[– OOP –]

Mrenal Kanti Das

Timeline: From August 05, 2025
To [End Date]

In these notes, one tries to understand the pseudocode with a mathematical interpretation. One disects the codes with a mathematical lens, focusing on the semantics of operations and control flow.

1. One has;

```
1.
int x = 0, y = 0;
if (x++ && y++)
  printf("(%d, %d)", x, y);
else
  printf("(%d, %d)", y, x);
```

Let one denote program memory as a vector state $\mathcal{S} = (x, y) \in \mathbb{Z}^2$ (the state of the program at any moment can be represented as a vector state, and here $\mathcal{S}$ is a 2D state vector, and each $\mathcal{S}_i$ of $\vec{\mathcal{S}_i} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \cdots\}$ is component that represents the state of a program, or a system.), where, in **Line 01** we define the initial program state as;

$$\mathcal{S}_0 := \{x = 0,\ y = 0\}$$

No computation occurs here, and the variables $x, y \in \mathbb{Z}$. Define the post-increment operation as a function that produces both a value and a state transition. Let $\text{post\_inc} : \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ be;

$$\text{post\_inc}(z) := (z, z+1)$$

the post-increment semantics can be expressed as;

$$\begin{aligned} x++ &:= \text{return } x; \quad x \leftarrow x+1 \\ y++ &:= \text{return } y; \quad y \leftarrow y+1 \end{aligned}$$

This operation returns the current value z, then updates the memory allocation to $z + 1$. We define a function δ_z for the state transition;

$$\delta_z : z \mapsto z + 1$$

In **Line 02**, we evaluate the condition $x++ \wedge y++$ using the semantics of logical AND and post-increment. One should know that the *&&* operator in C is a short-circuit operator, meaning if the first operand is false, the second operand is never evaluated. Because the whole expression can never be true if the first one is false. The $x++$ operation is evaluated first, which is evaluated to 0 and then updates x to 1. So;

$$\begin{aligned} x++ \Rightarrow \text{evaluates to } 0,\ \text{then updates } x \leftarrow x+1 \leftarrow 0+1 \leftarrow 1 \\ \text{Since } x++ = 0,\ \text{the logical } \textbf{AND} \text{ short-circuits} \Rightarrow y++ \text{ is not evaluated} \end{aligned}$$

Thus in **Line 03**, we have the state transition;

$$\mathcal{S}_1 := \{x = 1,\ y = 0\}$$

And,

$$\text{Condition: } (x++ \wedge y++) = (0 \wedge \_) = \text{false}$$

One can *generally* see the condition $(x++ \wedge y++)$ can be viewed as;

$$\text{Let } f(x,y) = \begin{cases} \mathit{true} \ / \ 1 & \text{if } x++ \neq 0 \text{ and } y++ \neq 0 \\ \mathit{false} \ / \ 0 & \text{otherwise (due to short-circuit)} \end{cases}$$

and the short-circuiting behavior;

$$A \wedge B := \begin{cases} B & \text{if } A \neq 0 \\ \mathit{false} & \text{if } A = 0 \quad \text{(B not evaluated)} \end{cases}$$

In light of such, one sees that in **Line 4**, since the condition was false, we execute;

$$\texttt{print}(y, x) \Rightarrow (0, 1)$$

Thus, one has the final state of the program;

$$\mathcal{S}_{\text{final}} := \{x = 1, y = 0\}, \quad \text{Output: } (0, 1)$$

At a glance, one has;

Execution:

$$\begin{aligned} \mathcal{S}_0 &= \{x = 0,\ y = 0\} \\ \text{evaluate } x{+}{+} \Rightarrow \text{yields } 0 &\quad (\text{but sets } x \leftarrow x + 1) \\ \mathcal{S}_1 &= \{x = 1,\ y = 0\} \\ \text{short-circuit: } x{+}{+} = 0 \Rightarrow &\text{ no evaluation of } y{+}{+} \\ \mathcal{S}_2 = \mathcal{S}_1 &= \{x = 1,\ y = 0\} \\ \text{else branch executes: print } &(y, x) = (0, 1) \end{aligned}$$

And one implements the code in C language;

C Code:

```
#include <stdio.h>

int main() {
    int x = 0;
    int y = 0;

    if (x++ && y++) {
        printf("(%d, %d)\n", x, y);
    }
    else {
        printf("(%d, %d)\n", y, x);
    }
    return 0;
}
```

One takes another similar code;

2.

```
int x = 0, y = 0;
if (++x && ++y)
  printf("(%d, %d)", x, y);
else
  printf("(%d, %d)", y, x);
```

In **Line 01**, one initializes the values of x and y to 0. The initial state of the program is;

$$\mathcal{S}_0 := \{x = 0,\ y = 0\}$$

One has the pre-increment operator $++x$ as a function;

$$\begin{aligned} \text{pre\_inc}_x &: \mathbb{Z}^2 \to \mathbb{Z} \times \mathbb{Z}^2 \\ \text{pre\_inc}_x(x, y) &= (x + 1,\ (x + 1,\ y)) \end{aligned}$$

Such is, it returns the incremented value $x+1$, and then updates the state to $(x+1, y)$. Similarly, for $++y$;

$$\text{pre\_inc}_y(x, y) = (y+1,\ (x,\ y+1))$$

Now we consider the Boolean condition $(++x \wedge ++y)$. Let the initial state be;

$$\mathcal{S}_0 = (x, y) = (0, 0)$$

One has for $++x$, and for $++y$;

$$\begin{aligned} \text{pre\_inc}_x(\mathcal{S}_0) &= (x_1, \mathcal{S}_1), \quad \text{where } x_1 = x+1,\ \mathcal{S}_1 = (x+1, y) \\ \text{pre\_inc}_y(\mathcal{S}_1) &= (y_1, \mathcal{S}_2), \quad \text{where } y_1 = y+1,\ \mathcal{S}_2 = (x+1, y+1) \end{aligned}$$

