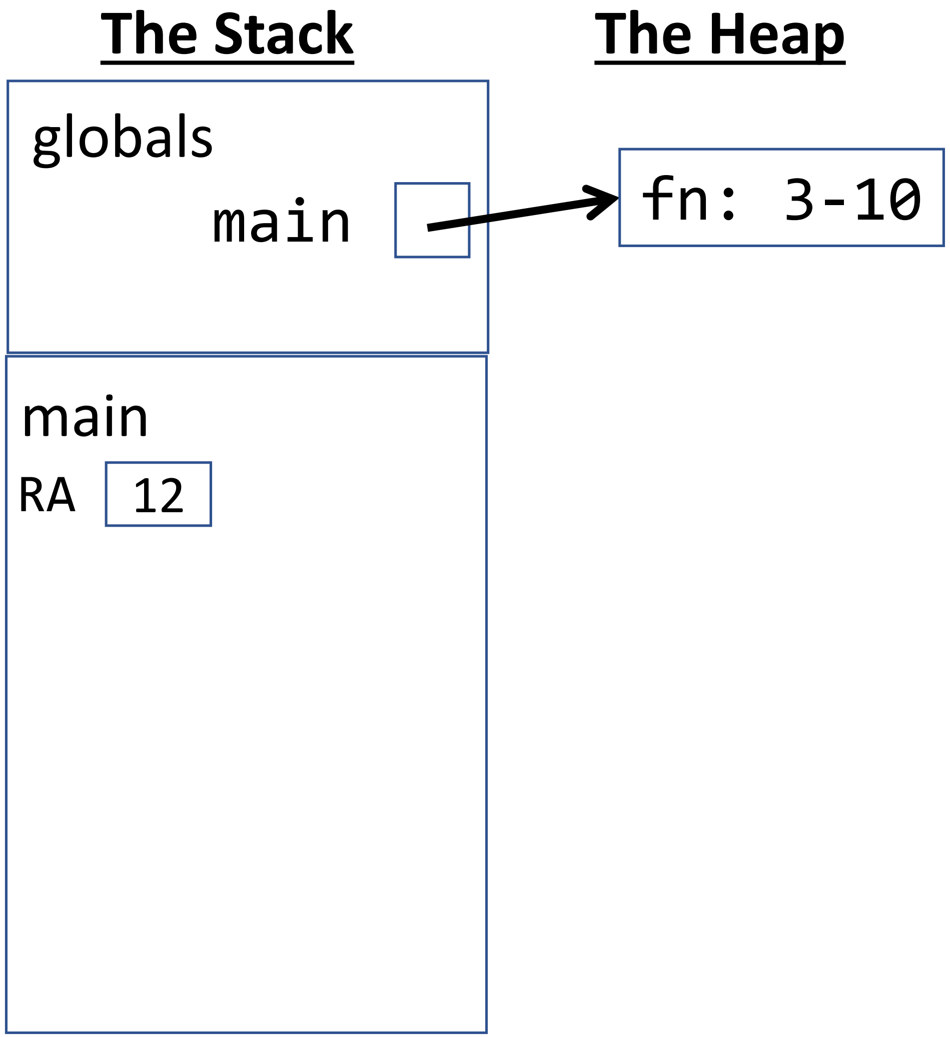
```
import { print } from "introcs";

export let main = async () => {
    for (let i = 0; i < 2; i++) {
        for (let i = 2; i > 0; i--) {
            print(i);
        }
        print(i);
    }
};

main();
```

Fast-Forward

Imagine execution has reached the point of encountering this for loop.

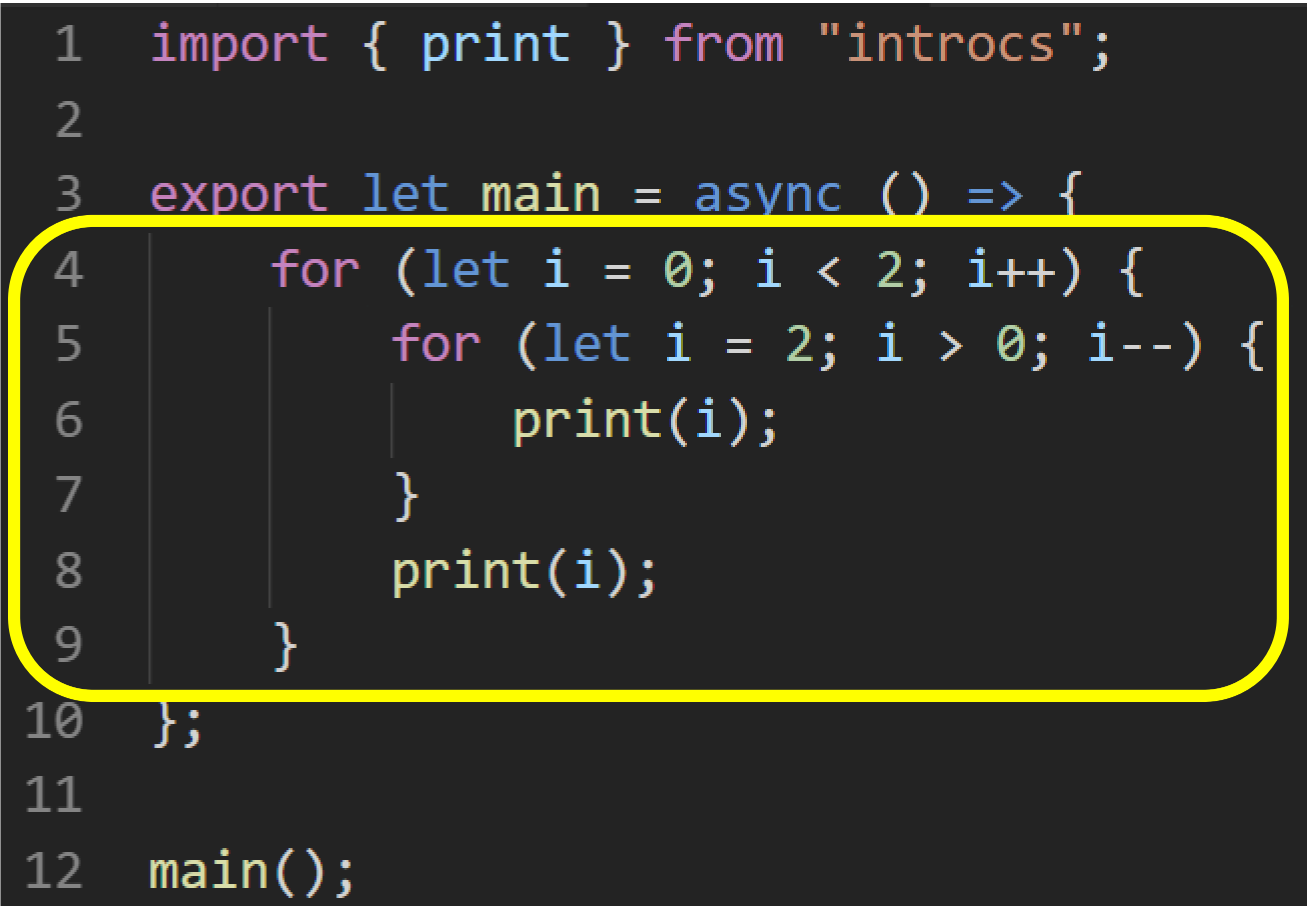


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

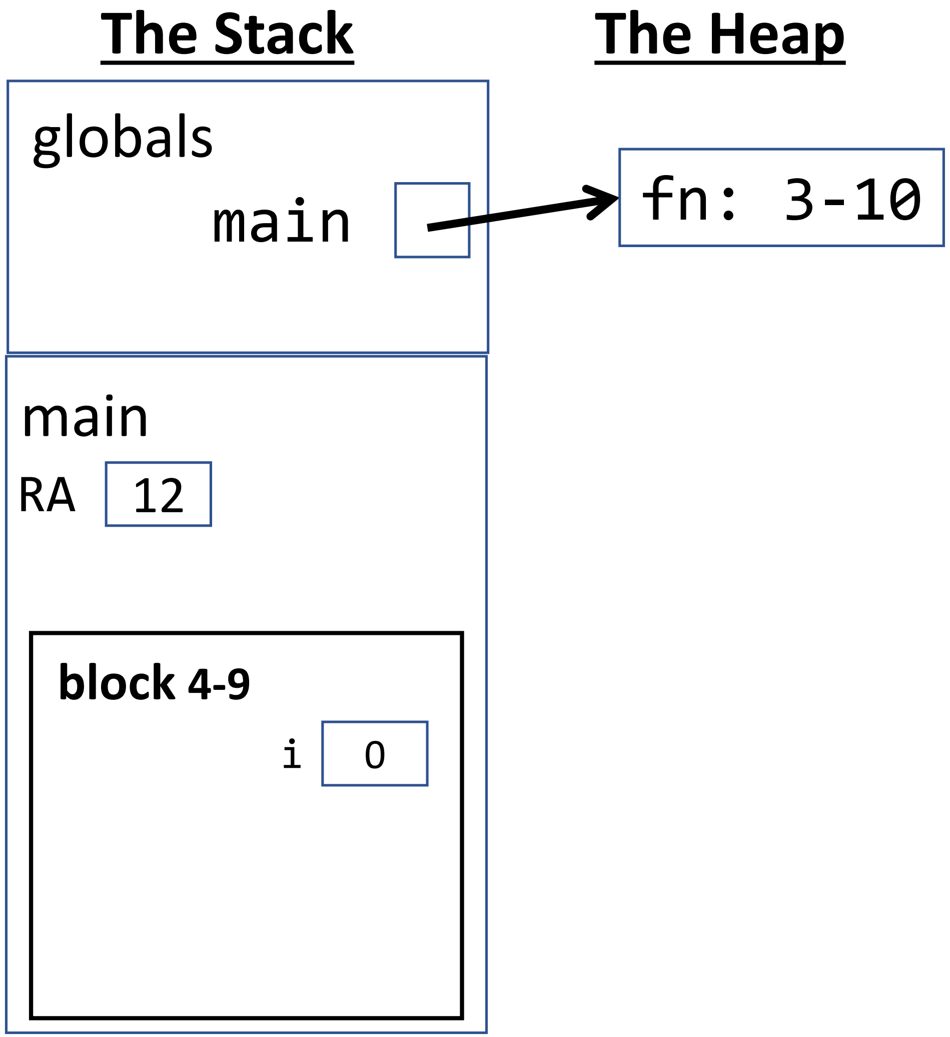


For Loop Block

A block is established to hold the counter variable of the *for* loop.

