4

Big Idea 3: Algorithms and Programming

"When you come to a fork in the road, take it."

—Yogi Berra

Chapter Goals

- Abstractions
- Variables
- Mathematical operators
- Operator precedence
- Modulus
- Assignment operators
- Display operators
- Input operators
- Relational and Boolean operators
- Robot
- Swap
- Searching
- Design steps
- Documentation
- Flowcharts
- List abstraction
- Traversing
- Procedures

What Is an Abstraction?

In computer science, an abstraction is a way to represent essential features without including the background details or explanations. Abstractions reduce complexity and allow for efficient design and implementation of complex software systems. Abstractions become a necessity as systems become more complex. For example, anytime you check your stories on Instagram, you are using a bunch of processes in the background that you have no control over. Without these abstractions, it would be difficult to send a message to a friend. With the use of abstractions, you can focus on content, not the technical details of how the application works.

Programmers also use abstractions. The purpose of abstraction is to hide coding details so the programmer can focus on the current problem. Computers can understand only binary machine code. Machine code is a strictly numerical language that runs fast but is hard to use.

For example, the following code written in machine code (you are NOT required to code in machine code for this AP exam) outputs the words "Hello World" to the screen.

```
 2  _10111000 _00100001 _00001010 _00000000 _00000000
 3  _10100011 _00001100 _00010000 _00000000 _00000110
 4  _10111000 _01101111 _01110010 _01101100 _01100100
 5  _10100011 _00001000 _00010000 _00000000 _00000110
 6  _10111000 _01101111 _00101100 _00100000 _01010111
 7  _10100011 _00000100 _00010000 _00000000 _00000110
 8  _10111000 _01001000 _01100101 _01101100 _01101100
 9  _10100011 _00000000 _00010000 _00000000 _00000110
10  _10111001 _00000000 _00010000 _00000000 _00000110
11  _10111010 _00010000 _00000000 _00000000 _00000000
12  _10111011 _00000001 _00000000 _00000000 _00000000
13  _10111000 _00000100 _00000000 _00000000 _00000000
14  _11001101 _10000000
15
16  _10111000 _00000001 _00000000 _00000000 _00000000
17  _11001101 _10000000
```

In SNAP! The same code can be written using the "say" abstraction.

In JAVA, the same code can be written using the System.out.print abstraction:

```
System.out.print("Hello World");
```

The SNAP! "say" abstraction and JAVA's System.out.print abstraction are both converted into machine code before a computer executes the command. This conversion from SNAP! or JAVA to machine code takes time but is significantly easier to code and debug. By using the abstractions, the programmer does not need to worry about the underlining code and can focus instead on the current problem that needs to be solved.

Abstractions allow for programmers to use semihuman language to program. The first language to use abstractions instead of machine language was COBOL, which was designed by Grace Hopper. Rarely will programmers deal directly in machine code. Machine code is a base language where no abstractions are implemented.

Programmers have worked to hide details by using abstractions. This allows the user to focus on the problem. Think of it as comparable to driving. A user's job is to navigate a course at speed. Abstraction keeps the details "under the hood" so that the "driver can drive." The driver has his or her job and does not need to worry about what makes the engine run; abstractions allow the machine to run and the driver or user to navigate his or her challenge.

Variables

Variables vary. The assignment operator allows a program to change the value represented by a variable.

Example

```
score ← 7 // This sets the value of score to 7.
score ← score + 3 // The new value of score is 10.
```

The value stored in "score" will be the most recent value assigned. In the above example, the value of score changed from 7 to 10.

Meaningful variable names help with the readability of program code and reduce the level of complexity of a program. For example, if a variable contains the score in a game, a good name for the variable could be "score". An example of a bad variable name that contains a score could be "*x*". The name "*x*" does not give a clue as to what the variable contains.

score ← 7 // score is a good name for a variable that will contain a score.

x ← 7 // *x* is a bad name because a programmer can forget what value *x* can hold, which can lead to errors.

Abstraction Examples Used on the AP Exam

Different program languages offer different levels of abstractions. High-level programming languages provide more abstractions than do lower-level languages. Coding in a programming language is often translated into code in another low-level language that the computer can execute.

Text: `DISPLAY (expression)` Block: `DISPLAY` [`expression`]	Displays the value of `expression`, followed by a space.

DISPLAY(expression) is an abstraction that is used on your AP exam to display a value of expression followed by a space. The input parameter for the DISPLAY abstraction is expression.

Text: `RANDOM (a, b)` Block: `RANDOM` [`a, b`]	Evaluates to a random integer from `a` to `b`, including `a` and `b`. For example, `RANDOM (1,3)` could evaluate to `1`, `2`, or `3`.

Another abstraction used on your AP exam is RANDOM(*a*, *b*), which evaluates to a random number from *a* to *b* inclusive. The input parameters in this abstraction are *a* and *b*.

Both abstractions were coded once but can be used an unlimited number of times without rewriting the code. This code reuse saves time and can prevent errors when writing code.

An abstraction generalizes functionality with input parameters that allow software reuse. Being aware of and using multiple levels of abstractions in developing programs helps to apply available resources and tools more effectively to solve problems.

Mathematical Operators

Operator	Meaning	Example
+	Addition	5 + 7 = 12
−	Subtraction	2 − 1 = 1
*	Multiplication	3 * 3 = 9
/	Division	3/2 = 1.5
MOD	Modulus	3 MOD 2 = 1

Operator Precedence (Order of Operations)

First: Parentheses
Second: MOD, *, /
Third: +, −

Example One

3 / 2 = 1.5
As there is only one operation in this example, simply divide 3 by 2 for a final answer of 1.5.

Example Two

6 + 3 * 5 = 21
In accordance with the order of operations, first multiply 3 by 5 to get a product of 15. Then add 6 to 15 for a final answer of 21.

Example Three

8 / 2 * 10 = 40
When operators have the same precedence, the equation should be evaluated from left to right. For instance, this equation contains both division and multiplication. In this case since the division is the first operator encountered when reading the calculation from left to right, first divide 8 by 2 to get a quotient of 4. Then multiply 4 by 10 for a final answer of 40.

Example Four

17 − 5 + 12 / 4 = 15

$\boxed{12}$ $\boxed{3}$

In this example, first divide 12 by 4 to get a quotient of 3. Then since addition and subtraction have the same precedence, evaluate the problem from left to right—subtract 5 from 17 for a difference of 12 and then add 3 for a final answer of 15.

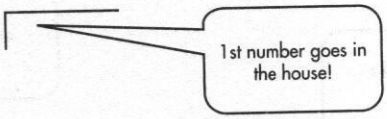
How to Solve Modulus

Although "modulus" (MOD) may be a new term for beginning programmers, the actual operation is one that should be familiar to those who know long division. A **modulus** is a mathematical operation that returns the remainder after an initial number (the **dividend**) is divided by another number (the **divisor**). Moduli are often used on your AP exam to determine whether a number is even or odd.

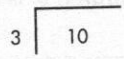
Example Five

10 MOD 3

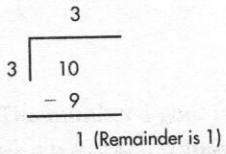
1. When solving, understanding which number is the dividend and which number is the divisor is crucial. It is important to note that when in a modulus operation, the number to the left of MOD is always the dividend and the number to the right of MOD is always the divisor. In this case, the dividend is 10 and the divisor is 3. A trick to remember this is to say, "Put the first number in the house."



