AP® Computer Science Principles Exam

SECTION I: Multiple-Choice Questions

DO NOT OPEN THIS BOOKLET UNTIL YOU ARE TOLD TO DO SO.

At a Glance

Total Time
2 hours
Number of Questions
70
Percent of Total Score
70%
Writing Instrument
Pencil required

Instructions

Section I of this examination contains 70 multiple-choice questions. Fill in only the ovals for numbers 1 through 70 on your answer sheet.

Indicate all of your answers to the multiple-choice questions on the answer sheet. No credit will be given for anything written in this exam booklet, but you may use the booklet for notes or scratch work. After you have decided which of the suggested answers is best, completely fill in the corresponding oval on the answer sheet. Give only one answer to each question. If you change an answer, be sure that the previous mark is erased completely. Here is a sample question and answer.

Sample Question

Chicago is a
(A) state
(B) city
(C) country
(D) continent
(E) county

Sample Answer

Use your time effectively, working as quickly as you can without losing accuracy. Do not spend too much time on any one question. Go on to other questions and come back to the ones you have not answered if you have time. It is not expected that everyone will know the answers to all the multiple-choice questions.

About Guessing

Many candidates wonder whether or not to guess the answers to questions about which they are not certain. Multiple-choice scores are based on the number of questions answered correctly. Points are not deducted for incorrect answers, and no points are awarded for unanswered questions. Because points are not deducted for incorrect answers, you are encouraged to answer all multiple-choice questions. On any questions you do not know the answer to, you should eliminate as many choices as you can, and then select the best answer among the remaining choices.

GO ON TO THE NEXT PAGE.

Quick Reference

Instruction	Explanation
Assignment, Display, and Input	
Text: `a ← expression` Block: 	Evaluates `expression` and then assigns a copy of the result to the variable `a`.
Text: `DISPLAY(expression)` Block: 	Displays the value of `expression`, followed by a space.

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

Arithmetic Operators and Numeric Procedures	

Text and Block:
`a + b`
`a - b`
`a * b`
`a / b`
	The arithmetic operators `+`, `-`, `*`, and `/` are used to perform arithmetic on `a` and `b`.
	For example, `17 / 5` evaluates to `3.4`.
	The order of operations used in mathematics applies when evaluating expressions.

Text and Block: `a MOD b`	Evaluates to the remainder when `a` is divided by `b`. Assume that `a` is an integer greater than or equal to `0` and `b` is an integer greater than `0`. For example, `17 MOD 5` evaluates to `2`. The `MOD` operator has the same precedence as the `*` and `/` operators.
Text: `RANDOM(a, b)` Block: 	Generates and returns a random integer from `a` to `b`, including `a` and `b`. Each result is equally likely to occur. For example, `RANDOM(1, 3)` could return `1`, `2`, or `3`.

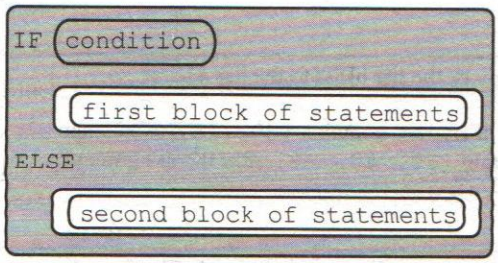
Instruction	Explanation
Relational and Boolean Operators	

Text and Block:
`a = b`
`a ≠ b`
`a > b`
`a < b`
`a ≥ b`
`a ≤ b`
	The relational operators =, ≠, >, <, ≤, and ≥ are used to test the relationship between two variables, expressions, or values. A comparison using relational operators evaluates to a Boolean value.
	
	For example, `a = b` evaluates to `true` if `a` and `b` are equal; otherwise it evaluates to `false`.

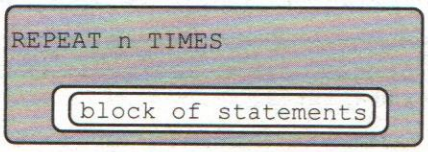
Text: `NOT condition` Block: NOT `condition`	Evaluates to `true` if `condition` is `false`; otherwise evaluates to `false`.
Text: `condition1` AND `condition2` Block: `condition1` AND `condition2`	Evaluates to `true` if both `condition1` and `condition2` are `true`; otherwise evaluates to `false`.
Text: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

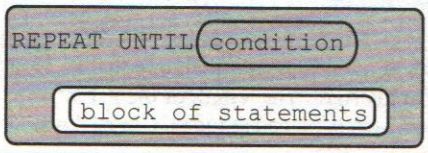
Text:
`IF (condition)`
`{`
`<block of statements>`
`}`
Block:
IF `condition`
`block of statements`
	The code in `block of statements` is executed if the Boolean expression `condition` evaluates to `true`; no action is taken if `condition` evaluates to `false`.