With the values, in **Line 02** we have;

$$\begin{aligned} x &\leftarrow x+1 \Rightarrow x = 1 \\ y &\leftarrow y+1 \Rightarrow y = 1 \\ \Rightarrow \mathcal{S}_1 &= (1, 1) \end{aligned}$$

Then the short-circuit evaluation is;

$$(++x \wedge ++y) = \begin{cases} \text{true} & \text{if } x_1 \neq 0 \text{ and } y_1 \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

Here we see;

$$(++x \neq 0) \wedge (++y \neq 0) \Rightarrow \texttt{true}$$

Therefore, the `if`-branch is executed, and we print the values of x and y;

$$\texttt{printf("(\%d, \%d)", x, y)} \Rightarrow (1, 1)$$

This can be seen as the flow of the program;

Execution:

$$\begin{aligned} \mathcal{S}_0 &= \{x = 0,\ y = 0\} \\ \text{evaluate } ++x &\Rightarrow x \leftarrow x+1,\ \text{yields } 1 \\ \text{evaluate } ++y &\Rightarrow y \leftarrow y+1,\ \text{yields } 1 \\ \mathcal{S}_1 &= \{x = 1,\ y = 1\} \\ \text{condition: } 1 \wedge 1 &\Rightarrow \texttt{true} \\ \text{if branch executes: print } (x, y) &= (1, 1) \end{aligned}$$

One has the C code for this logic;

C Code:

```c
#include <stdio.h>

int main() {
    int x = 0, y = 0;

    if (++x && ++y) {
        printf("(%d, %d)", x, y);
    } else {
        printf("(%d, %d)", y, x);
    }

    return 0;
}
```

NOTE: In C, the *&&* operator evaluates left-to-right and stops evaluation (short-circuits) if the first operand happens to be zero. This prevents unnecessary evaluation of the second operand, which is especially important if it has side effects. Here one sees that, the else block is entirely skipped, then continues sequentially to the next line after the entire if-else statement; return 0;. So, regardless of which branch runs, after completing the chosen block or branch, the program proceeds to execute the return 0; statement.

We now take a more complex example with both pre- and post-decrement operators, and logical operations;

3.
```
int x = 1, y = 1;
if (x - - || y - - && ++x)
{
  printf("Yes");
}
else {
  printf("No");
}
```

In **Line 01**, variables x and y are initialized as the state vector;

$$\mathcal{S}_0 := \{x = 1,\ y = 1\}$$

We model the operators as functions with side effects;

$$\text{post\_dec}_x : \mathbb{Z}^2 \to \{0,1\} \times \mathbb{Z}^2, \quad \text{post\_dec}_x(x,y) = \big(x,\ (x-1,y)\big)$$

It returns the original value of x, then updates state $x \to x - 1$. Similarly, for $y - -$;

$$\text{post\_dec}_y(x,y) = \big(y,\ (x, y-1)\big)$$

The pre-increment operator $++x$ is;

$$\text{pre\_inc}_x(x,y) = \big(x+1,\ (x+1,y)\big)$$

The condition in **Line 02** is evaluated as follows; $(x - - \vee (y - - \wedge ++x))$. Start with initial state $\mathcal{S}_0 = (1,1)$. Then evaluate $x - -$;

$$\text{post\_dec}_x(1,1) = (1,(0,1)) = (v_x, \mathcal{S}_1)$$

So, the value of $x - -$ is $v_x = 1$ and the state updates to;

$$\mathcal{S}_1 = (0,1)$$

The short-circuit OR behavior means if the first operand is true, the second operand is not evaluated. Since $x - -$ evaluates to 1 (true), we skip evaluating $y - - \wedge ++x$. Thus, we have;

$$x - - \neq 0 \implies \text{skip } y - - \wedge ++x$$

No further evaluation occurs; the state remains $\mathcal{S}_1$. As branching; since the condition evaluates to true $(1 \neq 0)$, the if-block executes;

```
printf("Yes")
```

and the else-block is skipped. Final state and output are;

$$\mathcal{S}_{\text{final}} = (0,1), \quad \text{output} = \text{"Yes"}$$

Execution:

$$\begin{aligned} \mathcal{S}_0 &= \{x = 1, y = 1\} \\ \text{evaluates } x-- : \text{value } &= 1, \text{ state update: } \mathcal{S}_1 = \{x = 0, y = 1\} \\ \text{short-circuit:} & \quad \text{skips evaluation of } y-- \wedge ++x \\ \mathcal{S}_2 &= \mathcal{S}_1 = \{x = 0, y = 1\} \\ \text{condition } &= \text{true} \Rightarrow \text{if-branch runs} \\ \text{output} &= \text{"Yes"} \end{aligned}$$

The C code for this logic is as follows;

C Code:

```
#include <stdio.h>

int main() {
    int x = 1, y = 1;

    if (x-- || y-- && ++x) {
        printf("Yes\n");
    } else {
        printf("No\n");
    }

    return 0;
}
```

NOTE about Line 02: Let the initial program state be;

$$\mathcal{S}_0 := \{x = 1,\ y = 1\}$$

The condition $x--\ ||\ y--\ \&\&\ ++x$ is evaluated left to right, using short-circuit semantics. First, consider the post-decrement operator $x--$, which evaluates to the original value of x, then decreases x by 1. We define the post decrement and then apply this to $\mathcal{S}_0$;

$$\text{post\_dec}_x(1, 1) = (1,\ (0, 1)) \Rightarrow \text{value } = 1,\ \text{new state } \mathcal{S}_1 := \{x = 0,\ y = 1\}$$

Since the value returned is non-zero ($1 \neq 0$), the logical OR `||` short-circuits — i.e., the right-hand side $y--\ \&\&\ ++x$ is not evaluated at all. Therefore, no change occurs to y, and $++x$ is not executed. Hence, the final state before entering the `if`-branch is;

$$\mathcal{S}_1 = \{x = 0,\ y = 1\}$$

Since the entire condition evaluates to `true`, the `if`-branch is executed.
NOTE on Short-Circuiting: Let $A, B \in \{0, 1\}$, where; 0 denotes `false`, and 1 denotes `true`. The logical OR operator, denoted by $A \vee B$, is defined as a binary function;

$$\vee : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$$

such that:

$$A \vee B = \begin{cases} 1 & \text{if } A = 1 \text{ or } B = 1 \\ 0 & \text{if } A = 0 \text{ and } B = 0 \end{cases}$$