2. Once you determine which number is the dividend and which is the divisor, set up the problem as a long-division equation. Make sure to place the dividend inside the long-division bracket (in the house) and the divisor outside the long-division bracket.



3. Then solve the long-division equation and find the integer remainder, which is the modulus answer.



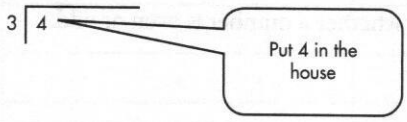
4. Thus, the answer to 10 MOD 3 = 1.

Notes

1. If the divisor is a multiple of the dividend, it will divide evenly with no remainder, resulting in a modulus calculation of 0.
   - 4 MOD 2 = 0
2. If the dividend is less than the divisor, the resulting modulus calculation will equal the value of the dividend.
   - 3 MOD 4 = 3
3. A zero to the right of MOD results in a DIVIDE BY ZERO error.
   - 6 MOD 0 = DIVIDE BY ZERO error
4. A zero to the left of MOD is feasible and results in a modulus calculation of 0.
   - 0 MOD 6 = 0

Example Six

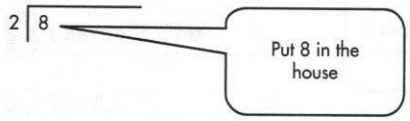
The modulus is the integer remainder when two numbers are divided.
4 MOD 3



The dividend is 4, and 3 is the divisor. The number 3 goes into 4 one time. Write a 1 on top, and multiply it by the divisor: 1 multiplied by 3 is 3. Then subtract 3 from 4. This gives 4 minus 3 equals 1, which is the remainder. The remainder, 1, is the answer.

Example Seven

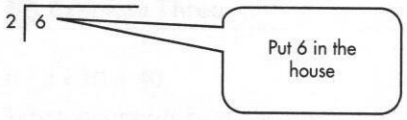
8 MOD 2



The dividend is 8, and 2 is the divisor. The number 2 goes into 8 four times. Write a 4 on top, and multiply it by the divisor: 4 multiplied by 2 is 8. Then subtract 8 from 8. This gives 8 minus 8 equals 0, which is the remainder. The remainder, 0, is the answer.

Example Eight

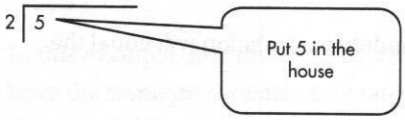
6 MOD 2



The dividend is 6, and 2 is the divisor. The number 2 goes into 6 three times. Write a 3 on top, and multiply it by the divisor: 2 multiplied by 3 is 6. Then subtract 6 from 6. This gives 6 minus 6 equals 0, which is the remainder. The remainder, 0, is the answer.

Example Nine

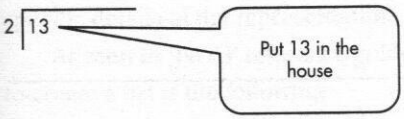
5 MOD 2



The dividend is 5, and 2 is the divisor. The number 2 goes into 5 two times. Write a 2 on top, and multiply it by the divisor: 2 multiplied by 2 is 4. Then subtract 4 from 5. This gives 5 minus 4 equals 1, which is the remainder. The remainder, 1, is the answer.

Example Ten

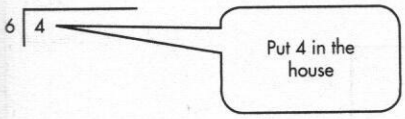
13 MOD 2



The dividend is 13, and 2 is the divisor. The number 2 goes into 13 six times. Write a 6 on top, and multiply it by the divisor: 6 multiplied by 2 is 12. Then subtract 12 from 13. This gives 13 minus 12 equals 1, which is the remainder. The remainder, 1, is the answer.

Example Eleven

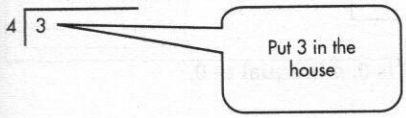
4 MOD 6



The dividend is 4, and 6 is the divisor. The number 6 goes into 4 zero times. Write a 0 on top, and multiply it by the divisor: 0 multiplied by 4 is 0. Then subtract 0 from 4. This gives 4 minus 0 equals 4, which is the remainder. The remainder, 4, is the answer.

Example Twelve

3 MOD 4



The dividend is 3, and 4 is the divisor. The number 4 goes into 3 zero times. Write a 0 on top, and multiply it by the divisor: 0 multiplied by 4 is 0. Then subtract 0 from 3. This gives 3 minus 0 equals 3, which is the remainder. The remainder, 3, is the answer.

Assignment Operators

Text: `a ← expression` Block: `a ← expression`	Evaluates `expression` and assigns the result to the variable *a*.

Note

First the expression is evaluated. Then the variable is set to the value that was calculated for the expression.

Example Thirteen

What is the value of *a* after the expression is evaluated?

a ← 3	*a* = 3

a is equal to 3.

Example Fourteen

What is the value of *a* after the expression is evaluated?

a ← 4 * 5 + 6	*a* = 26

First multiply 4 by 5 to get 20, and then add 6 to get 26.

Example Fifteen

What is the value of *a* after the expression is evaluated?

a ← 26 MOD 2	*a* = 0

Since 2 goes into 26 a total of 13 times and since 26 minus 26 is 0, *a* is equal to 0.

Example Sixteen

What is the value of *a* after the expression is evaluated?

a ← 17 MOD 2	*a* = 1

Since 2 goes into 17 a total of 8 times and since 16 minus 17 is 1, *a* is equal to 1.

Example Seventeen

What is the value of *a* after the expression is evaluated?

a ← 33 MOD 1	*a* = 0

Since 1 goes into 33 a total of 33 times and since 33 minus 33 is 0, *a* is equal to 0.

Lists

A list is an ordered sequence of elements starting with the index 1. A list is a data abstraction that reduces the complexity in programs by giving a collection of data a name without referencing the specific details of the representation.

As seen in the AP reference guide (provided to students on the AP exam), the correct format to create a list is the following:

Text:
`aList ← [ value1, value2, value3...]`
 	
Block:
`aList ← [value1, value2, value3]`
	Creates a new list where the `aList[1] = value1, aList[2] = value2` **and** `aList[3] = value3`.

Text: `aList ← [ ]` **Block:** `aList ← [ ]`	**Creates a new empty list and assigns it to the name** `aList`.
Text: `aList ← bList` **Block:** `aList ← bList`	**Assigns a copy of the** `bList` **to** `aList`.

Example Eighteen

```
namesOfMyDogs ← [ "Waffles", "Novack the 3rd", "Benji"]
                index     1            2            3
```

An element is an individual value in the list that is assigned a unique index. For all list operations, if a list index is less than 1 or greater than the length of the list, an error message is produced and the program will terminate.

Example Nineteen

namesOfMyDogs ← ["Waffles", "Novack the 3rd", "Benji"]
newList = namesOfMyDogs

```
a ← newList [0]
```

This code causes an error. This index will be out of bounds since newList has only the indexes 1, 2, and 3.

```
a ← newList [4]
```

This code causes an error. This index will be out of bounds since newList has only the indexes 1, 2, and 3.