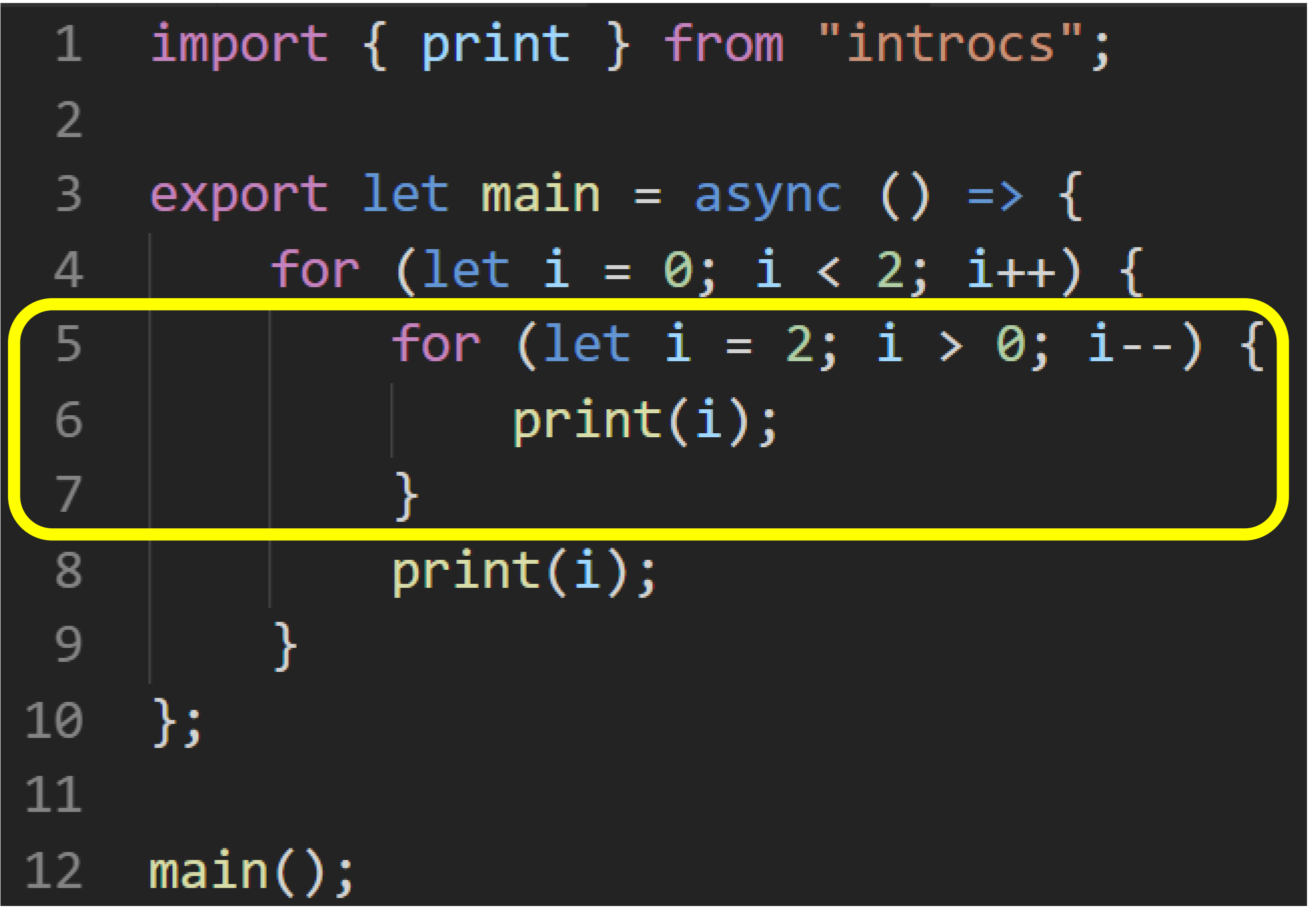


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

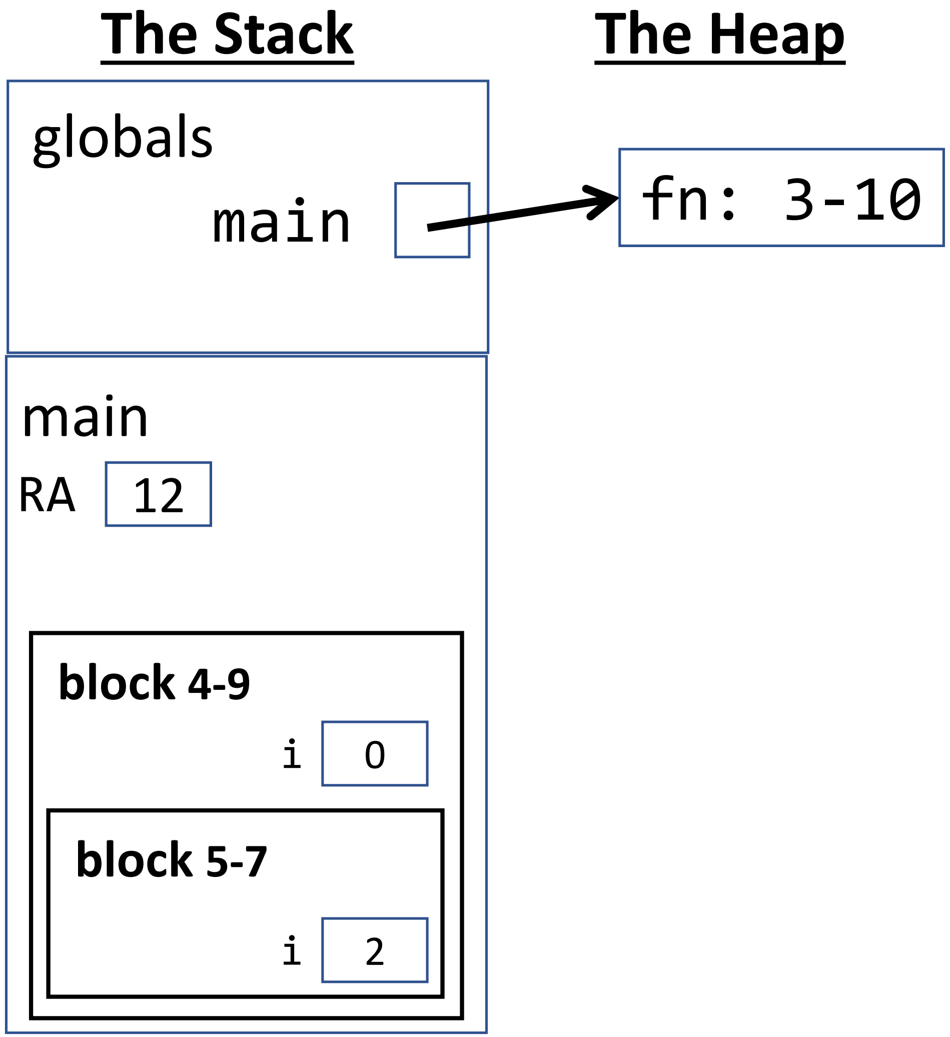


For Loop Block

Another block is established inside the current block to hold the counter variable of the nested *for* loop. Notice since each loop has its own block they're separate.