This satisfies the inclusion rule: the OR operation returns true if at least one operand is true. In C, the `||` operator uses **short-circuit evaluation**, which means; if $A \neq 0$, then B is *not evaluated*, and if $A = 0$, then B *is evaluated.* This can be formally written as;

$$A||B = \begin{cases} 1 & \text{if } A \neq 0 \quad \text{(B not evaluated)} \\ B & \text{if } A = 0 \quad \text{(B is evaluated)} \end{cases}$$

One has a truth table for the logical OR operation;

A	**B**	**A $\vee$ B**
0	0	0
0	1	1
1	0	1
1	1	1

Table 1: Truth Table for Logical OR ($\vee$)

One might ask, but the truth table shows only the logical result of an operation given its inputs. It does not encode the process by which the result is computed — specifically, it does not reflect short-circuit evaluation behavior, which is an implementation strategy in imperative programming languages like C, not logic itself. The table tells us that it is a commutative, associative, and idempotent binary operation. For the short circuit behavior, one can think of it as a computational strategy that optimizes performance by avoiding unnecessary evaluations. Meaning, if the value of A is true, then the whole expression $(A \vee B)$ is true (see the table), and there is no need to evaluate B at all. This is a practical implementation detail that can lead to performance improvements in certain cases, especially when B involves expensive computations or side effects.
NOTE on Table: To understand what the table says, one lets $A, B, C \in \{0, 1\}$. The logical OR (disjunction) operation $\vee : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$ is defined by;

$$A \vee B := \begin{cases} 1 & \text{if } A = 1 \text{ or } B = 1 \\ 0 & \text{if } A = 0 \text{ and } B = 0 \end{cases}$$

And such a definition satisfies, or displays the following properties;
Commutativity:

$$\forall A, B \in \{0, 1\}, \quad A \vee B = B \vee A$$

Associativity:

$$\forall A, B, C \in \{0, 1\}, \quad A \vee (B \vee C) = (A \vee B) \vee C$$

Idempotence:

$$\forall A \in \{0, 1\}, \quad A \vee A = A$$

One sees; and understands the behavior of the table. The table shows the results of the logical

A	**B**	**A $\vee$ B**	**B $\vee$ A**	**A $\vee$ A**
0	0	0	0	0
0	1	1	1	0
1	0	1	1	1
1	1	1	1	1

Table 2: Truth table for logical OR operations

OR operation for all combinations of A and B. The first two columns represent the inputs, and the last three columns show the results of the operation. The first column is A, the second is B, and the third column is $A \vee B$. The fourth column is $B \vee A$, which is equal to $A \vee B$ due to commutativity. The fifth column is $A \vee A$, which is equal to A due to idempotence.

Because of ones and zeroes, it displays a behvior of the logical OR operation, which is a binary operation that takes two inputs and produces one output. The output is true (1) if at least one of the inputs is true (1), and false (0) if both inputs are false (0). Now, it does not hurt to see the initials to be something other than the 0s' and 1s'.

One has;

4.

```
int a = 3, int b = 4;
(a > b) ?  (a = b) :  (b = a);
printf("(%d, %d)", a, b);
```

At the start;

$$\mathcal{S}_0 := (a,b) = (3,4)$$

The ternary operator in C;

$$a > b?(a = b) : (b = a)$$

is semantically equivalent to the if-else;

$$\text{if } a > b \text{ then } a = b \text{ else } b = a$$

Define assignment as a function that updates the state: if we assign $a := b$, then $f_a(\mathcal{S}) = (b,b)$, if we assign $b := a$, then $f_b(\mathcal{S}) = (a,a)$. From the initial state $\mathcal{S}_0 = (3,4)$, evaluate;

$$3 > 4 \equiv \text{false}$$

Therefore, the condition fails, and the else branch executes. Else branch is $b := a$. Applying f_b to $\mathcal{S}_0$;

$$\mathcal{S}_1 = f_b(3,4) = (a,a) = (3,3)$$

One has the flow as;

Execution:

$$\begin{aligned} \text{initialize: } & \mathcal{S}_0 = (a,b) = (3,4) \\ \text{evaluate } a > b \text{ ? } (a = b) \ : \ (b = a) & \\ \text{condition } & a > b = 3 > 4 = \text{false} \\ \text{else-branch executes: } & \quad b := a \\ \text{apply } f_b : & \, f_b(3,4) = (a,a) = (3,3) \\ & \mathcal{S}_1 = (a,b) = (3,3) \end{aligned}$$

One sees the C code;

C Code:

```c
#include <stdio.h>

int main() {
    int a = 3, b = 4;

    a > b ? (a = b) : (b = a);

    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Moving on...

5.

```
int a = 3, int b = 4;
if (!a > -b) { print(a, b);
}
else { print(b, a);
}
```

The initial program state is;

$$\mathcal{S}_0 := (a, b) = (3, 4)$$

The logical NOT operator in **Line 03** can be defined as a function;

$$\neg : \mathbb{Z} \to \{0, 1\}, \quad \neg x = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{if } x \neq 0 \end{cases}$$

The $\neg$ operator is a truth value inversion, which evaluate $\neg a$ as;

$$\neg a = \neg 3 = 0$$

One has the comparison;

$$(\neg a) > (-b) \implies 0 > -4 = \text{true}$$

Therefore, the condition;

$$(\neg a > -b) = \text{true}$$

implies the execution of the then-branch, which outputs $(a, b) = (3, 4)$. No mutation of state occurs, so the final state is;

$$\mathcal{S}_1 = (3, 4)$$

Execution:

$$\begin{aligned} \mathcal{S}_0 &= (a, b) = (3, 4) \\ \text{evaluate condition} &\quad \neg a > -b \\ \text{step 1: } \neg a &= \neg 3 = 0 \quad (\text{since C interprets nonzero as true} \to !3 = 0) \\ \text{step 2: } 0 &> -4 = \text{true} \\ \text{condition is true} &\Rightarrow \text{then-branch executes} \\ \text{output} &= \texttt{a = 3, b = 4} \\ \mathcal{S}_1 &= \mathcal{S}_0 = (3, 4) \end{aligned}$$

C Code:

```
#include <stdio.h>

int main() {
    int a = 3;
    int b = 4;

    if (!a > -b) {
        printf("(%d, %d)\n", a, b);
    }
    else {
        printf("(%d, %d)\n", b, a);
    }
    return 0;
}
```

Another, another one;

```
6.
int a = 2, int b = 4;
int n = a > b ? (a) : (b);
print(n);
```

In **Line 01**, variables a and b are initialized as the state vector;

$$\mathcal{S}_0 := \{a = 2,\ b = 4,\ n = \perp\}$$

where $\perp$ is indicating that n is not yet assigned. The ternary conditional operator ? : can be modeled as a function;

$$\text{tern} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

defined by;

$$\text{tern}(c, x, y) = \begin{cases} x, & \text{if } c \neq 0, \\ y, & \text{if } c = 0. \end{cases}$$

In **Line 02**, the condition $a > b$ is evaluated with respect to the current state $\mathcal{S}_0$;

$$\mathit{eval}(\mathcal{S}_0, a > b) = \begin{cases} 1, & \text{if } \mathcal{S}_0(a) > \mathcal{S}_0(b), \\ 0, & \text{otherwise.} \end{cases}$$

We have;

$$n := \text{tern}(\text{condition}, v_a, v_b \in \mathbb{Z})$$

where n is still not assigned any values ($\perp$). One can say that at this point of the program n does not hold any meaningful value; it is uninitialized. And v_a, and v_b are the current values of a, and b. Since $2 \not> 4$, we have;

$$\mathit{eval}(\mathcal{S}_0, a > b) = 0.$$

Therefore, the ternary expression evaluates as;

$$\text{tern}\Big(\mathit{eval}(\mathcal{S}_0, a > b), \mathcal{S}_0(a), \mathcal{S}_0(b)\Big) = \text{tern}(0, 2, 4) = 4.$$

The state updates with the assignment to n as;

$$\mathcal{S}_1 := \mathcal{S}_0[n \mapsto 4] = \{a = 2,\ b = 4,\ n = 4\}.$$

Finally, the print statement outputs the current value of n,

$$\text{output} = 4,$$

and the program terminates with the final state $\mathcal{S}_1$.

Execution:

$$\begin{aligned} \mathcal{S}_0 &= (a, b, n) = (2, 4, \perp) \\ \text{evaluate condition} &\quad a > b \\ \text{step 1: } & 2 > 4 = \text{false} \\ \text{condition is false} &\Rightarrow \text{ternary operator selects } b = 4 \\ \text{assignment: } & n := 4 \\ \mathcal{S}_1 &= (a, b, n) = (2, 4, 4) \\ \text{output} &= 4 \end{aligned}$$

C Code:

```
#include <stdio.h>

int main() {
    int a = 2, b = 4;
    int n = a > b ? a : b;
    printf("%d\n", n);
    return 0;
}
```

One has;

7 & 8.

```
int x;
x = 3, 4, 5;
print(x);

x = (3, 4, 5);
print(x);
```

One sees the program state as a mapping from variables to values;

$$\mathcal{S} : \{\mathrm{x}\} \to \mathbb{Z} \cup \{\perp\}$$

where $\perp$ means the value is still not assigned. Initially, before execution;

$$\mathcal{S}_0(\mathrm{x}) = \perp$$

The expression;

$$x = 3, 4, 5;$$

is parsed according to operator precedence as;

$$(x = 3),\ 4,\ 5;$$

The comma operator evaluates each expression from left to right, discarding all values except the last in an expression context. Evaluate the assignment;

$$\text{eval}(\mathcal{S}_0, x = 3) = (3, \mathcal{S}_1), \quad \mathcal{S}_1 = \mathcal{S}_0[\mathrm{x} \mapsto 3]$$

One continues;

$$\text{eval}(\mathcal{S}_1, 4) = (4, \mathcal{S}_1)$$

$$\text{eval}(\mathcal{S}_1, 5) = (5, \mathcal{S}_1)$$

Therefore, the entire expression evaluates to;

$$\text{eval}(\mathcal{S}_0, x = 3, 4, 5) = (5, \mathcal{S}_1)$$

Since the value of the entire expression is not used or assigned, only the side effects matter. The final program state is;

$$\mathcal{S}_1 = \{\mathrm{x} \mapsto 3\}$$

When the `print` statement executes, it reads the value;

$$\text{output} = \mathcal{S}_1(\mathrm{x}) = 3$$

and prints it.

Execution:

$$\begin{aligned}
\mathcal{S}_0 &= (x) = (\bot) \\
\text{evaluate expression} \quad & x = 3,4,5 \\
& \text{step 1: } (x=3) \implies \mathcal{S}_1 = (x) = (3) \\
& \text{step 2: } \text{4 evaluated, no state change} \\
& \text{step 3: } \text{5 evaluated, no state change} \\
\text{final state} &\Rightarrow \mathcal{S}_1 = (x) = (3) \\
\text{output} &= 3
\end{aligned}$$

Again let the program state be;

$$\mathcal{S} : \{\mathrm{x}\} \to \mathbb{Z} \cup \{\bot\}$$

The evaluation function;

$$\mathrm{eval}(\mathcal{S}, e) = (v, \mathcal{S}')$$

which evaluates expression e in state $\mathcal{S}$ producing value v and updated state $\mathcal{S}'$. The comma operator evaluates left to right and returns the last value;

$$\mathrm{eval}(\mathcal{S}, (3,4,5)) = (5, \mathcal{S})$$

The assignment;

$$x = (3,4,5)$$

is evaluated by first evaluating the right-hand side;

$$\mathrm{eval}(\mathcal{S}_0, (3,4,5)) = (5, \mathcal{S}_0)$$

and then updating the state;

$$\mathcal{S}_1 = \mathcal{S}_0[x \mapsto 5]$$

Finally, printing x reads;

$$\mathrm{eval}(\mathcal{S}_1, x) = (5, \mathcal{S}_1) \implies \text{output} = \mathcal{S}_1(\mathrm{x}) = 5$$

and outputs 5.

Execution:

$$\begin{aligned}
\mathcal{S}_0 &= (x) = (\bot) \\
\text{evaluate } (3,4,5) &\to 5 \\
\text{assign } x &:= 5 \Rightarrow \mathcal{S}_1 = (x) = (5) \\
\text{output} &= 5
\end{aligned}$$

C Code:

```c
#include <stdio.h>
int main()
{
    int x;
    x = 3, 4, 5;
    printf("%d\n", x);  // will print 3
    x = (3, 4, 5);
    printf("%d\n", x);  // will print 5

    return 0;
}
```

One has;

9.

```
int x = 3;
print(x * x, ++x, x++);
```

Initial program state;

$$\mathcal{S}_0 = \{x = 3\}$$

Assuming left-to-right evaluation of the function arguments, we evaluate $x * x$;

$$v_1 = \mathcal{S}_0(x) \times \mathcal{S}_0(x) = 3 \times 3 = 9$$

State remains unchanged;

$$\mathcal{S}_1 = \mathcal{S}_0 = \{x = 3\}$$

The ++x pre-increment;

$$v_2 = \mathcal{S}_1(x) + 1 = 4$$

The state updates;

$$\mathcal{S}_2 = \mathcal{S}_1[x \mapsto 4] = \{x = 4\}$$

The x++ post-increment;

$$v_3 = \mathcal{S}_2(x) = 4$$

Updated state;

$$\mathcal{S}_3 = \mathcal{S}_2[x \mapsto 5] = \{x = 5\}$$

Final output arguments to printf;

$$(9, 4, 4)$$

Final program state;

$$\mathcal{S}_3 = \{x = 5\}$$

Execution:

$$\begin{aligned}
\mathcal{S}_0 &= (x) = (3) \\
\text{evaluate } x * x &= 9 \\
\text{evaluate } ++x : x &= 4, \text{ value } = 4 \\
\text{evaluate } x++ : \text{ value } &= 4, x = 5 \\
\text{output} &= \texttt{9, 4, 4} \\
\mathcal{S}_{\text{final}} &= (x) = (5)
\end{aligned}$$

Java Code:

```
public class Main {
    public static void main(String[] args) {
        int x = 3;
        System.out.printf("%d, %d, %d\n", x * x, ++x, x++);
    }
}
```

NOTE: The order of evaluation of function arguments is unspecified in C, so this code exhibits undefined behavior. Another thing, which we were going for anyway, that we will be using JAVA to see things from now on. We have reasons, and let us follow the approach Knuth's analysis of algorithms takes, and somewhat similar to the operational semantics of programming languages. We also will be following his student Sedgewick's book too.

Prime Numbers: Primes are very interesting as numbers, because they are something distinguishable among the integers. They are the building blocks of the integers, in the sense that every integer can be factored into primes. The fundamental theorem of arithmetic states that every integer greater than 1 is either a prime itself or can be uniquely factored as a product of primes. This property makes primes essential in number theory and various applications, including cryptography. And one of the puzzling questions of modern mathematics is the distribution of primes among the integers. The prime number theorem gives an asymptotic form for the number of primes less than a given integer, but many questions about the distribution of primes remain open, such as the Riemann Hypothesis and the Twin Prime Conjecture. Anyway...

Let $\mathbb{N} = \{1, 2, 3, \dots\}$ denote the natural numbers. We first define the notion of divisibility; for $a, b \in \mathbb{N}$,

$$a \mid b \iff \exists k \in \mathbb{N} \, : \, b = a \cdot k.$$

A number $p \in \mathbb{N}$ with $p > 1$ is called a *prime number* if it satisfies either of the following equivalent conditions;

(i) p has no positive divisors other than 1 and itself, i.e.

$$\forall d \in \mathbb{N}, \quad d \mid p \;\Rightarrow\; (d = 1 \;\vee\; d = p).$$

(ii) p satisfies Euclid's lemma: for all $a, b \in \mathbb{N}$,

$$p \mid (ab) \;\Rightarrow\; (p \mid a \;\vee\; p \mid b).$$

A number $n > 1$ which is not prime is called *composite*. Formally,

$$n \text{ is composite} \iff \exists a, b \in \mathbb{N}, \quad 1 < a < n, \; 1 < b < n, \; n = a \cdot b.$$

Thus, the set of all primes can be defined as

$$\mathbb{P} = \{\, p \in \mathbb{N} \;\mid\; p > 1 \;\wedge\; \forall d \in \mathbb{N}, \; (d \mid p \Rightarrow d = 1 \vee d = p) \,\}.$$

Now we look at a code to check if a number is prime;

10.1 Prime