Display Operators

Text:
`DISPLAY(expression)`
Block:
`DISPLAY [expression]`
	Displays the value of `expression`, followed by a space

Example Twenty

What will the following program display?

```
a ← 3
b ← 17
a ← b
DISPLAY(a)
```

As shown by the following chart, all variables must be initialized before *a* can be printed. Note that *a* is initialized to the value of 3 and *b* is set to 17. The next step sets the value of *a* to the value of *b*, which is 17.

a	*b*
~~3~~	17
17	

Answer: 17

Example Twenty-One

What will the following program display?

```
a ← 3
b ← 14
c ← 5
a ← c
b ← a
DISPLAY(a)
DISPLAY(b)
DISPLAY(c)
```

Answer: 5 5 5

a	b	c
~~3~~	~~14~~	5
5	5	

Example Twenty-Two

What will the following program display?

```
a ← 5 + 4 * 2
b ← 4 MOD 5
DISPLAY a + b
```

a	b	DISPLAY
5 + 8 = 13	4 MOD 5 = 4	17

Answer: 17

Input Operators

Text:	Accepts a value from the user and returns it.
`INPUT()`	
Block: `INPUT`	

Example Twenty-Three

What will the following program display if the INPUT function reads an even number such as 4?

```
a ← INPUT()
a ← a MOD 2
DISPLAY(a)
```

For 4 (or any even number) divided by 2, the remainder will always be 0.

Answer: 0

Example Twenty-Four

What will the following program display if the INPUT function reads an odd number such as 5?

```
a ← INPUT()
a ← a MOD 2
DISPLAY(a)
```

An odd number will always have a remainder of 1 when divided by 2.

Answer: 1

Relational and Boolean Operators

Text and block:	The relational operators =, ≠, >, <, ≥, and ≤ are used to test the relationship between two variables< expressions< or values>
$a = b$	$a = b$ evaluates to true if a and b are equal; otherwise, it evaluates to false. For example, when a is 3 and b is 3, $a = b$ evaluates to true. However, when a is 3 and b is 5, $a = b$ evaluates to false.
$a \neq b$	$a \neq b$ evaluates to true if a and b are not equal; otherwise, it evaluates to false.
$a > b$	$a > b$ evaluates to true if a is greater than b; otherwise, it evaluates to false.
$a < b$	$a < b$ evaluates to true if a is less than b; otherwise, it evaluates to false.
$a \geq b$	$a \geq b$ evaluates to true if a is greater than or equal to b; otherwise, it evaluates to false.
$a \leq b$	$a \leq b$ evaluates to true if a is less than or equal to b; otherwise, it evaluates to false.

Example Twenty-Five

What will the following program display?

```
a ← 3
b ← 3 MOD 5
DISPLAY a = b
```

a	b	DISPLAY
3	3 MOD 5 = 3	true

a is initialized to 3; b is initialized to the remainder when 3 is divided by 5. Since 3 is equal to 3, the program will display true.

Answer: True

Example Twenty-Six

What will the following program display?

```
a ← 4 * (3 + 6)
b ← 4 * 3 + 6
DISPLAY (a ≤ b)
```

a	*b*	DISPLAY
36	18	false

Using the order of operations, *a* will be initialized to 36 since the addition occurs first due to the parentheses. Note that *b* will be initialized to 18 since the multiplication occurs first due to operator precedence. Because 36 is not less than or equal to 18, this program displays false.

Answer: False

Numeric Procedures

Text:
`RANDOM(a, b)`
 	
Block:
`RANDOM [a, b]`
	Evaluates to a random integer from `a` to `b`, including `a` and `b`.
	
	For example, `RANDOM(1, 3)` could evaluate to 1, 2, or 3.

Example Twenty-Seven

If the below code was executed several times, what is the percentage of times "true" would be expected to be displayed?

```
a ← [RANDOM [1,2]]
[DISPLAY [a = 1]]
```

RANDOM can pick 1 or 2. The chance of 1 being picked is 1/2 or 50%.

Answer: 50%

Example Twenty-Eight

If the below code was executed several times, what is the percentage of times "false" would be expected to be displayed?

```
a ← [RANDOM [1,4]]
[DISPLAY [a = 3]]
```

The numbers 1, 2, 3, and 4 are all possible selections. The chance of 3 being selected is 1/4 or 25%. The chance of 3 not being selected is 3/4, which is 75%.

Answer: 75%

Example Twenty-Nine

If the below code was executed several times, what is the percentage of times "true" would be expected to be displayed?

```
a ← RANDOM 1,10
DISPLAY a ≤ 3
```

The numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 can all be selected. Since $a \leq 3$ includes 1, 2, and 3, the answer is $3/10 = 30\%$.

Answer: 30%

Boolean Tables

Instruction	Explanation
Text: `NOT condition` Block: `NOT condition`	Evaluates to "true" if condition is false; otherwise evaluates to "false."
Text: `condition1 AND condition2` Block: `condition1 AND condition2`	Evaluates to "true" if both condition1 and condition2 are true; otherwise, evaluates to "false."
Text: `condition1 OR condition2` Block: `condition1 OR condition2`	Evaluates to "true" if either condition1 is true or condition2 is true or if both condition1 and condition2 are true; otherwise, evaluates to "false."

A Boolean value is a data type that has only two possible values, true or false. On your AP exam, you will be tested on three Boolean operators. Those operators are AND, OR, and NOT.

Boolean questions simplify if you can use noncomputer science terms. For example, if you are a picky eater and you will be happy only if you have both of your favorite foods (Pizza AND Yams), ask yourself if you are happy in the following conditions.

Happy with....	Pizza AND Yams = Happy
	True AND True = True
Happy with....	Pizza AND Peanuts = Unhappy
	True AND False = False
Happy with....	Peas AND Peanuts = Unhappy
	False AND False = False

Happy with.... Tuna AND Pizza = Unhappy
False AND True = False

Actually
Happy with.... Tuna AND Anything in the world = Unhappy
False AND Anything in the world = False

Now let's assume that you are not a picky eater and you will be happy if you have only one of your favorite foods (Pizza OR Yams). Ask yourself are you happy in the following conditions.

Happy with.... Pizza OR Yams = Happy
True OR True = True
Happy with.... Pizza OR Peanuts = Happy
True OR False = True
Happy with.... Peas OR Peanuts = Unhappy
False OR False = False
Happy with.... Tuna OR Pizza = Happy
False OR True = True

Actually
Happy with.... Pizza OR Anything in the world = Happy
False AND Anything in the world = True

The operator NOT reverses the Boolean value.

NOT Happy = Unhappy
NOT True = False

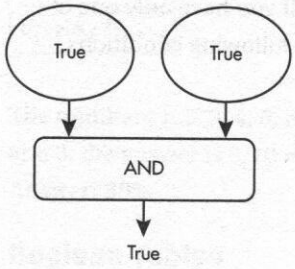
NOT Unhappy = Happy
NOT False = True

The above results are summarized in the Boolean table below.

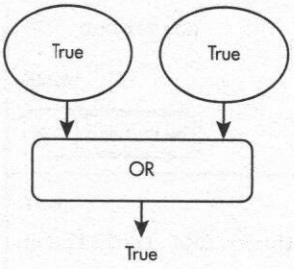
Condition One	Logical Operator	Condition Two	Evaluation	Result
True	AND	True	Both arguments are true.	True
True	AND	False	One argument is false.	False
False	OR	True	One argument is true.	True
False	OR	False	Both arguments are false.	False
False	AND	True or False	When evaluating an "AND" logical operation, if at least one condition is false . . .	False
True	OR	True or False	When evaluating an "OR" logical operation, if at least one condition is true . . .	True

Note that examples 30 to 37 each show one the two possible formats for how the AP exam will ask Boolean questions. It is the same information as shown in the Boolean table except drawn a different way.