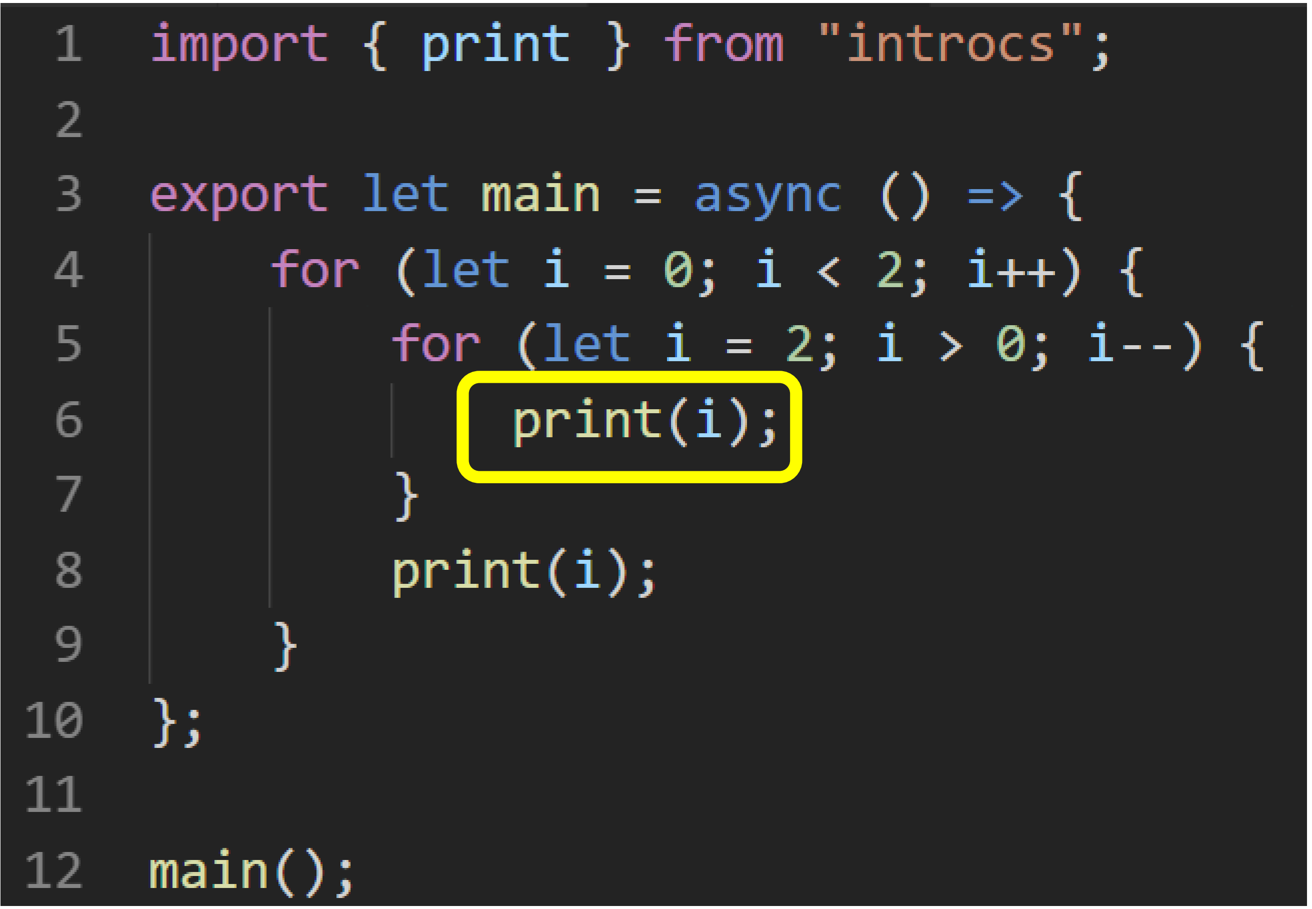


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

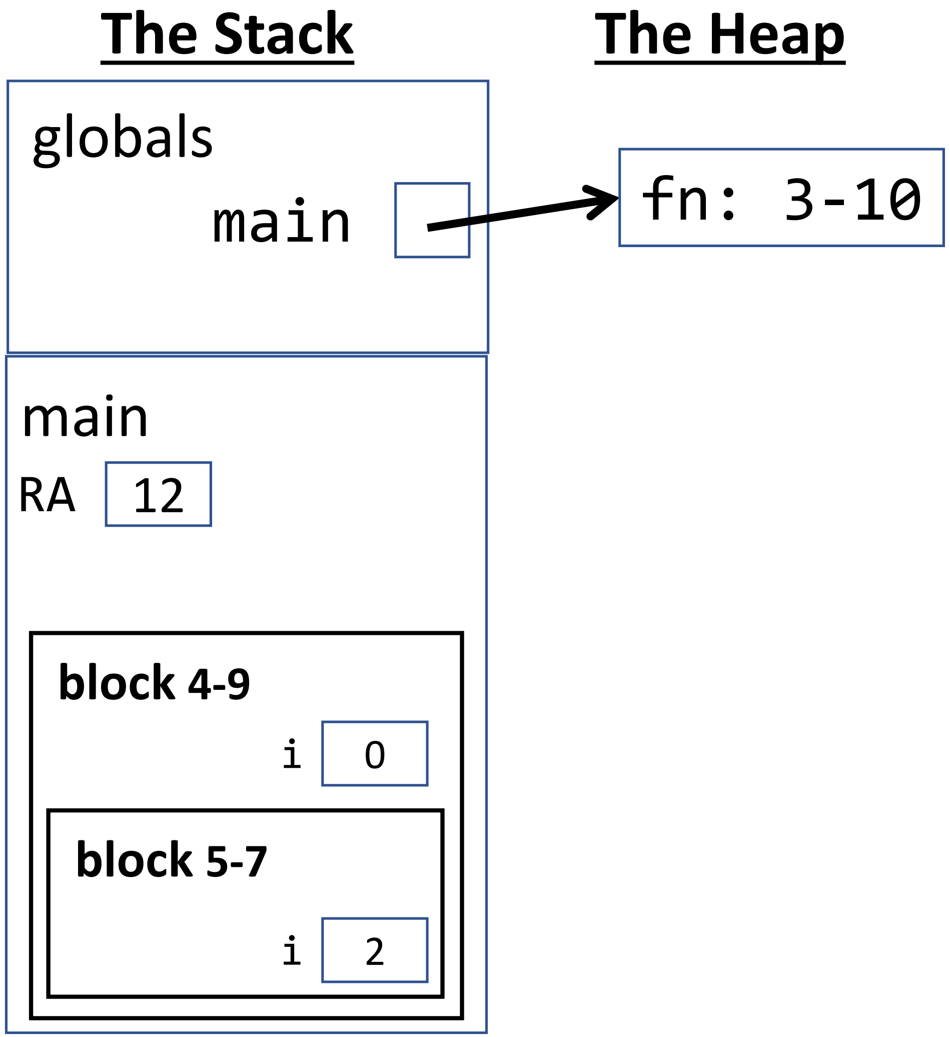


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      for (let i = 0; i < 2; i++) {
 5          for (let i = 2; i > 0; i--) {
 6              print(i);
 7          }
 8          print(i);
 9      }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