```
int $n$;
if $n \leq 1$
  print "not prime"
else;
  $i \leftarrow 2$
  while $i^2 \leq n$
    if $n \equiv 0 \pmod{i}$
      print "not prime"
      stop
    $i \leftarrow i + 1$
  print "prime"
```

We model the program state as a mapping from variables to values;

$$\mathcal{S} : \{\mathtt{n}, \mathtt{i}, \mathtt{isPrime}\} \to \mathbb{Z} \cup \{\perp\},$$

where $\perp$ represents an unassigned value. Initially one has;

$$\mathcal{S}_0(\mathtt{n}) = \perp, \quad \mathcal{S}_0(\mathtt{i}) = \perp, \quad \mathcal{S}_0(\mathtt{isPrime}) = \perp$$

Then one takes the input;

$$\mathcal{S}_1 = \mathcal{S}_0[\mathrm{n} \mapsto n]$$

The state will update, and check the base case;

$$\text{if } n \leq 1 \implies \text{print "not prime" (stop)}$$

Here in this line, if the integer is not prime, it will be caught by the base case check, and the program will terminate early. Then, the program approaches to initialize the loop;

$$\mathcal{S}_2 = \mathcal{S}_1[\mathrm{i} \mapsto 2,\ \texttt{isPrime} \mapsto 1]$$

One has in the while loop $i^2 \leq n$;

$$\begin{aligned} \text{if } n \equiv 0 \pmod{i} \implies \mathcal{S}_2[\texttt{isPrime} \mapsto 0], \text{ print "not prime" (stop)} \\ \text{else increment } i : \mathcal{S}_2 \leftarrow \mathcal{S}_2[\mathrm{i} \mapsto i+1] \end{aligned}$$

The program proceeds to termination;

$$\text{if no divisor found up to } \sqrt{n},\ \texttt{isPrime} = 1 \implies \texttt{print "prime"}$$

Remark: This formalism treats the program as checking divisibility symbolically for all i such that $2 \leq i \leq \sqrt{n}$. As a table, such can be seen in the execution trace;

Step	Program State $\mathcal{S}$	Condition	Action
0	$n = \perp, i = \perp, \texttt{isPrime} = \perp$	–	Start
1	$n = n$	–	Take input n
2	$i = 2, \texttt{isPrime} = 1$	$n > 1$	Initialize loop
3	$i = i$	$n \equiv 0 \pmod{i}$? false	$i \mapsto i+1$
4	$i = i$	$n \equiv 0 \pmod{i}$? false	$i \mapsto i+1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathcal{S}_{final}$	and so on	$i^2 > n$	$\texttt{isPrime} = 1 \rightarrow$ print "prime"

At a glance;

Execution:

$$\begin{aligned} \mathcal{S}_0 = (\mathrm{n}, \mathrm{i}, \texttt{isPrime}) &= (\perp, \perp, \perp) \\ \text{input } n : \mathcal{S}_1 = (\mathrm{n}, \mathrm{i}, \texttt{isPrime}) &= (n, \perp, \perp) \\ \text{check } n \leq 1 &: \text{false, continue} \\ \text{initialize } i = 2,\ \text{isPrime} = 1 : \mathcal{S}_2 &= (n, i, \text{isPrime}) = (n, 2, 1) \\ \text{while } i^2 \leq n &: \text{evaluate divisibility } n \equiv 0 \pmod{i} \\ \text{if true} &\implies \text{print "not prime", stop} \\ \text{if false} &\implies i \mapsto i+1,\ \text{continue loop} \\ &\vdots \\ &\text{repeat loop until } i^2 > n \\ \text{no divisor found} &\implies \text{print "prime"} \\ \mathcal{S}_{\text{final}} = (\mathrm{n}, \mathrm{i}, \texttt{isPrime}) &= (n, i, 1) \end{aligned}$$

Java Code:

```java
package com.mycompany.hello;

import java.util.Scanner;

public class PrimeCheck {

    public static void main(String[] args) {
        // --- State S_0: variables uninitialized ---
        int n;          // input number
        int i;          // loop index
        boolean isPrime; // flag

        // --- Input: State S_1 ---
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a number: ");
        n = sc.nextInt();

        // --- Check n <= 1 ---
        if (n <= 1) {
            System.out.println("Not prime");
            return; // stop execution
        }

        // --- Initialize: State S_2 ---
        i = 2;
        isPrime = true;

        // --- While loop: State transitions ---
        while (i * i <= n) {
            if (n % i == 0) {
                // divisor found \rightarrow not prime
                isPrime = false;
                break; // stop loop
            }
            i = i + 1; // transition i \rightarrow i+1
        }

        // --- Final state: report result ---
        if (isPrime) {
            System.out.println("Prime");
        } else {
            System.out.println("Not prime");
        }
    }
}
```

We know that this program looks complicated now, but that is more of a reason to get started with JAVA. WHich we will be doing from now on, after the armstrong number thing... One then proceeds to see the case of the Narcissistic numbers, which is in a given number base b is a number that is the sum of its own digits each raised to the power of the number of digits.

Armstrong Numbers: An Armstrong number (also known as a narcissistic number, pluperfect digital invariant, or plus perfect number) is a number that is equal to the sum of its own digits each raised to the power of the number of digits. For example, 153 is an Armstrong number because it has three digits, and $1^3 + 5^3 + 3^3 = 153$. Similarly, 9474 is an Armstrong number because it has four digits, and $9^4 + 4^4 + 7^4 + 4^4 = 9474$. The concept can be generalized to any base, but it is most commonly discussed in base 10. Armstrong numbers are interesting in recreational mathematics and are often used in programming challenges and exercises.

We look at the formal definition; let $n \in \mathbb{Z}_{\geq 0}$ be a non-negative integer. Define the number of digits of n as

$$k = \lfloor \log_{10} n \rfloor + 1.$$

and, let the decimal expansion of n be;

$$n = \sum_{i=0}^{k-1} d_i \cdot 10^i, \quad d_i \in \{0, 1, \ldots, 9\}, \quad d_{k-1} \neq 0.$$

then we have an n that is called an **Armstrong number,** or **Narcissistic number**, or **pluperfect digital invariant (PPDI)**, or **plus perfect number**, if;

$$n = \sum_{i=0}^{k-1} d_i^k.$$

Equivalently let $D(n) = \{d_0, d_1, \ldots, d_{k-1}\}$ be the set of digits of n, then;

$$n \text{ is Armstrong} \iff n = \sum_{d \in D(n)} d^{\lfloor \log_{10} n \rfloor + 1}.$$

As example;

$$\begin{aligned} n &= 153, & k &= \lfloor \log_{10} 153 \rfloor + 1 = 3, & 1^3 + 5^3 + 3^3 &= 153, \\ n &= 9474, & k &= \lfloor \log_{10} 9474 \rfloor + 1 = 4, & 9^4 + 4^4 + 7^4 + 4^4 &= 9474. \end{aligned}$$

One has;

11. Armstrong Number