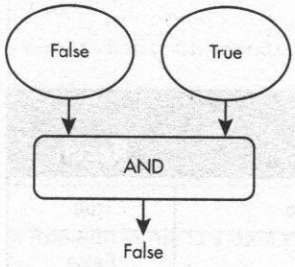
Example Thirty



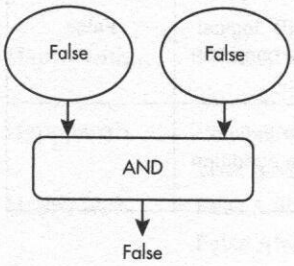
Example Thirty-One



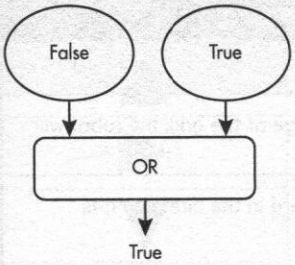
Example Thirty-Two



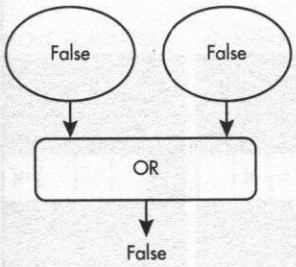
Example Thirty-Three



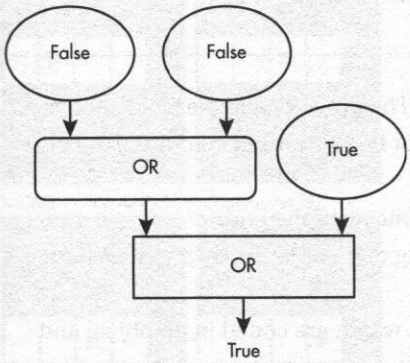
Example Thirty-Four



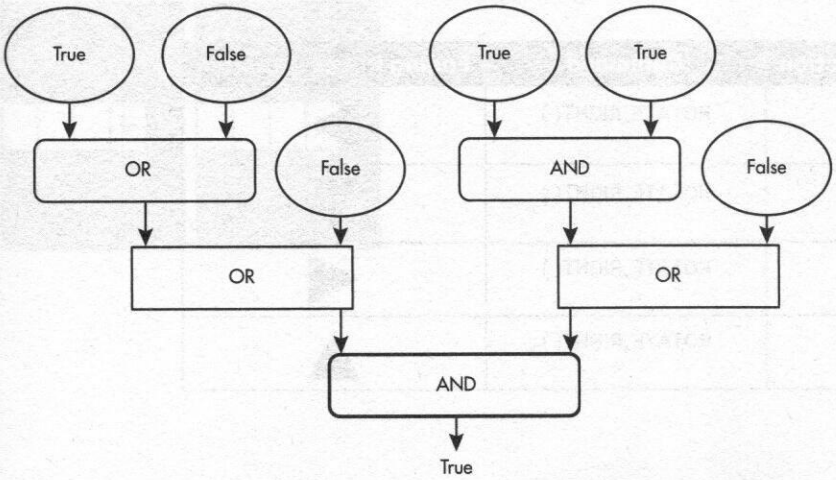
Example Thirty-Five



Example Thirty-Six



Example Thirty-Seven



The Robot

Instruction	Explanation
Robot	
If the robot attempts to move to a square that is not open or is beyond the edge of the grid, the robot will stay in its current location, and the program will terminate.	

Text:
`MOVE_FORWARD()`
Block:
`MOVE_FORWARD`
	The robot moves one square forward in the direction it is facing.

Text:
`ROTATE_LEFT()`
Block:
`ROTATE_LEFT`
	The robot rotates in place 90 degrees counterclockwise (i.e., makes an in-place left turn).

Text:
`ROTATE_RIGHT()`
Block:
`ROTATE_RIGHT`
	The robot rotates in place 90 degrees clockwise (i.e., makes an in-place right turn).

Text:
`CAN_MOVE(direction)`
Block:
`CAN_MOVE direction`
	Evaluates to `"true"` if there is an open square one square in the direction relative to where the robot is facing; otherwise evaluates to `"false"`. The value of direction can be `left`, `right`, `forward`, or `backward`.

The robot on the AP exam has both direction and location. The direction can be facing left, facing right, facing up, or facing down. The location of the robot is the row and column of the grid in which the robot is currently sitting.

The robot will end any program if the robot is asked to move off the grid.

The robot can change its direction and location by using the four listed abstractions in the table below.

The robot questions can contain procedures and loops, which are coded in graphical and text-based programming language.

Rotate the Robot

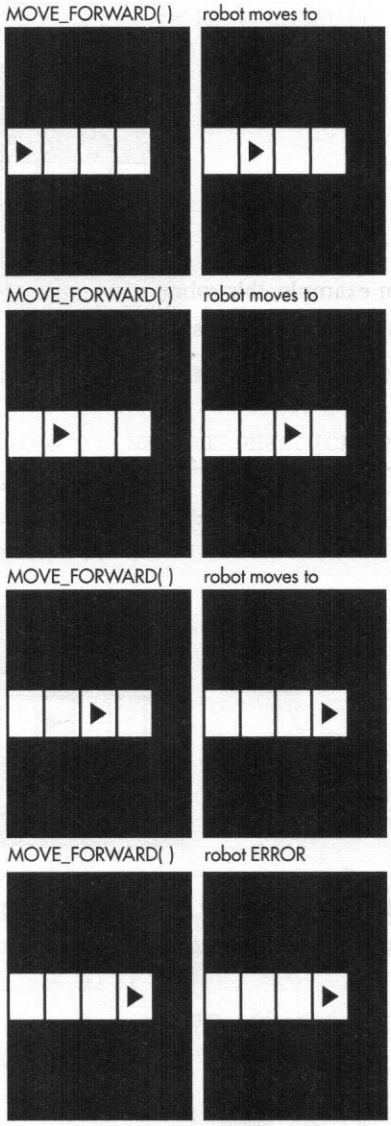
ROTATE_RIGHT will rotate the robot 90 degrees clockwise.

Initial Robot Direction	Command	Ending Robot Direction
▲	ROTATE_RIGHT( )	▶
▶	ROTATE_RIGHT( )	▼
▼	ROTATE_RIGHT( )	◀
◀	ROTATE_RIGHT( )	▲

ROTATE_LEFT will rotate the robot 90 degrees counterclockwise.

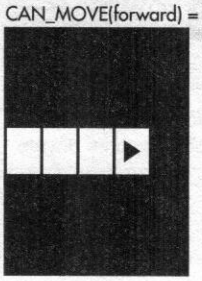
Initial Robot Direction	Command	Ending Robot Direction
▲	ROTATE_LEFT( )	◀
▶	ROTATE_LEFT( )	▲
▼	ROTATE_LEFT( )	▶
◀	ROTATE_LEFT( )	▼

Move the Robot Forward

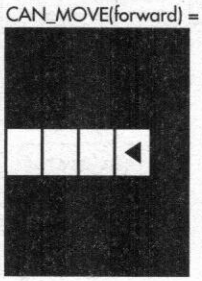


To prevent the robot from moving off the grid and resulting in an error, use the CAN_MOVE(direction) abstraction.