fn: 3-10

main

RA 12

block 4-9

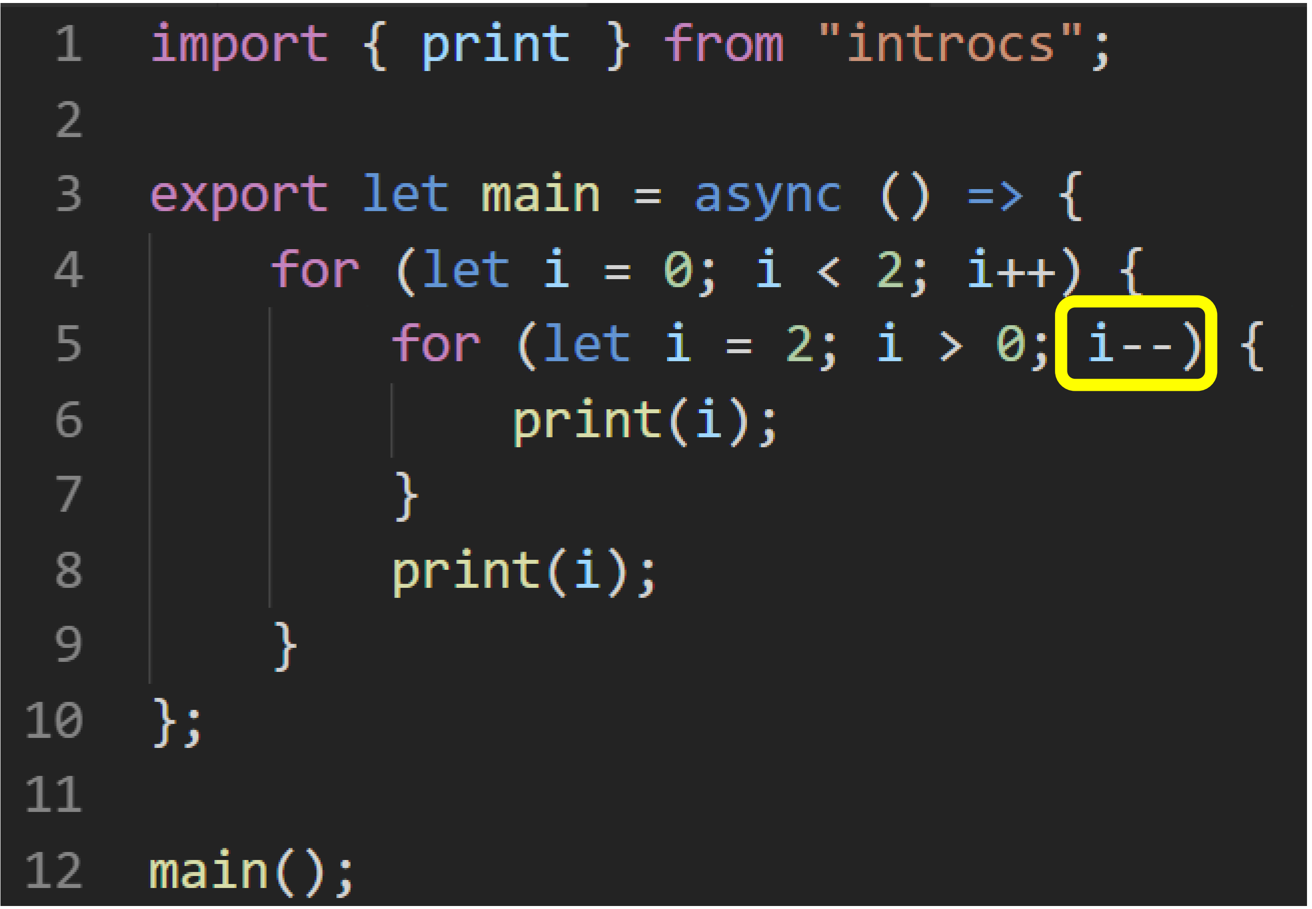
i 0

block 5-7

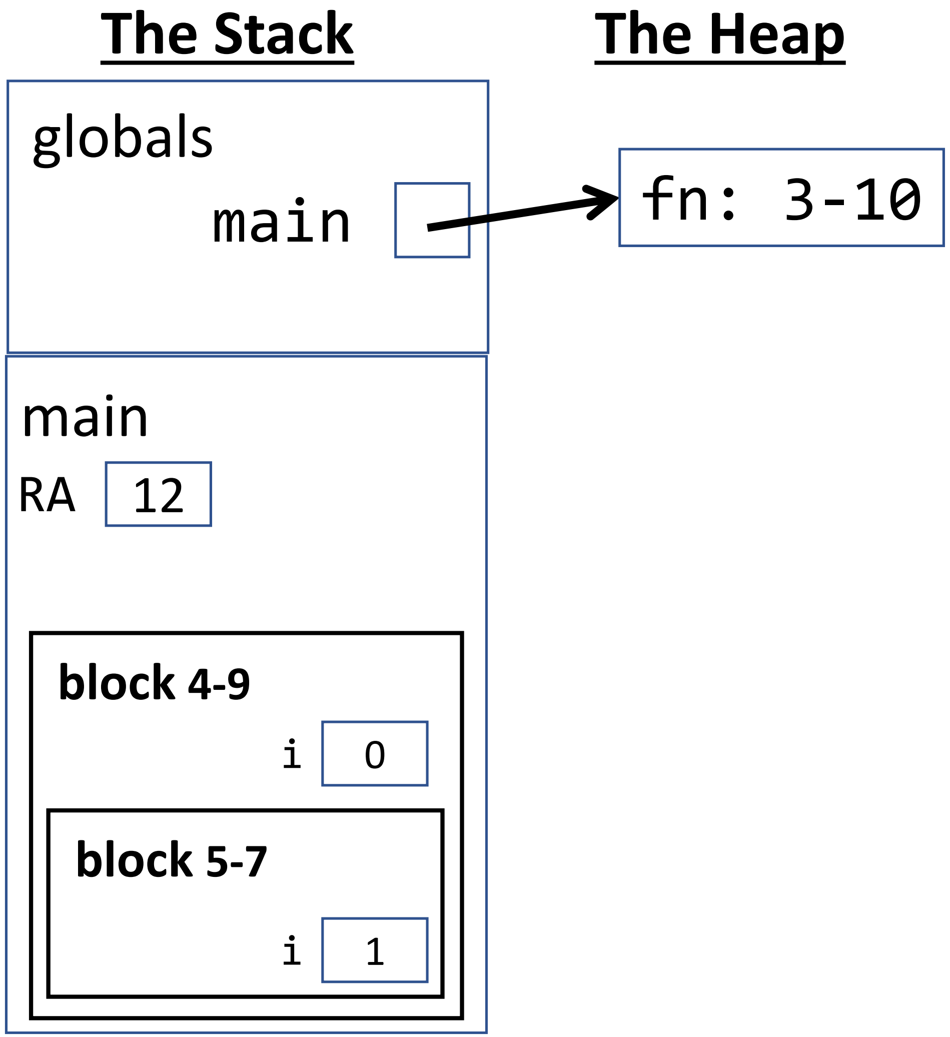
i 2

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

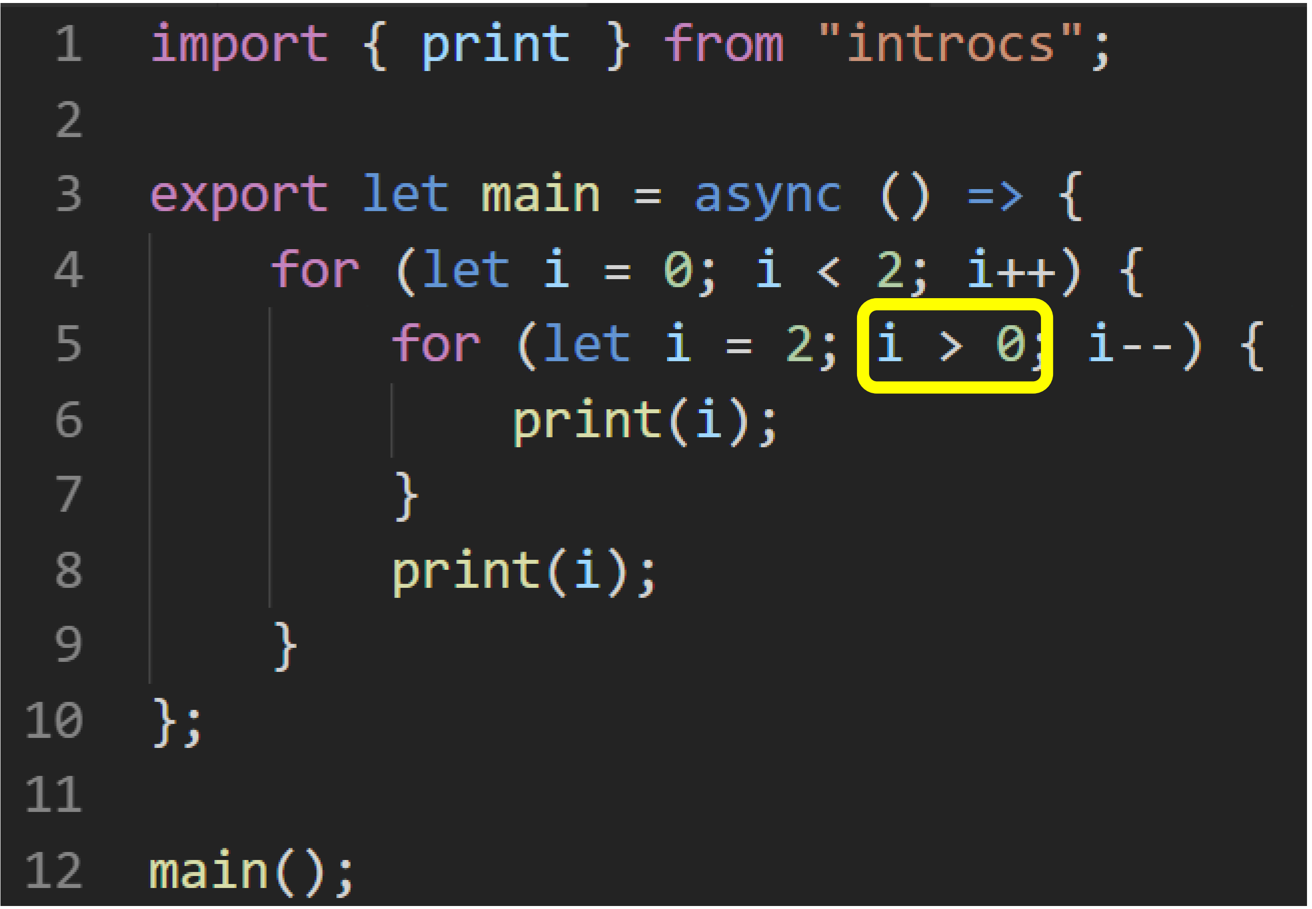


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

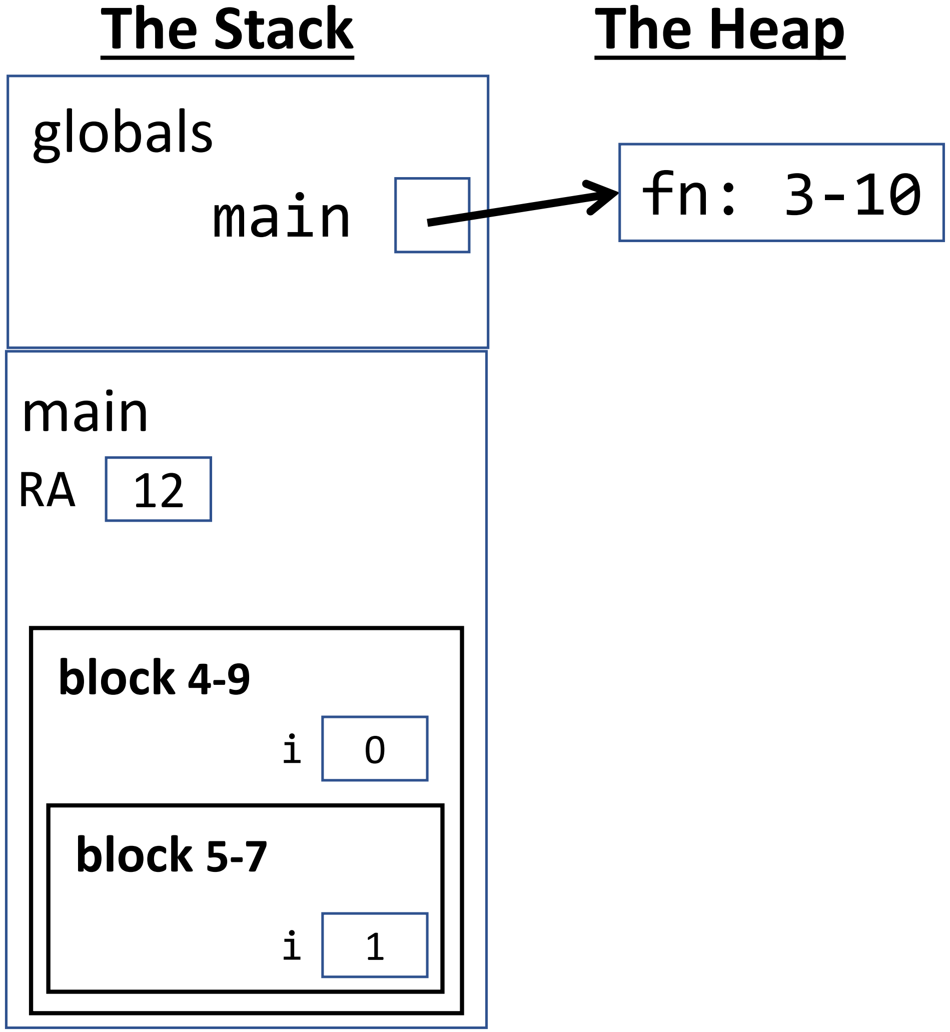