```
int $n$;
$k \leftarrow \lfloor \log_{10} n \rfloor + 1$
sum $\leftarrow 0$
temp $\leftarrow n$
while temp > 0
  digit $\leftarrow$ temp mod 10
  sum $\leftarrow$ sum + digit$^k$
  temp $\leftarrow \lfloor$temp/10$\rfloor$
end while
if sum = n
  print "Armstrong"
else
  print "Not Armstrong"
```

We model the program state as a mapping from variables to values;

$$\mathcal{S} : \{\mathrm{n}, \mathtt{temp}, \mathtt{sum}, \mathtt{digit}, k\} \to \mathbb{Z} \cup \{\perp\},$$

where $\perp$ means the variable is uninitialized. Initially one has;

$$\mathcal{S}_0(\mathrm{n}) = \mathcal{S}_0(\mathtt{temp}) = \mathcal{S}_0(\mathtt{sum}) = \mathcal{S}_0(\mathtt{digit}) = \mathcal{S}_0(k) = \perp$$

Taking input n;

$$\mathcal{S}_1 = \mathcal{S}_0[\mathrm{n} \mapsto n]$$

One computes the number of digits;

$$k = \lfloor \log_{10} n \rfloor + 1, \quad \mathcal{S}_2 = \mathcal{S}_1[k \mapsto k]$$

Initialize sum and temp;

$$\texttt{sum} \leftarrow 0, \quad \texttt{temp} \leftarrow n, \quad \mathcal{S}_3 = \mathcal{S}_2[\texttt{sum} \mapsto 0, \texttt{temp} \mapsto n]$$

While $\texttt{temp} > 0$:

$$\begin{aligned} \texttt{digit} &\leftarrow \texttt{temp} \bmod 10 \\ \texttt{sum} &\leftarrow \texttt{sum} + \texttt{digit}^k \\ \texttt{temp} &\leftarrow \lfloor \texttt{temp}/10 \rfloor \end{aligned}$$

Final check;

if $\texttt{sum} = n$ then print "Armstrong", else print "Not Armstrong"

Mathematically, the program computes $\texttt{sum} = \sum_{i=0}^{k-1} d_i^k$, where $d_0, \dots, d_{k-1}$ are the decimal digits of n, and checks whether $n = \sum_{i=0}^{k-1} d_i^k$. The final program state is;

$$\mathcal{S}_{\text{final}} = \{\text{n} \mapsto n,\ \texttt{temp} \mapsto 0,\ \texttt{sum} \mapsto \sum_{i=0}^{k-1} d_i^k,\ k \mapsto \lfloor \log_{10} n \rfloor + 1,\ \texttt{digit} \mapsto d_0 \text{ (last extracted digit)}\}.$$

Execution:

$$\begin{aligned} \mathcal{S}_0 &= (\text{n}, \texttt{temp}, \texttt{sum}, \texttt{digit}, k) = (\perp, \perp, \perp, \perp, \perp) \\ \text{input } n : \mathcal{S}_1 &= (\text{n}, \texttt{temp}, \texttt{sum}, \texttt{digit}, k) = (n, \perp, \perp, \perp, \perp) \\ \text{compute digits } k = \lfloor \log_{10} n \rfloor + 1 : \mathcal{S}_2 &= (n, \perp, \perp, \perp, k) \\ \text{initialize } \texttt{sum} = 0, \texttt{temp} = n : \mathcal{S}_3 &= (n, n, 0, \perp, k) \\ \text{while } \texttt{temp} > 0 &: \text{extract digit, update sum, reduce temp} \\ \text{iteration } \texttt{digit} &= \texttt{temp} \bmod 10, \texttt{sum} \mapsto \texttt{sum} + \texttt{digit}^k, \texttt{temp} \mapsto \lfloor \texttt{temp}/10 \rfloor \\ &\vdots \quad \text{repeat until } \texttt{temp} = 0 \\ \text{final check if } \texttt{sum} = n &\implies \text{print "Armstrong" else "Not Armstrong"} \\ \mathcal{S}_{\text{final}} &= (\text{n}, \texttt{temp}, \texttt{sum}, \texttt{digit}, k) = (n, 0, \sum_{i=0}^{k-1} d_i^k, d_0 \text{ (last extracted)}, k) \end{aligned}$$

We will study the code of Armstrong numbers later, for the time being let's get started with Java...

Java is a high-level, class-based, object-oriented programming language developed by James Gosling at Sun Microsystems. It is platform-independent, multithreaded, simple, secure, and object-oriented. Everything in Java is treated as a process; memory utilization is important for performance. Knowledge of C++ can help, but Java simplifies many concepts, such as inheritance, function overloading, and memory management.

A typical Java program consists of `public`, `class`, and `main`. Classes define blueprints for objects, `static` belongs to the class rather than an object, and `void` indicates no return value. The Java runtime involves the **JDK**, **JRE**, and **JVM**, converting human-readable code to machine-executable instructions. The compiler produces **bytecode**, which the JVM interprets at runtime, handling memory allocation for static variables, heap objects, and the stack for local variables.
Mathematical Perspective: Java programs can be viewed as transformations on program states;

$$\text{Program} : \text{Input} \xrightarrow{\text{Bytecode + JVM}} \text{Output}$$

Variables are elements of sets, e.g., integers $n \in \mathbb{Z}$ and booleans $b \in \{0, 1\}$. Methods are functions $f : \text{Inputs} \to \text{Outputs}$. Conditional statements are piecewise functions;

$$f(x) = \begin{cases} g(x), & \text{if condition is true} \\ h(x), & \text{if condition is false} \end{cases}$$

Loops represent iterative processes;

$$\texttt{for(i=0;i<n;i++) sum+=i} \implies \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Objects can be represented as mappings from identifiers to values;

$$\mathcal{M} : \text{Variables} \to \text{Values}$$

Parallel threads are independent functions;

$$\{f_1(x), f_2(y), \ldots, f_k(z)\} \text{ executing concurrently}$$

Finally, the JVM itself is a function;

$$\text{JVM} : \mathcal{B} \to \mathcal{E}, \quad \mathcal{B} = \text{Bytecode}, \ \mathcal{E} = \text{Execution state} + \text{Output}$$

This framework allows us to **reason mathematically about program correctness, complexity, and memory usage.** When a Java program runs, it interacts with the computer's hardware and memory through the **Java Virtual Machine (JVM)**. The program executes instructions step by step, which are ultimately carried out by the CPU. During runtime, the program allocates memory for variables, objects, and class data, but it can only access memory assigned to it by the JVM. This ensures that it cannot interfere with other programs or the system.

The program can interact with the user via input/output interfaces such as the terminal or GUI, allowing the user to provide input and receive output. From the system's perspective, the program temporarily controls parts of the CPU and memory, performs computations, and produces results, all within a **sandboxed environment** managed by the JVM.

Security restrictions prevent the program from performing unsafe operations. The program has power only while it runs; once execution ends, it no longer has control over the CPU or memory. Mathematically, a program can be thought of as a transformation on the system state. And thus one can model the entire execution as a function, and to strace what the program is doing, one can look at the state transitions; thinking the program to be a state machine. Funny, but somewhat fractal like structures. Anyway, we will not be needing that, but simply;

$$\mathcal{P} : \text{Initial System State} + \text{Input} \longrightarrow \text{Final System State} + \text{Output}$$

Each line of code updates the program state in a well-defined manner, ensuring that computations are deterministic and safe within the runtime environment. We look at a very simple program that executes a very basic task, and also is a complete java program.

Hello World