CAN_MOVE(forward) = FALSE



CAN_MOVE(forward) = TRUE

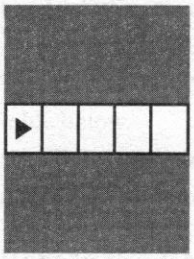


REPEAT_UNTIL Loop

Loops will repeat a section of code until a condition is met. For example, this robot in example thirty-eight will repeat lines 1, 2, 3 and 4 until the condition in line 1 executes as true. Once line 1 executes to "true," the program is over.

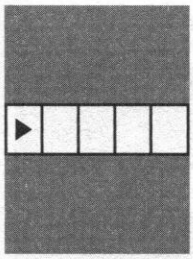
Example Thirty-Eight

Robot starting location and direction shown.



Robot code:

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

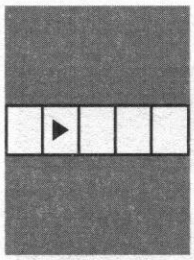


```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

The current robot can move forward so CAN_MOVE(forward) will equal true. To end the loop, the CAN_MOVE(forward) must equal "false."

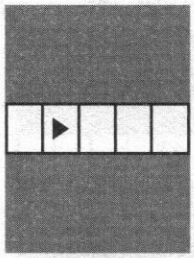
Enters loop at line 2.

```
Line 3:    MOVE_FORWARD()
```



Ends loop at line 4 and resets back to the testing condition on line 1.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```



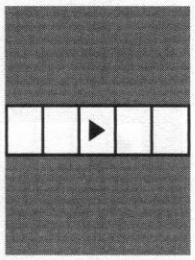
Back to line 1 and retests loop condition.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

The current robot can move forward so CAN_MOVE(forward) will equal true. To end the loop, the CAN_MOVE(forward) must equal "false."

Enters loop at line 2.

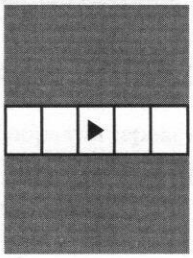
```
Line 3:    MOVE_FORWARD()
```



Ends loop at line 4 and resets back to the testing condition on line 1.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

Back to line 1 and retests loop condition.

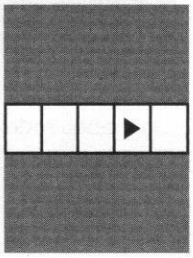


```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

The current robot can move forward so CAN_MOVE(forward) will equal true. To end the loop, the CAN_MOVE(forward) must equal "false."

Enters loop at line 2.

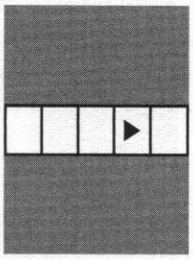
```
Line 3: MOVE_FORWARD()
```



Ends loop at line 4 and resets back to the testing condition on line 1.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

Back to line 1 and retests loop condition.

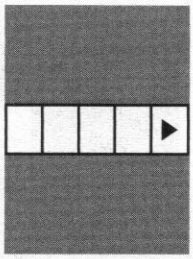


```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

The current robot can move forward so CAN_MOVE(forward) will equal true. To end the loop, the CAN_MOVE(forward) must equal "false."

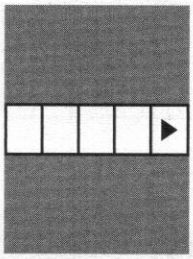
Enters loop at line 2.

```
Line 3:    MOVE_FORWARD()
```



Ends loop at line 4 and resets back to the testing condition on line 1.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```



Back to line 1 and retests loop condition.

```
Line 1: REPEAT_UNTIL(CAN_MOVE(forward) = false)
Line 2: {
Line 3:    MOVE_FORWARD()
Line 4: }
```

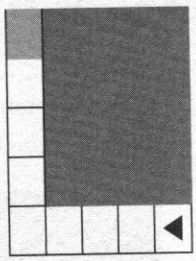
The current robot can move forward so CAN_MOVE(forward) will equal true. To end the loop, the CAN_MOVE(forward) must equal "false."

Example Thirty-Nine

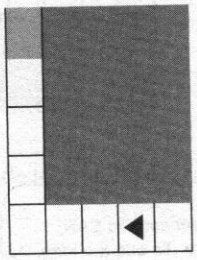
For the following grid, the program below is intended to move the robot to the gray square. The program uses the procedure Goal_Reached( ), which returns "true" if the robot is in the gray square and returns "false" otherwise.