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

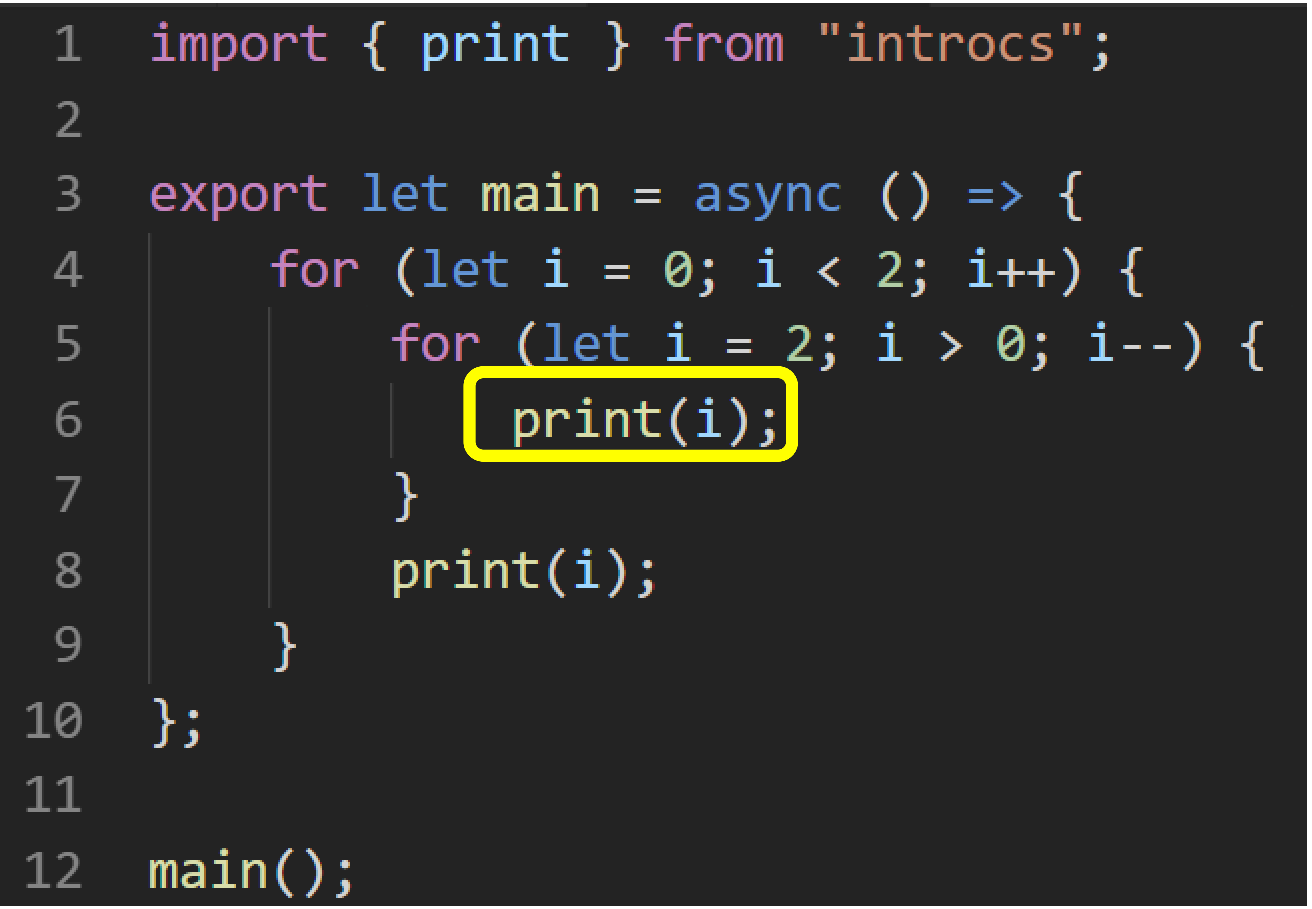


```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      for (let i = 0; i < 2; i++) {
 5          for (let i = 2; i > 0; i--) {
 6              print(i);
 7          }
 8          print(i);
 9      }
10  };
11
12  main();
```