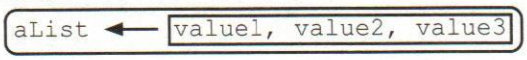
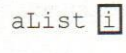
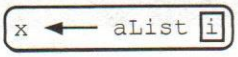
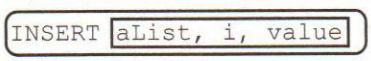
Instruction	Explanation
Selection—Continued	

Text:
`IF(condition)`
`{`
`<first block of statements>`
`}`
`ELSE`
`{`
`<second block of statements>`
`}`
Block:
 
	The code in `first block of statements` is executed if the Boolean expression `condition` evaluates to `true`; otherwise the code in `second block of statements` is executed.

Iteration	

Text:
`REPEAT n TIMES`
`{`
`<block of statements>`
`}`
Block:
 
	The code in `block of statements` is executed `n` times.

Text:
`REPEAT UNTIL(condition)`
`{`
`<block of statements>`
`}`
Block:
 
	The code in `block of statements` is repeated until the Boolean expression `condition` evaluates to `true`.

Instruction	Explanation
List Operations	
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 terminates.	
Text: `aList ← [value1, value2, value3, ...]` Block: 	Creates a new list that contains the values `value1, value2, value3`, and `...` at indices `1, 2, 3`, and `...` respectively and assigns it to `aList`.
Text: `aList ← []` Block: 	Creates an empty list and assigns it to `aList`.
Text: `aList ← bList` Block: 	Assigns a copy of the list bList to the list `aList`. For example, if `bList` contains `[20, 40, 60]`, then `aList` will also contain `[20, 40, 60]` after the assignment.
Text: `aList[i]` Block: 	Accesses the element of `aList` at index `i`. The first element of `aList` is at index `1` and is accessed using the notation `aList[1]`.
Text: `x ← aList[i]` Block: 	Assigns the value of `aList[i]` to the variable `x`.
Text: `aList[i] ← x` Block: 	Assigns the value of `x` to `aList[i]`.
Text: `aList[i] ← aList[j]` Block: 	Assigns the value of `aList[j]` to `aList[i]`.
Text: `INSERT(aList, i, value)` Block: 	Any values in `aList` at indices greater than or equal to `i` are shifted one position to the right. The length of the list is increased by 1, and `value` is placed at index `i` in `aList`.

Instruction	Explanation
List Operations—Continued	

Text:
`APPEND(aList, value)`
Block:
`APPEND aList, value`
	The length of `aList` is increased by 1, and `value` is placed at the end of `aList`.

Text:
`REMOVE(aList, i)`
Block:
`REMOVE aList, i`
	Removes the item at index `i` in `aList` and shifts to the left any values at indices greater than `i`. The length of `aList` is decreased by 1.

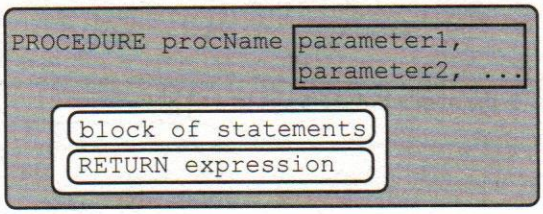
Text:
`LENGTH(aList)`
Block:
`LENGTH aList`
	Evaluates to the number of elements in `aList`.

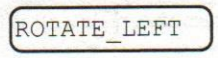
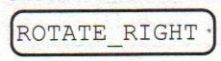
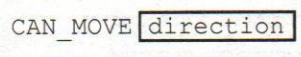
Text:
`FOR EACH item IN aList`
`{`
`<block of statements>`
`}`
Block:
`FOR EACH item IN aList`
`block of statements`
	The variable `item` is assigned the value of each element of `aList` sequentially, in order, from the first element to the last element. The code in `block of statements` is executed once for each assignment of `item`.

Procedures and Procedure Calls	

Text:
`PROCEDURE procName(parameter1, parameter2, ...)`
`{`
`<block of statements>`
`}`
Block:
`PROCEDURE procName parameter1, parameter2, ...`
`block of statements`
	Defines `procName` as a procedure that takes zero or more arguments. The procedure contains `block of statements`.
	
	The procedure `procName` can be called using the following notation, where `arg1` is assigned to `parameter1`, `arg2` is assigned to `parameter2`, etc.:
	`procName(arg1, arg2, ...)`

Instruction	Explanation
Procedures and Procedure Calls—Continued	

Text:
`PROCEDURE procName(parameter1, parameter2, ...)`
`{`
`<block of statements>`
`RETURN(expression)`
`}`
Block:
 
 	
	Defines `procName` as a procedure that takes zero or more arguments. The procedure contains `block of statements` and returns the value of `expression`. The `RETURN` statement may appear at any point inside the procedure and causes an immediate return from the procedure back to the calling statement.
	
	The value returned by the procedure `procName` can be assigned to the variable `result` using the following notation:
	`result ← procName(arg1, arg2, ...)`

Text: `RETURN(expression)` Block:  	Returns the flow of control to the point where the procedure was called and returns the value of `expression`.
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:  	The robot moves one square forward in the direction it is facing.
Text: `ROTATE_LEFT()` Block:  	The robot rotates in place 90 degrees counterclockwise (i.e., makes an in-place left turn).
Text: `ROTATE_RIGHT()` Block:  	The robot rotates in place 90 degrees clockwise (i.e., makes an in-place right turn).
Text: `CAN_MOVE(direction)` Block:  	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`.