```
REPEAT UNTIL(Goal_Reached())
{
  IF(CAN_MOVE(right))
  {
    ROTATE_RIGHT()
  }
  ELSE
  {
    IF(CAN_MOVE(left))
    {
    ROTATE_LEFT()
    }
  }
  IF(CAN_MOVE(forward))
  {
    MOVE_FORWARD()
    }
  }
}
```

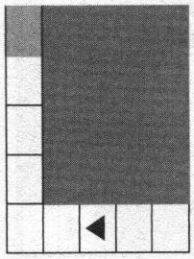
Starting Map



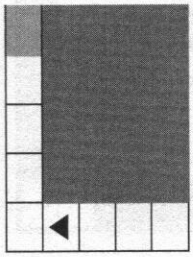
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



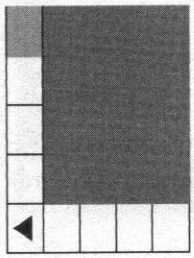
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



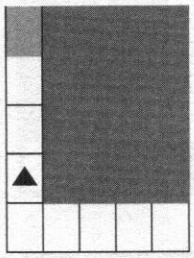
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



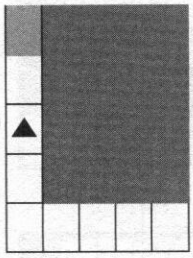
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



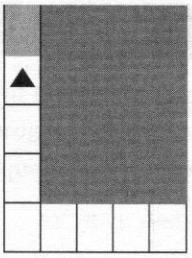
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	True	Turn_Right
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



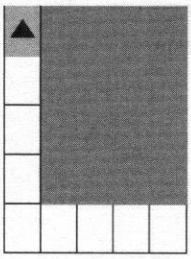
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward

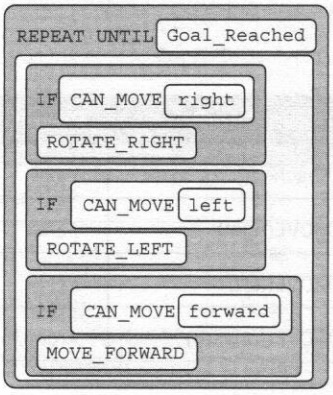


Boolean Condition	Result	Robot Moves
Goal_Reached()	True	Ends Procedure
CAN_MOVE(right)		
CAN_MOVE(left)		
CAN_MOVE(forward)		

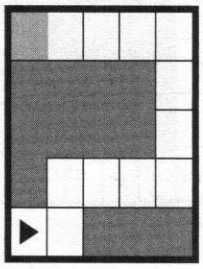
Example Forty

Note: Same code used as in example thirty-nine. The grid has changed.

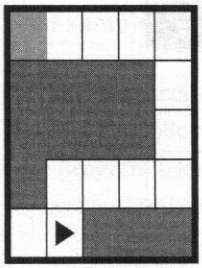
For the following grid, the program below is intended to move the robot to the gray square. The program uses the procedure Goal_Reached( ), which returns "true" if the robot is in the gray square and returns "false" otherwise.



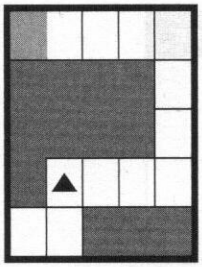
Starting Map



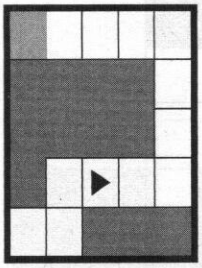
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



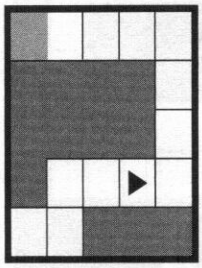
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	True	Turn_Left
CAN_MOVE(forward)	True	Forward



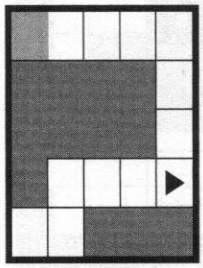
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	True	Turn_Right
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



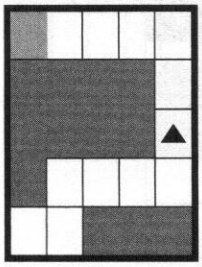
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



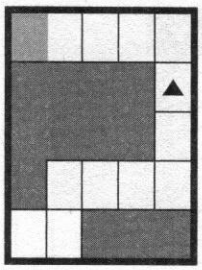
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



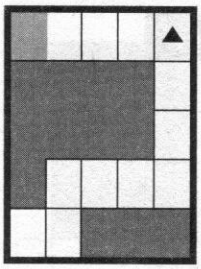
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	True	Turn_Left
CAN_MOVE(forward)	True	Forward



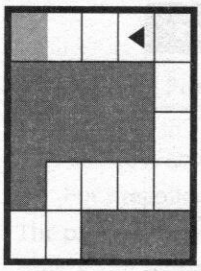
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



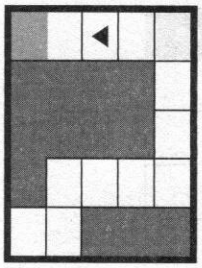
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



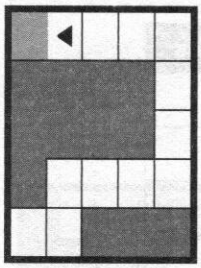
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	True	Turn_Left
CAN_MOVE(forward)	True	Forward



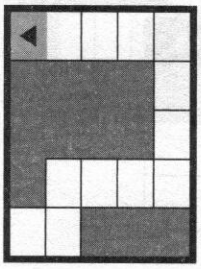
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	True	Ends Procedure
CAN_MOVE(right)		
CAN_MOVE(left)		
CAN_MOVE(forward)		

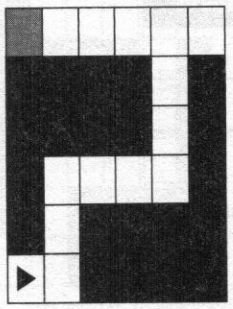
Example Forty-One

Note: Same code as example forty. The grid has changed.

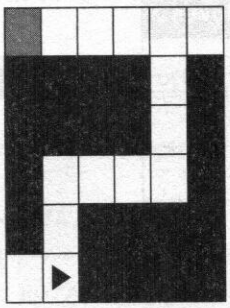
For the following grid, the program below is intended to move the robot to the gray square. The program uses the procedure Goal_Reached( ), which returns "true" if the robot is in the gray square and returns "false" otherwise.

```
REPEAT UNTIL Goal_Reached
    IF CAN_MOVE right
        ROTATE_RIGHT
    IF CAN_MOVE left
        ROTATE_LEFT
    IF CAN_MOVE forward
        MOVE_FORWARD
```

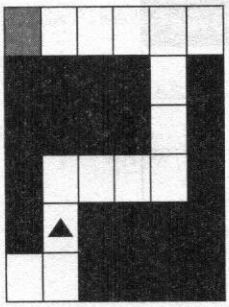
Starting Map



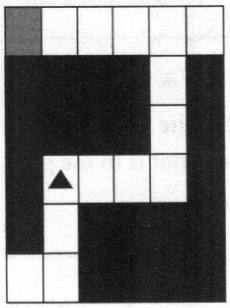
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



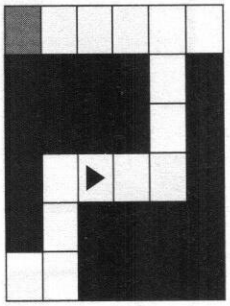
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	True	Turn_Left
CAN_MOVE(forward)	True	Forward



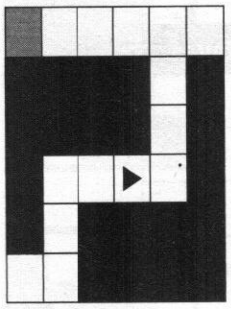
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



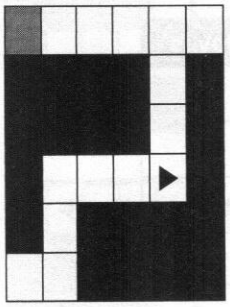
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	True	Turn_Right
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



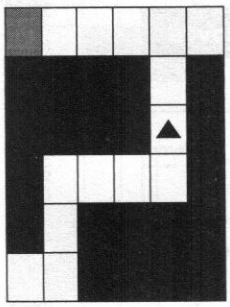
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



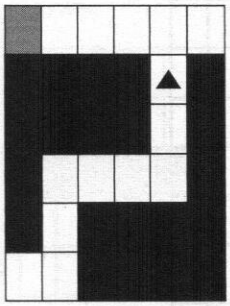
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



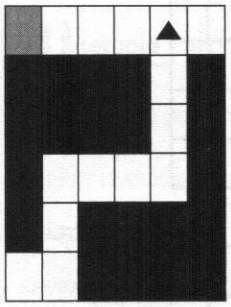
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	True	Turn_Left
CAN_MOVE(forward)	True	Forward



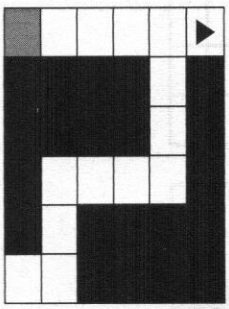
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	True	Turn_Right
CAN_MOVE(left)	False	
CAN_MOVE(forward)	True	Forward



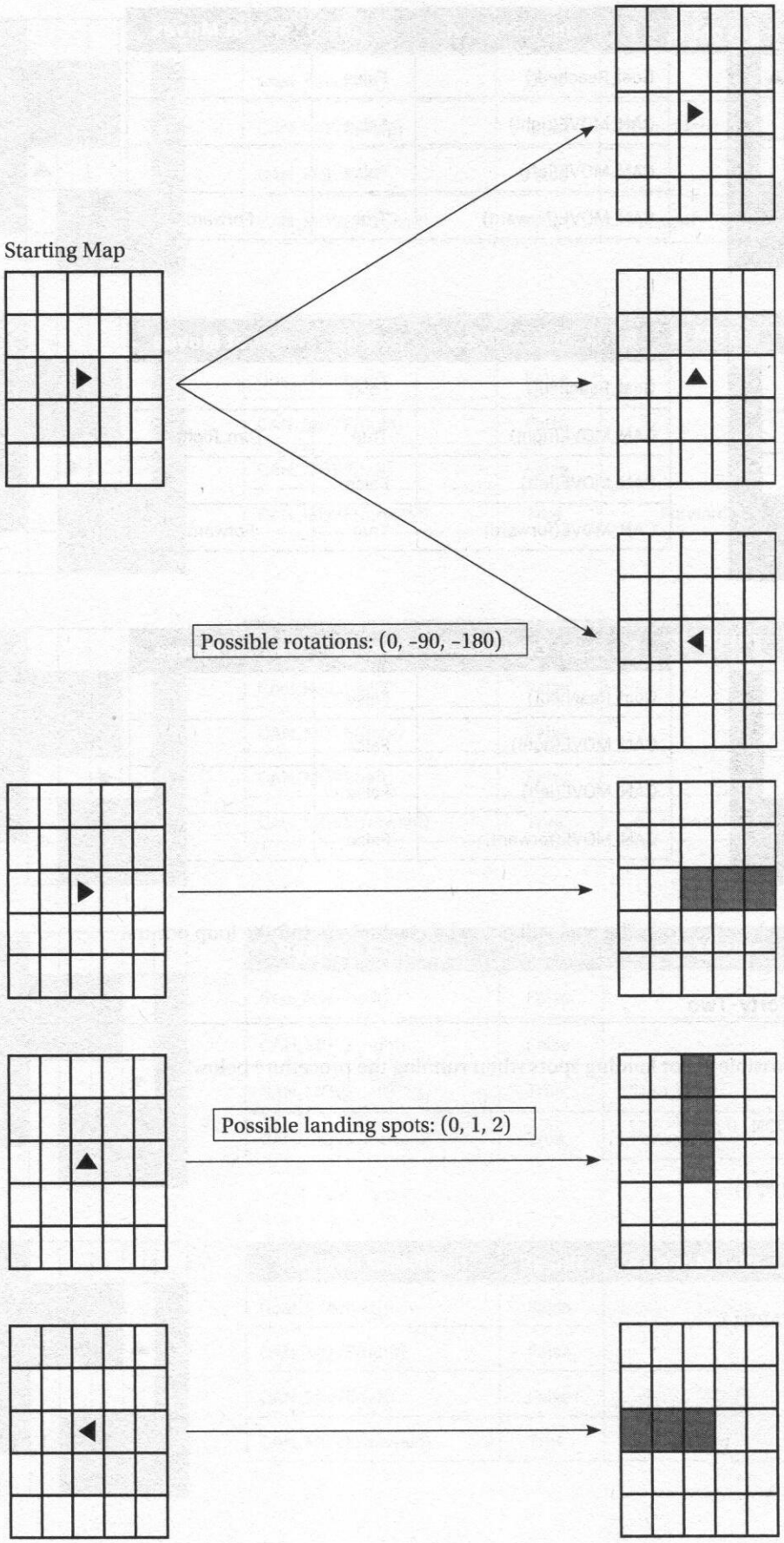
Boolean Condition	Result	Robot Moves
Goal_Reached()	False	
CAN_MOVE(right)	False	
CAN_MOVE(left)	False	
CAN_MOVE(forward)	False	

The robot is stuck in the grid. The goal will never be reached. An infinite loop occurs.

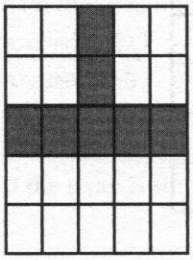
Example Forty-Two

What are the possible robot landing spots when running the procedure below?