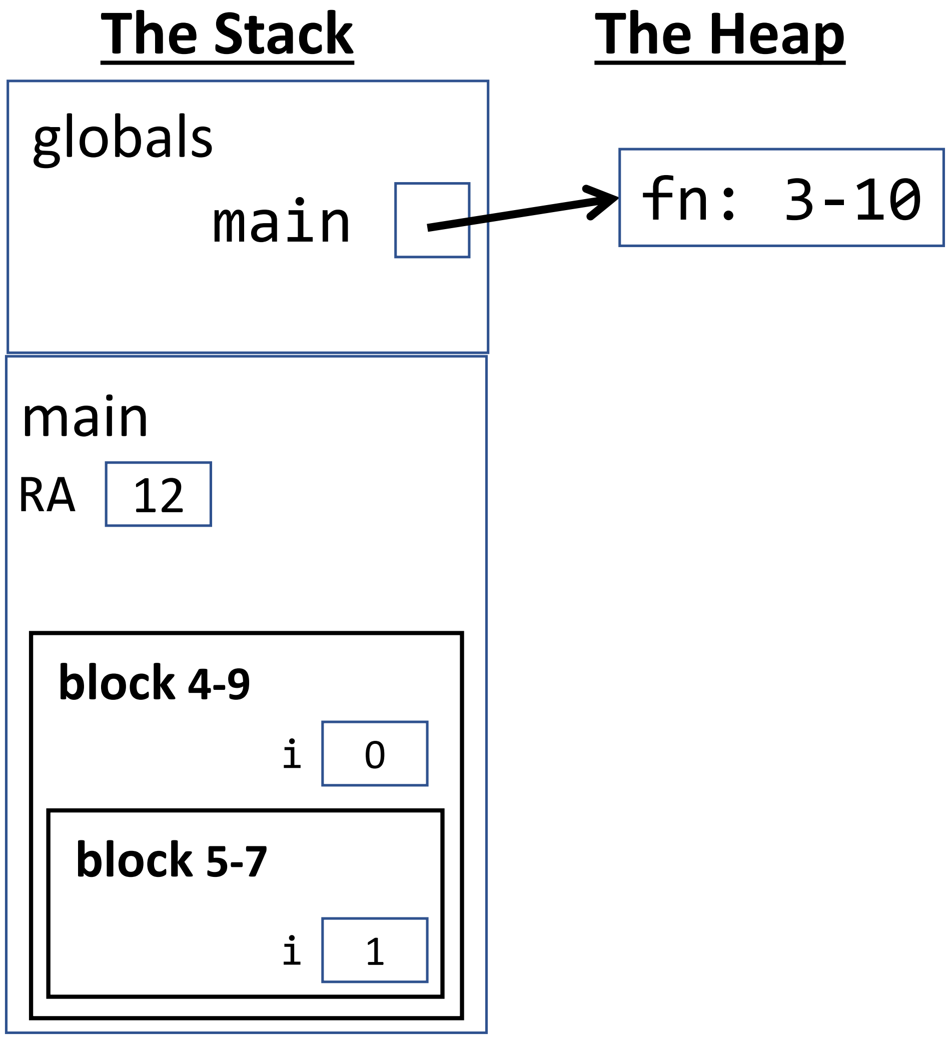


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



The Stack

The Heap

globals

main

fn: 3-10

main

RA 12

block 4-9

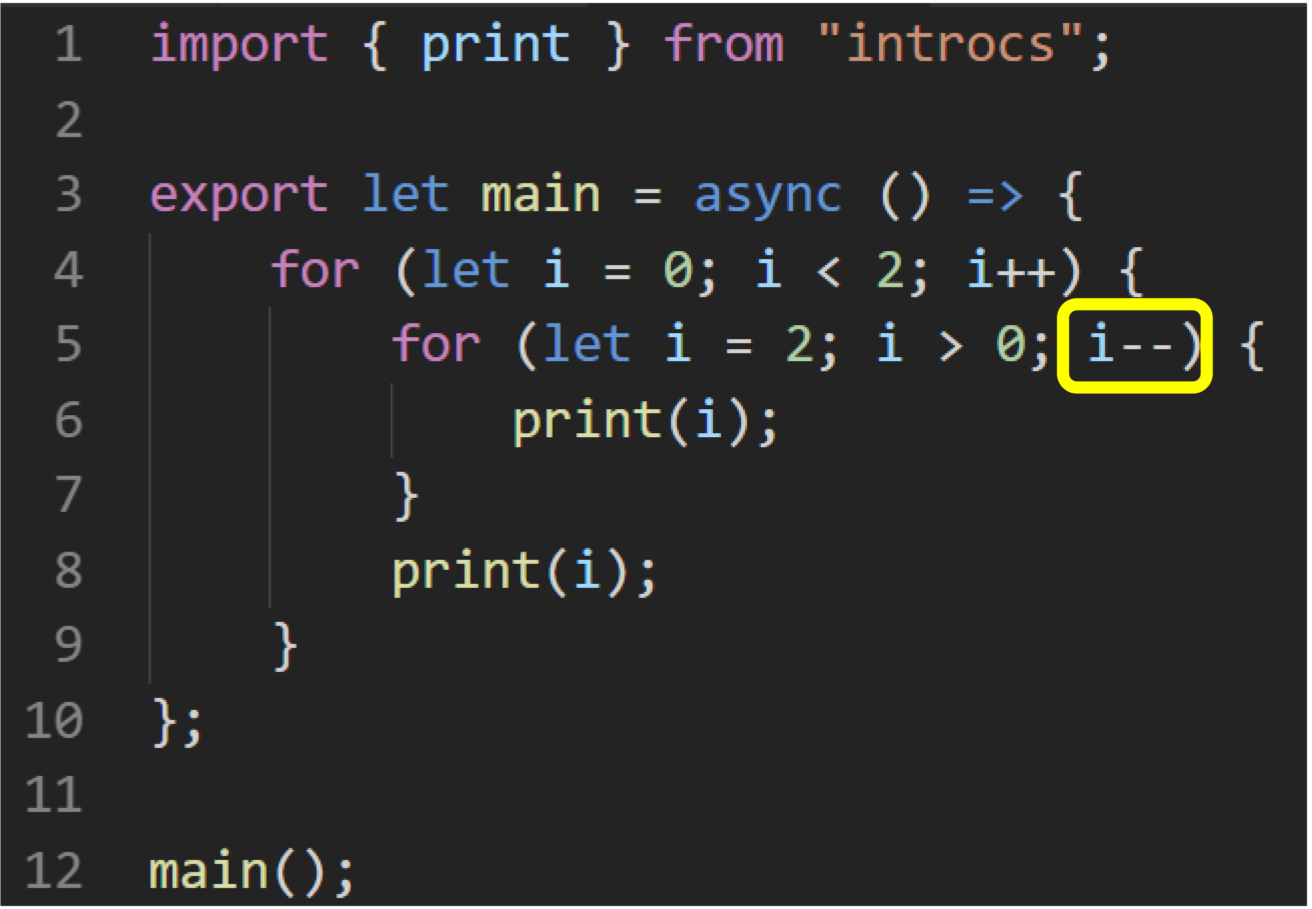
i 0

block 5-7

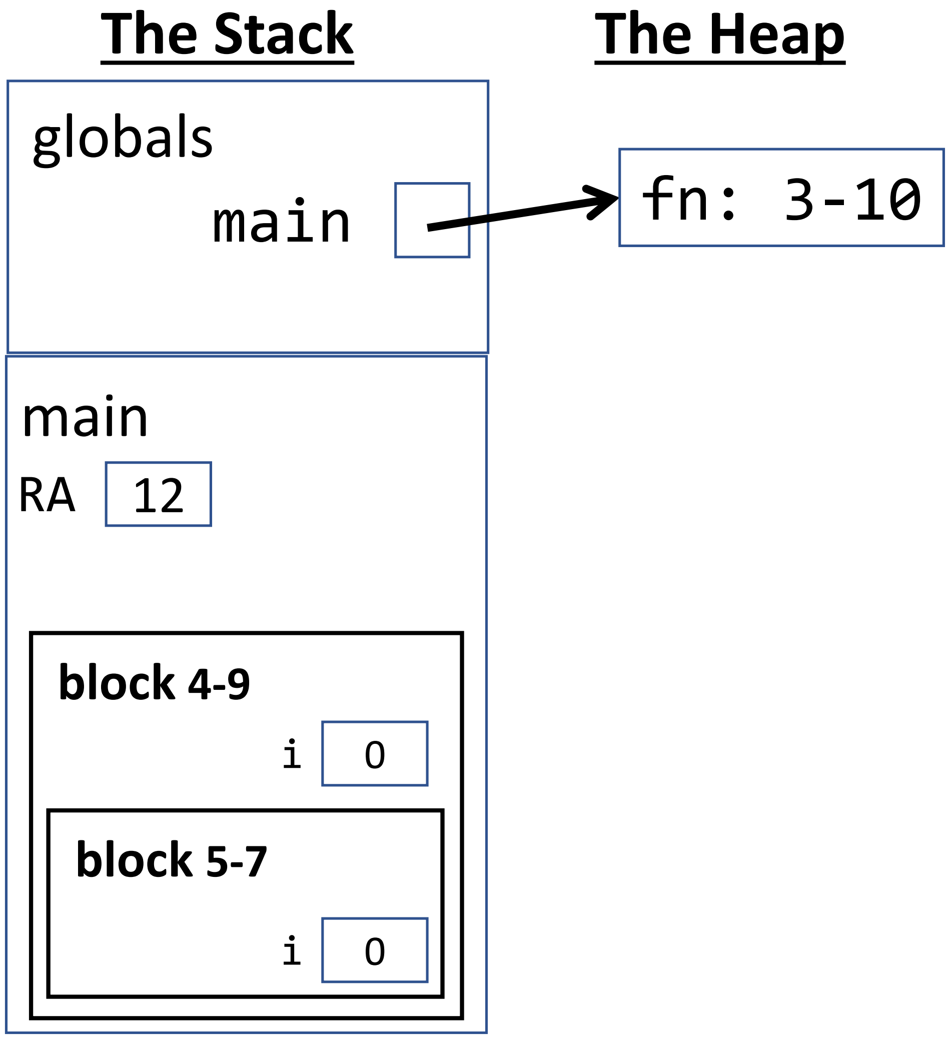
i 1

Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

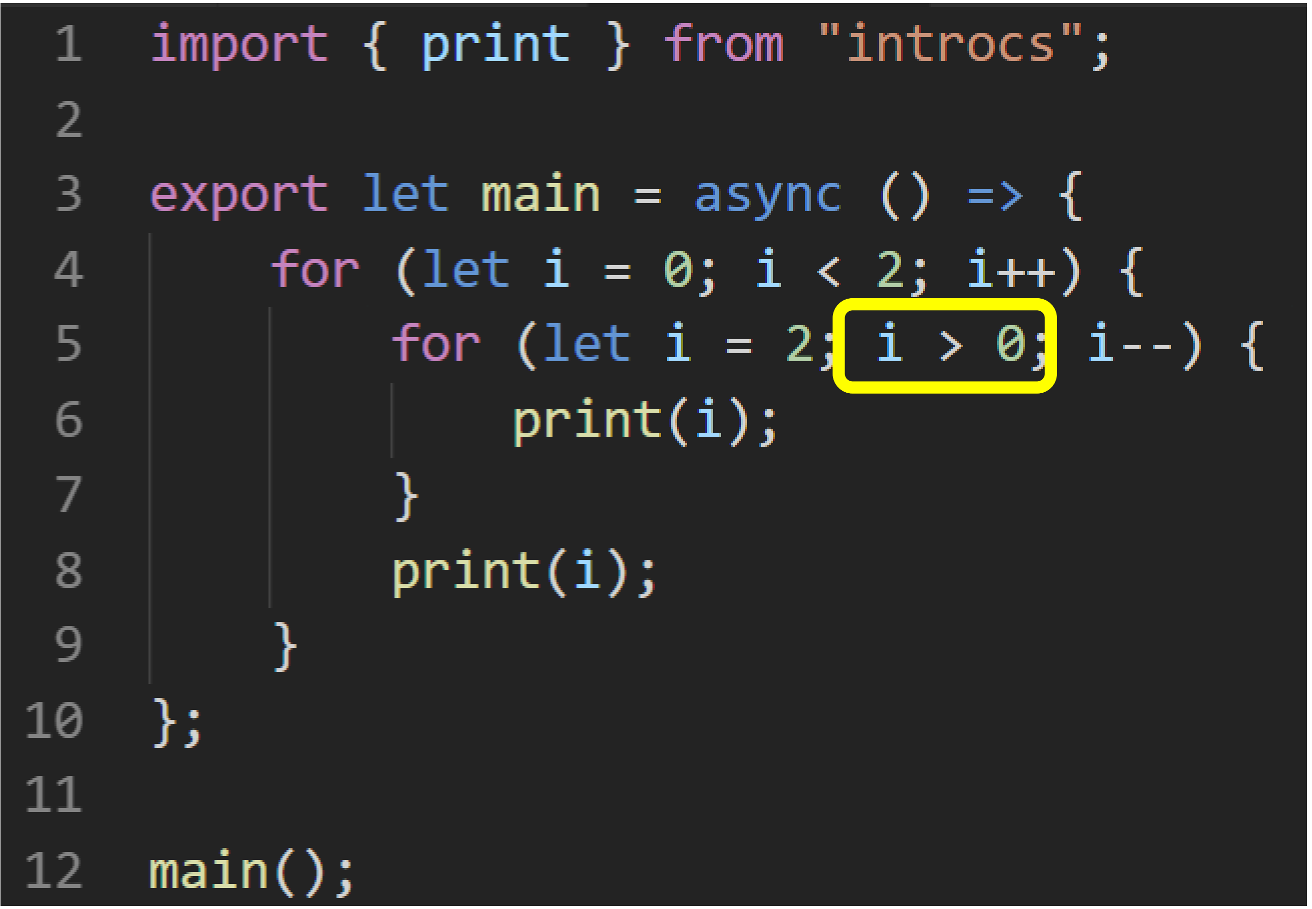


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

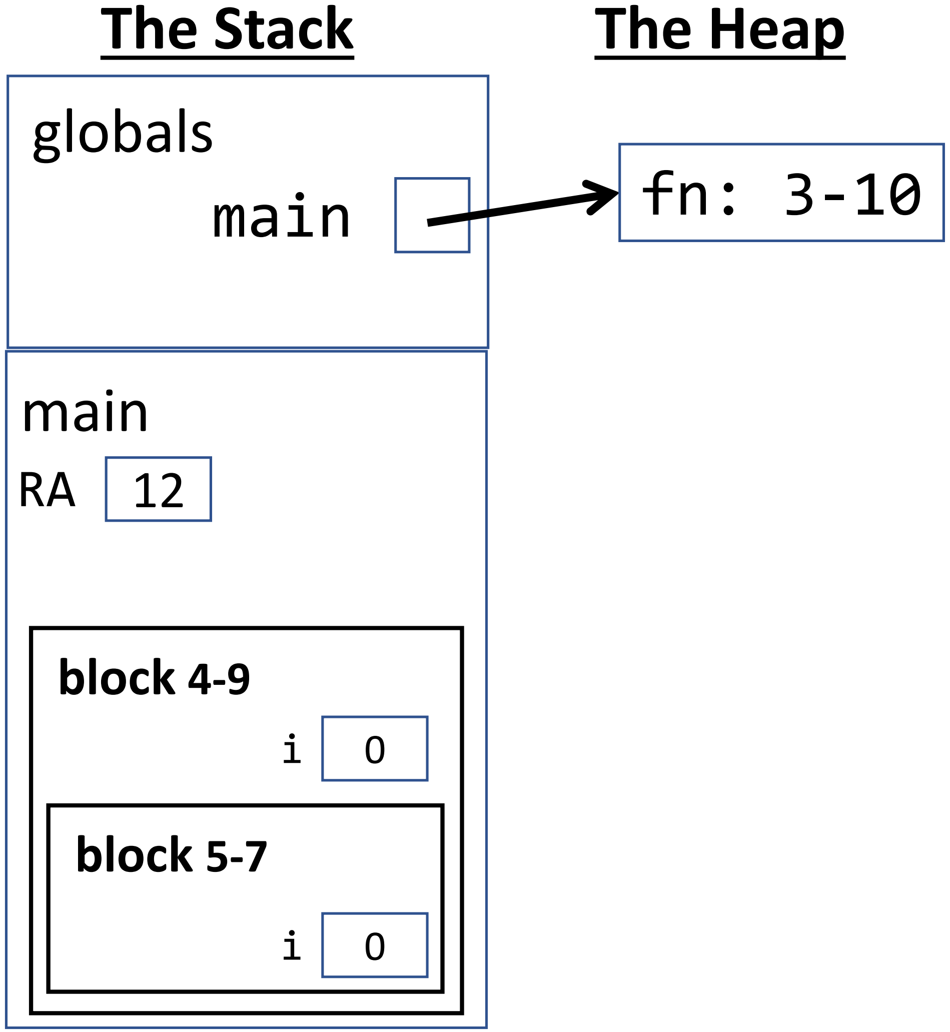


Name Resolution

How does the processor know which i? It looks in the most specific surrounding block first (lines 5-7 more specific than 4-9) checking blocks from the inside out.