```java
package com.mycompany.hello;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

The program state space can be modeled as;

$$\mathcal{S} = \{\text{ClassLoader}, \text{Memory}, \text{Output}\} \times \mathbb{Z}$$

We then declare a `package`, which is a namespace that organizes classes and interfaces (`package com.mycompany.hello;`). The class declaration defines a blueprint for objects. The `main` method is the entry point of the program, where execution begins. The statement inside the main method prints "Hello, World!" to the console. This defines the namespace hierarchy. Mathematically, it creates a mapping;

$$\mathcal{P} : \text{ClassNames} \to \text{Namespace}$$

`public class Hello` is where we start defining the class. The keyword `public` is an access modifier that allows the class to be accessible from other classes. The class name `Hello` is an identifier for the class. This defines a mapping from the class name to its definition;

$$\text{Hello} = \{m : m \in \text{Methods}\} \cup \{f : f \in \text{Fields}\}$$

`public static void main(String[] args)` is the main method declaration. `public` means it can be accessed from outside the class. `static` means it belongs to the class rather than an instance of the class. `void` means it does not return any value. `main` is the name of the method, and `String[] args` is an array of strings that can be passed as command-line arguments. This defines a function;

$$\text{main} : \text{String}^n \to \text{void}$$

And `System.out.println("Hello, World!");` is a statement that prints "Hello, World!" to the console. `System` is a built-in class that provides access to system resources. `out` is a static field of the `System` class that represents the standard output stream. `println` is a method of the `PrintStream` class (the type of `out`) that prints a string followed by a newline character. This defines a function;

$$\text{println} : \text{String} \to \text{Output} \times \{\text{newline}\}$$

Now one sees that the program is a state machine, and the program state transitions as follows;

$$\mathcal{S}_0 \xrightarrow{\text{load}} \mathcal{S}_1 \xrightarrow{\text{execute}} \mathcal{S}_2 \xrightarrow{\text{output}} \mathcal{S}_3$$

And maybe it is a point where we should declare that we'd be taking the approach from the code perspective. Meaning, we see the code first, and then we try to understand what the code is doing. And then we try to see the mathematical perspective of the code. This is more of a practical approach, and also helps in understanding the code better. So, we will be following this approach from now on. And thus, we see the program to be a state machine. We see;

Execution: Hello World

```
𝒮₀ = (ClassLoader=⊥, Memory=⊥, Output=⊥)

load class => 𝒮₁ = (ClassLoader=Hello loaded, Memory=⊥, Output=⊥)

invoke main => 𝒮₂ = (ClassLoader=Hello loaded, Memory=Stack+Heap
    initialized, Output=⊥)

execute println => 𝒮₃ = (ClassLoader=Hello loaded, Memory=Stack updated,
    Output="Hello, World!")

halt => 𝒮₄ = (ClassLoader=Program terminated, Memory released, Output="
    Hello, World!")
```

If it not obvious by now, but all programs can be represented like this, regardless of languages. Specifying the language is just a matter of syntax. The semantics is what matters. And we will be following this approach from now on. We will see the code first, and then we will try to understand what the code is doing. And then we will try to see the mathematical perspective of the code. This is more of a practical approach, and also helps in understanding the code better. One can follow a different language, but one should think that it does not matter anyway. Because, maybe to some, syntaxes are barriers to understanding. Well, such is a difference between coding, and programming.

code 01: Hello World

```java
public class code_01 {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Execution Trace:

$$\begin{aligned}
\mathcal{S}_0 &= \{\text{Program starts}, \text{args} = \text{String}[0]\} \\
\text{Execute: } & \texttt{System.out.println("Hello, World!")} \\
&\Rightarrow \text{Evaluate argument: "Hello, World!"} \\
&\Rightarrow \text{Print to standard output} \\
\mathcal{S}_1 &= \{\text{Console output} = \text{"Hello, World!"}\} \\
\text{End of main method} & \\
\mathcal{S}_2 &= \{\text{Program terminates}\}
\end{aligned}$$

Program Analysis:

Step	Line	Operation	System State
1	2	`main()` invoked	$\mathcal{S}_0$: args = [], no output
2	3	`println()` called	Stack: println("Hello, World!")
3	3	String evaluated	"Hello, World!" computed (no variables)
4	3	Output operation	$\mathcal{S}_1$: Console gets "Hello, World!"
5	4	End of main	$\mathcal{S}_2$: Program terminates