```
REPEAT(RANDOM(0,2))
{
  ROTATE_LEFT()
}
REPEAT(RANDOM(0,2))
{
  MOVE_FORWARD()
}
```


Starting Map
Possible rotations: (0, -90, -180)
Possible landing spots: (0, 1, 2)

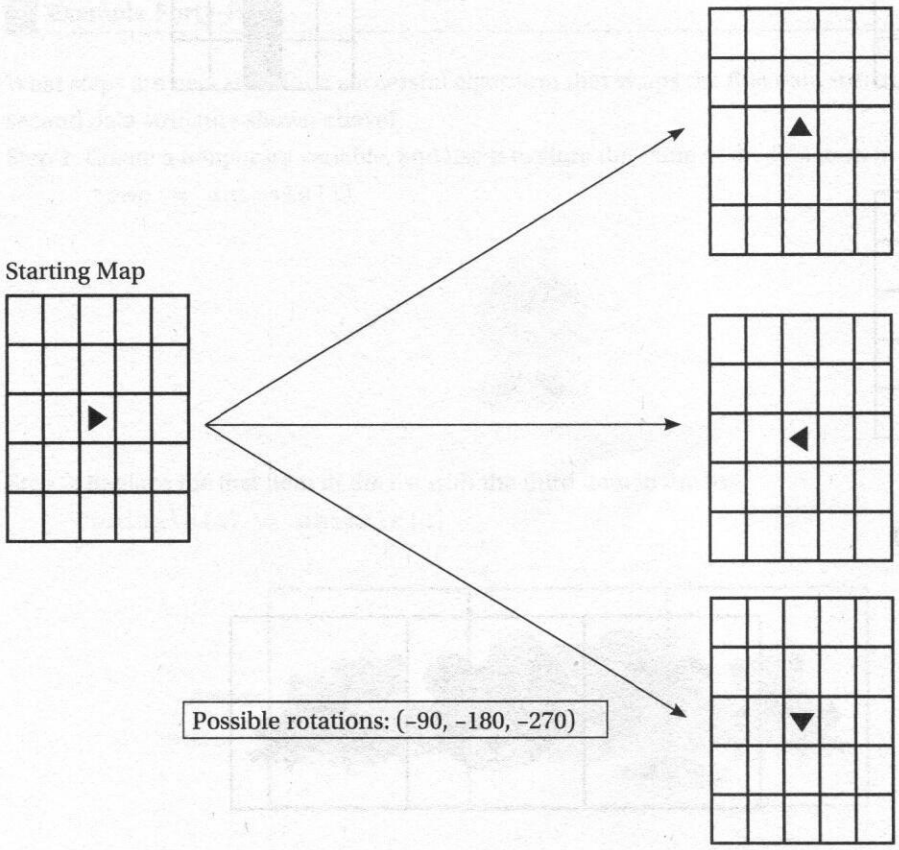
Answer:

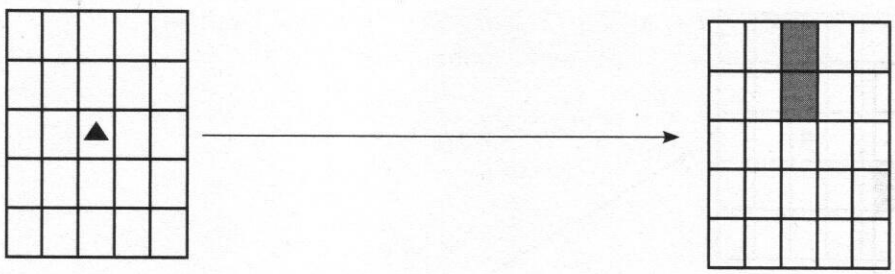


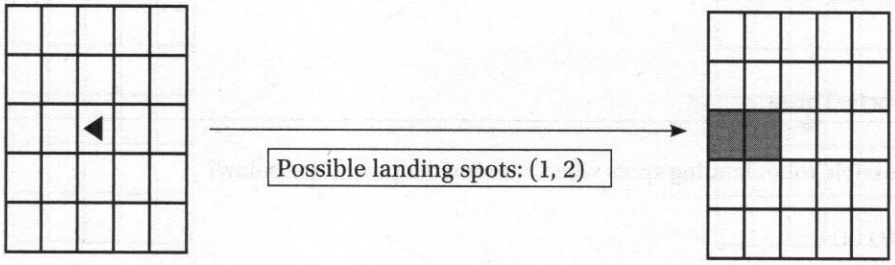
Example Forty-Three

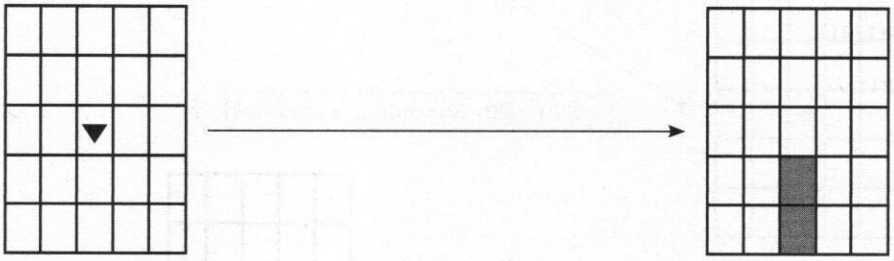
What are the possible robot landing spots when running the procedure below?