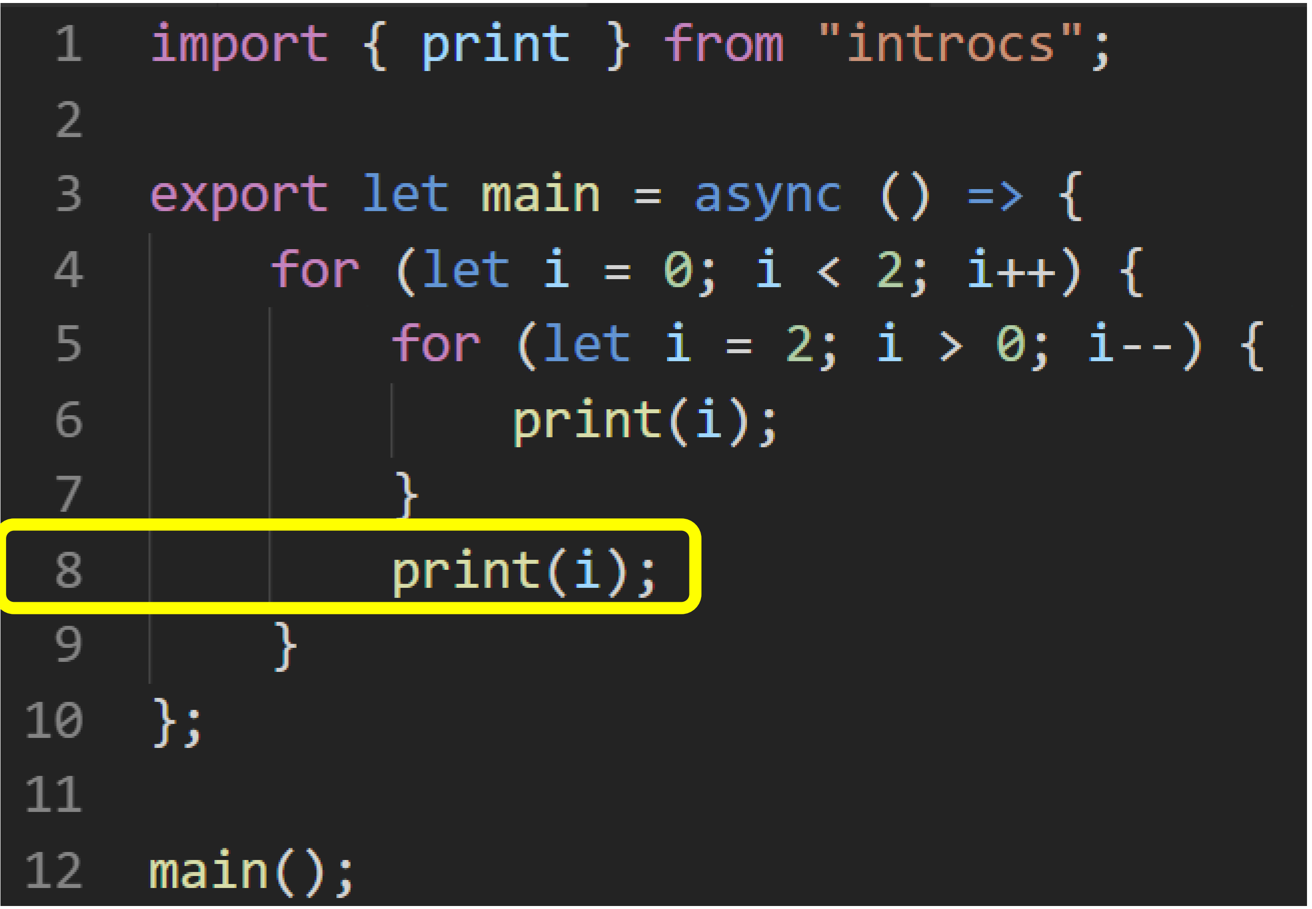


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

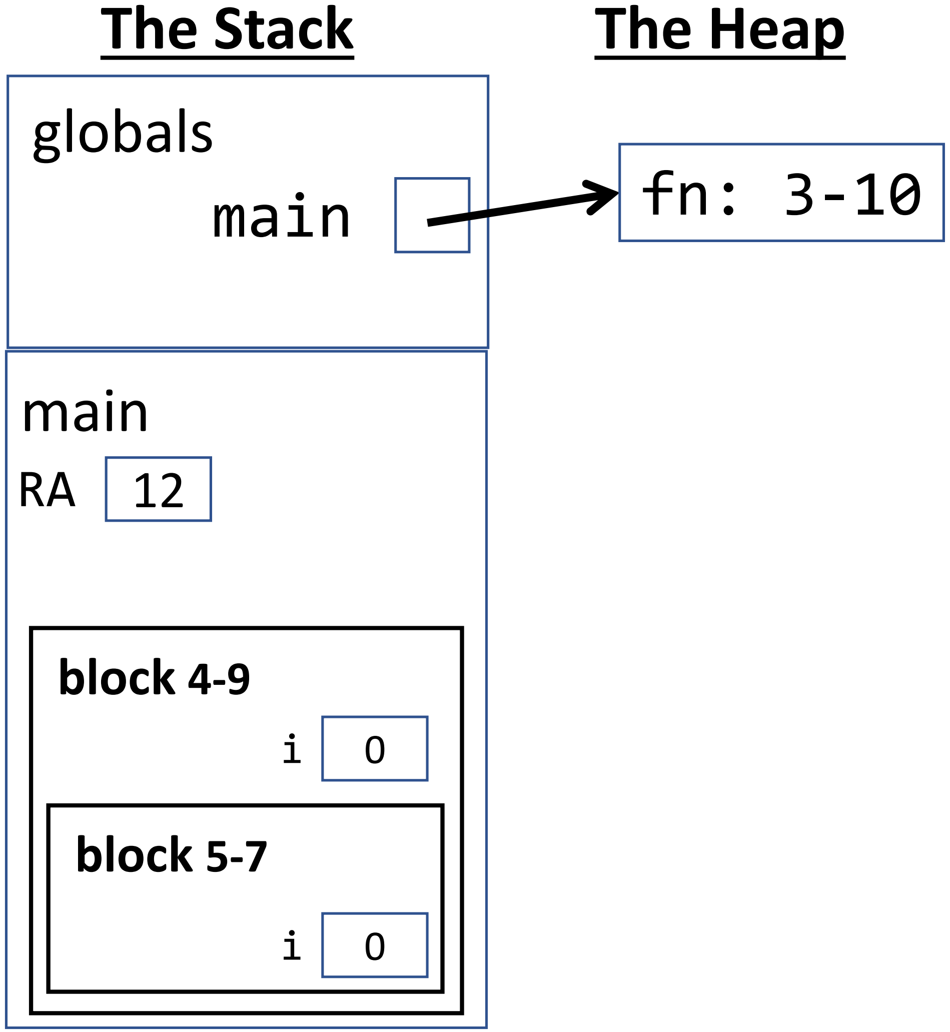


Name Resolution

Notice when line 8 is reached it is no longer in the block containing lines 5-7. At this point i refers to the one defined in the block surrounding lines 4-9.

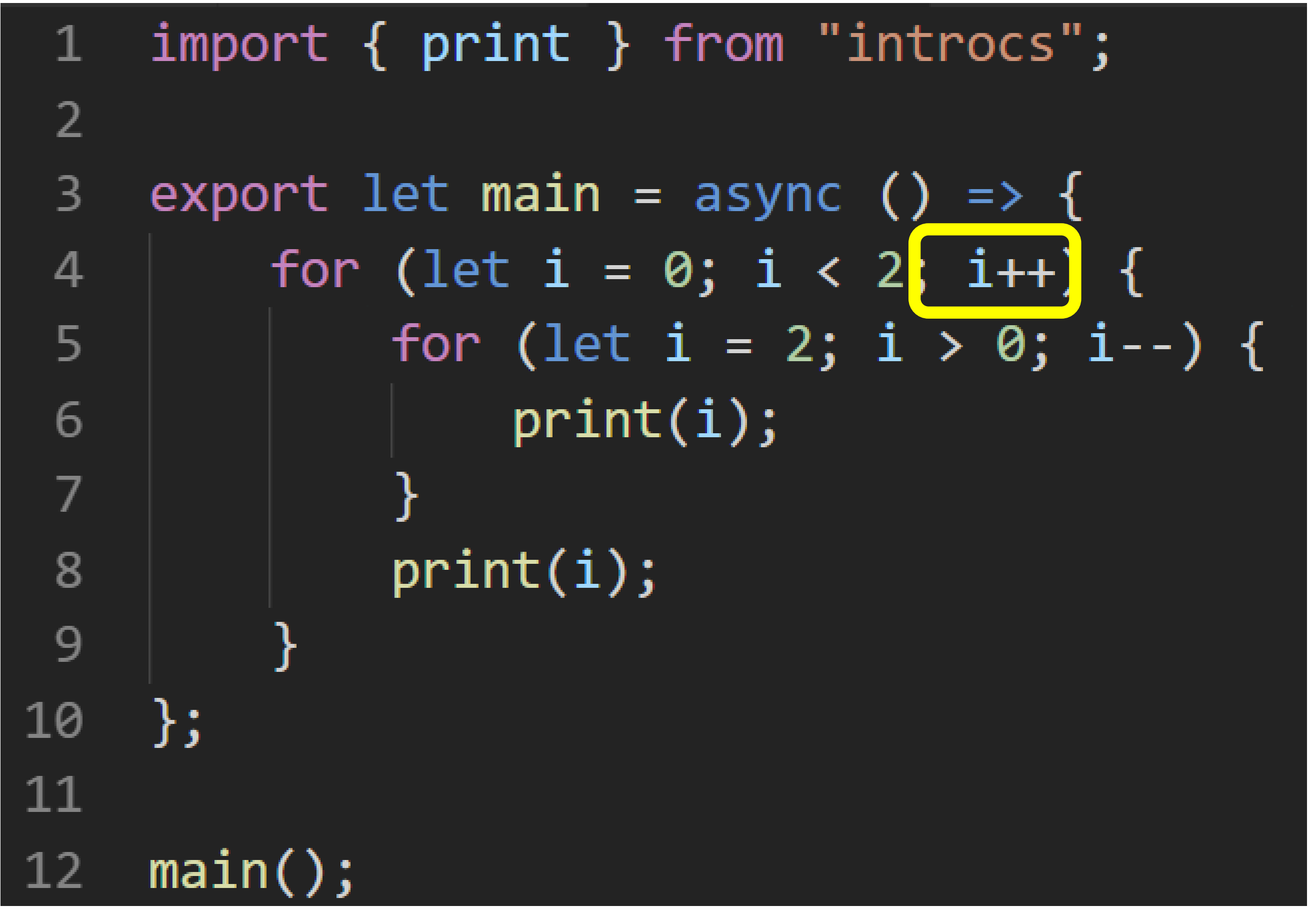


```
1  import { print } from "introcs";
2
3  export let main = async () => {
4      for (let i = 0; i < 2; i++) {
5          for (let i = 2; i > 0; i--) {
6              print(i);
7          }
8          print(i);
9      }
10 };
11
12 main();
```