```
REPEAT(RANDOM(1,3))
{
  ROTATE_LEFT()
}
REPEAT(RANDOM(1,2))
{
  MOVE_FORWARD()
}
```

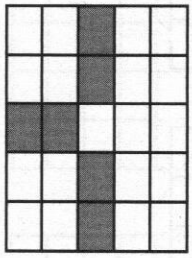






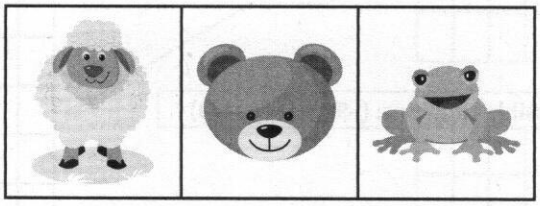


Answer:



The Swap

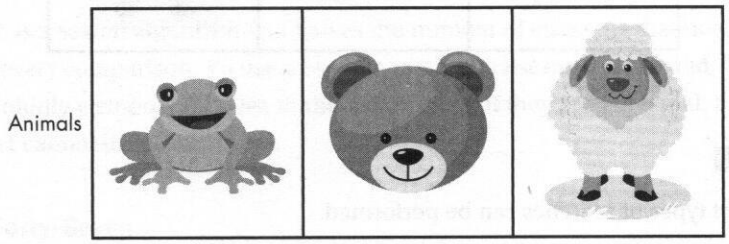
Animals



A common algorithm is the swap. In the above animal data structure, we want to swap the sheep with the frog. Currently, the list contents are the following:

```
animals[1] = sheep
animals[2] = bear
animals[3] = frog
```

If the swap is successful, the animals' data structure will swap the sheep with the frog.



After a successful swap, the list contents will be the following:

```
animals[1] = frog
animals[2] = bear
animals[3] = sheep
```

Example Forty-Four

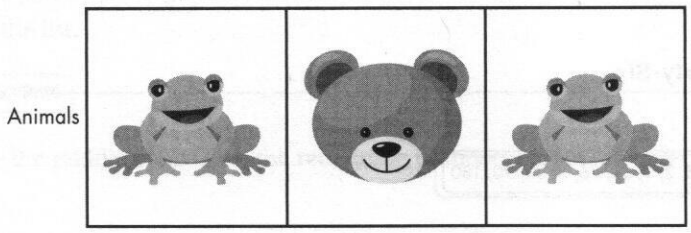
What steps are necessary for a successful algorithm that swaps the first data structure into the second data structure shown above?

Step 1: Create a temporary variable, and use it to store the value of the first item in the list.

```
temp = animals[1]
```

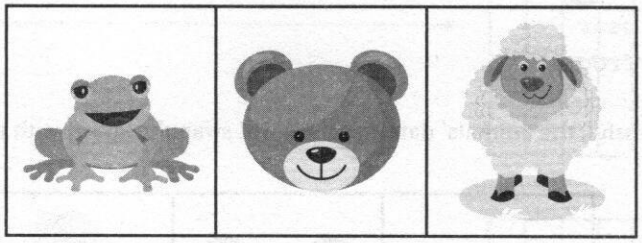
Step 2: Replace the first item in the list with the third item in the list.

```
animals[1] = animals[3]
```



Step 3: Replace the third item in the list with the item that was stored in the temporary variable.

```
animals[3] = temp
```



Searching

Several different types of searches can be performed.

Linear Search

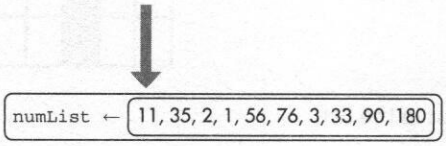
A linear search (or sequential search) is an algorithm for finding an element in a list. This search starts from the beginning of a list and sequentially checks each element of the list until a match is found or the entire list is searched without finding the element. A linear search can be used for either a **sorted list** or an **unsorted list**.

If a list has n elements, the worst case for the number of searches would be n. For example, if a list has 50 elements, the worst case would be 50 comparisons. However, the best case would be if the element you are looking for was found with the first comparison.

Example Forty-Five

numList ← [11, 35, 2, 1, 56, 76, 3, 33, 90, 180]

Using a linear search, how many comparisons would it take to find the number 11?

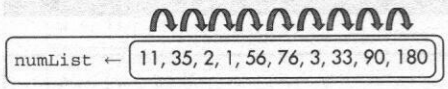


Answer: 1 comparison. This is the best case for a linear search.

Example Forty-Six

numList ← [11, 35, 2, 1, 56, 76, 3, 33, 90, 180]

Using a linear search, how many comparisons would it take to find the number 180?

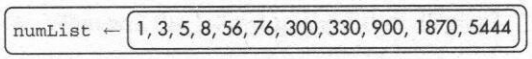


Answer: 10 comparisons. This is the worst case for a linear search of a list with 10 elements.

Binary Search

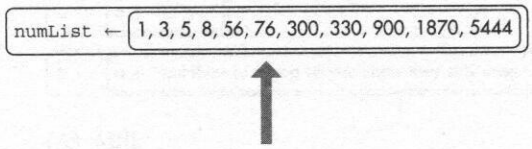
A binary search is a search algorithm that halves the number of elements that need to be searched after every comparison. To use a binary search, **the list must be sorted**. This search compares the middle element of the list to the target value. If they are not equal, the half in which the target cannot lie is eliminated.

Example Forty-Seven

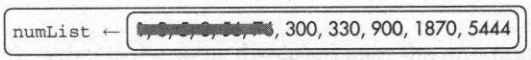


What steps are needed for a binary search to find the number 300?

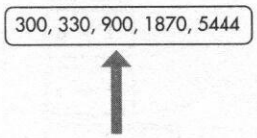
Step 1: Compare the middle element.



Since 76 is not equal to the target and 300 cannot be on the left side of the list, we throw out the left side of the list.

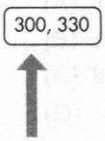


Step 2: Compare the middle element of the remaining numbers.



Since 900 is not equal to the target and 300 cannot be on the right side of the list, we throw out the right side of the list.

Step 3: Compare the middle element of the remaining numbers (round down).



300 = target