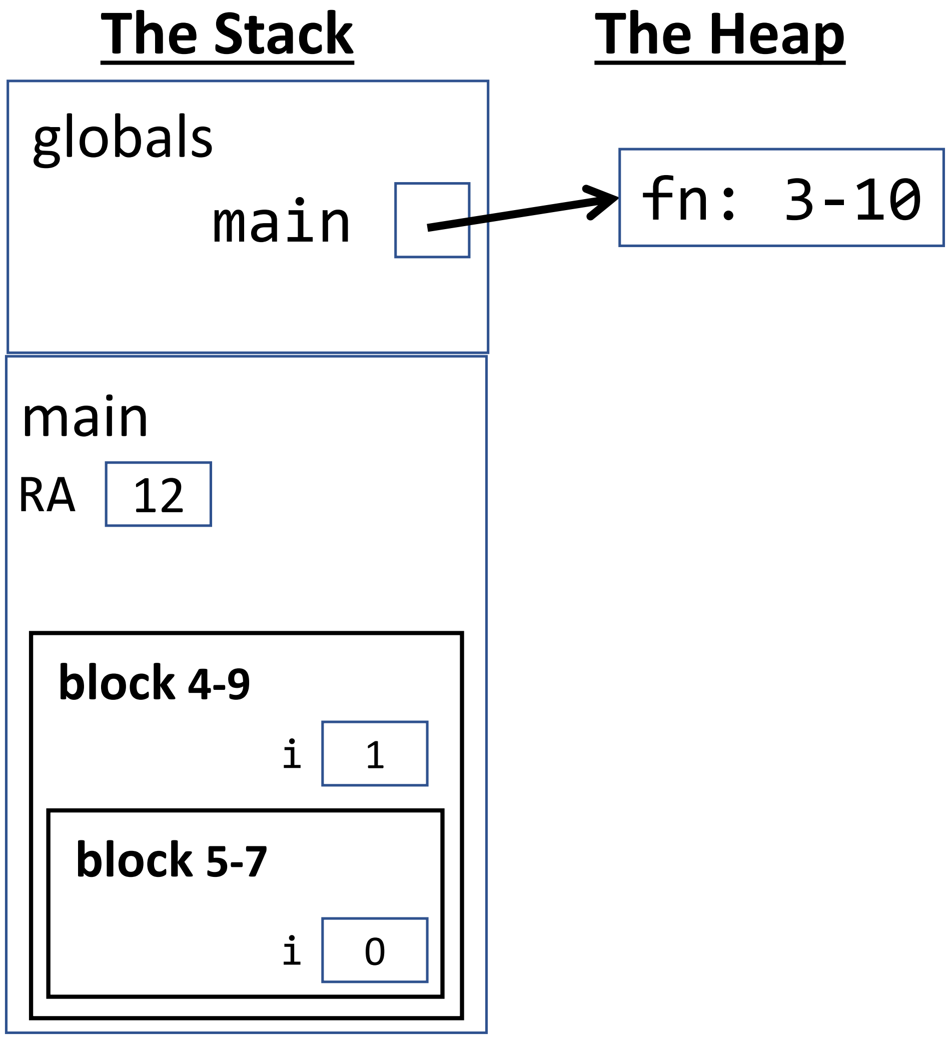


Name Resolution

Notice when line 8 is reached it is no longer in the block containing lines 5-7. At this point i refers to the one defined in the block surrounding lines 4-9.

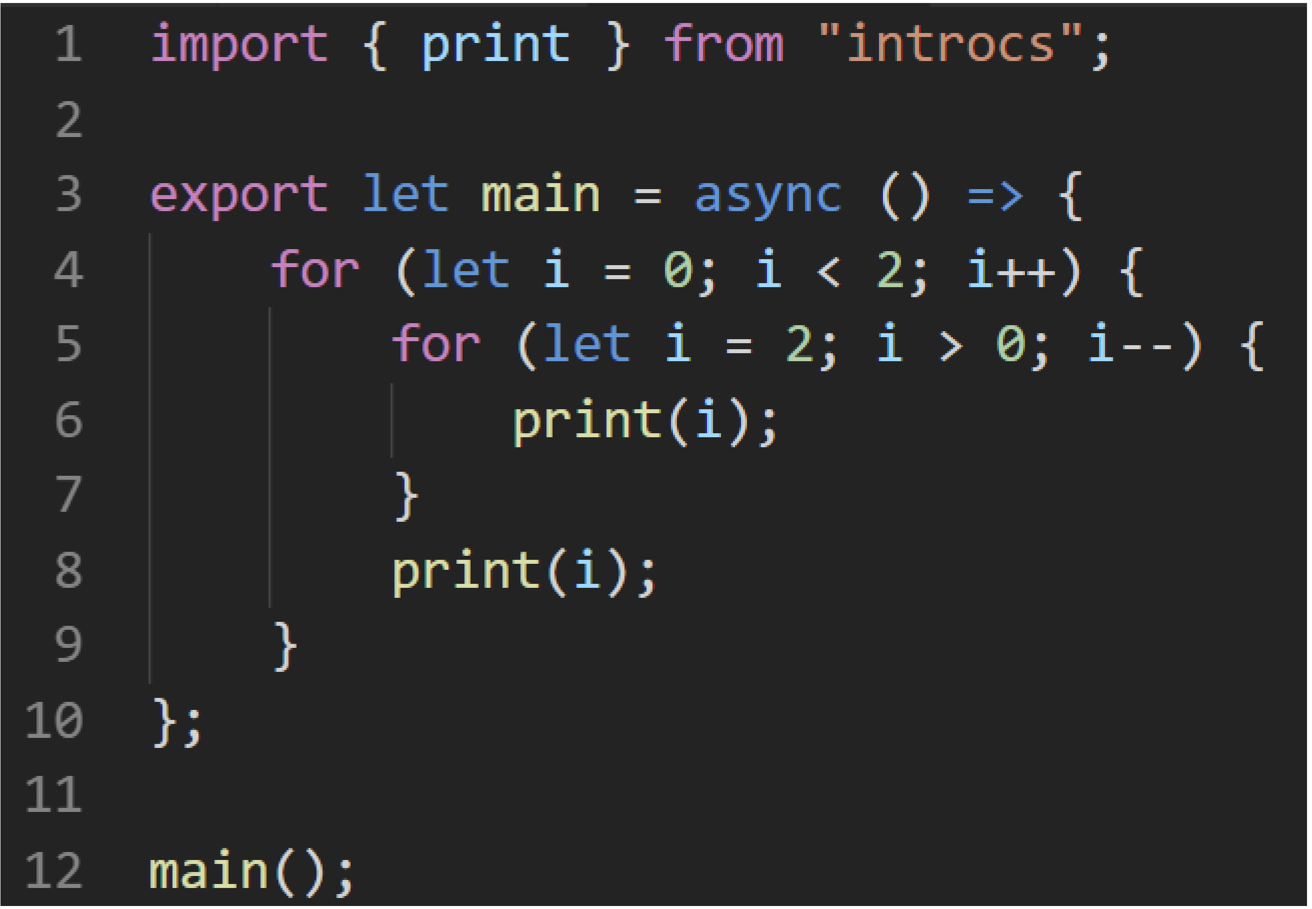


```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```

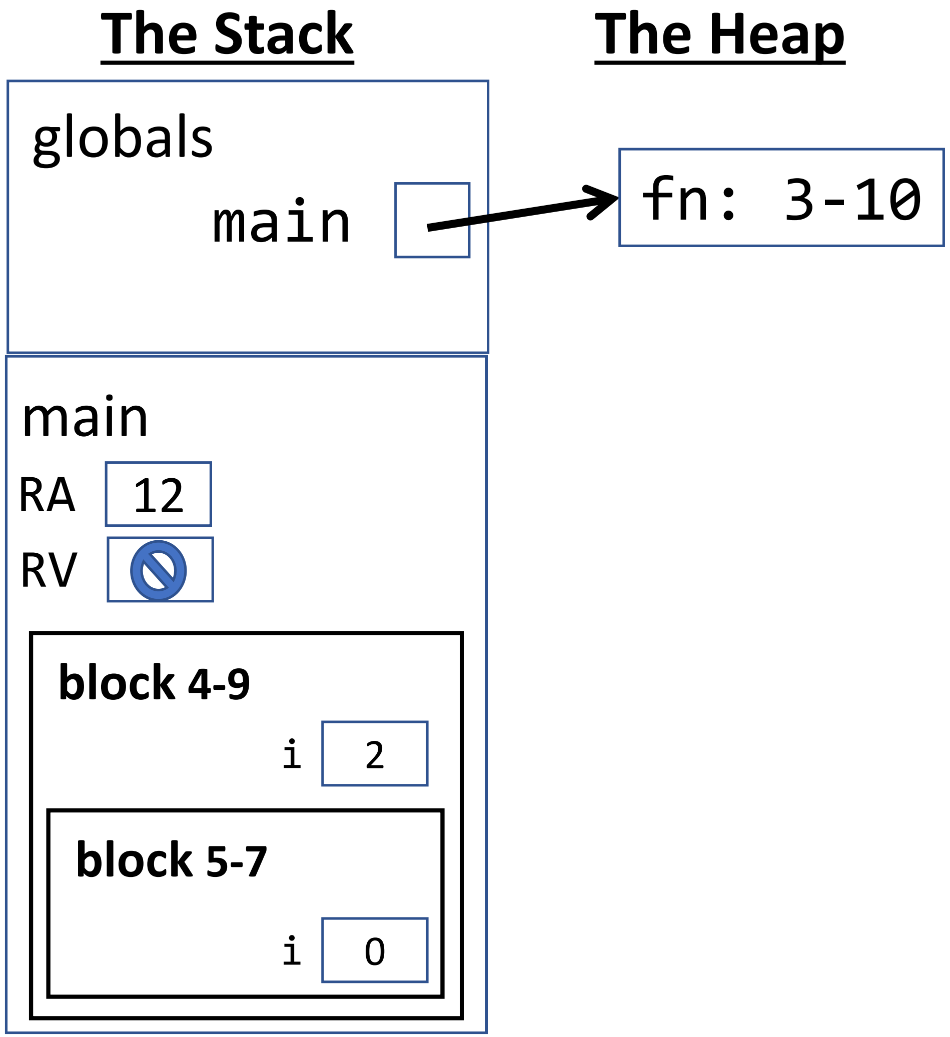


Fast-Forward

The final environment diagram would ultimately look like this. Reminder: Shadowing variable names in this way is bad practice! It is only shown here to illustrate the underlying rules at play.



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       for (let i = 0; i < 2; i++) {
5           for (let i = 2; i > 0; i--) {
6               print(i);
7           }
8           print(i);
9       }
10  };
11
12  main();
```



Name Resolution Rules

To find the space in memory *any* ***name*** (technically called an ***identifier***) is bound to in your program, follow these steps. The first rule to match wins.

1. If currently inside of a block: check the block first.
2. If currently inside of a nested block: check the blocks from inside-to-out.
3. Check the current frame (the last one added without an RV).
4. Check the Globals frame.