The following are — for the most part — answers from students that got full credit for a question. In some cases I have added comments or a supporting statement. I copy answers without attribution or permission.

Proff Towell

1.) (Required) Suppose there exists a language, DyGo, which uses dynamic scoping but is otherwise identical to Go. Consider the following DyGo program. (The program below will not compile under Go, but will under DyGo.)

Part 1) What is the output of this program?

AA 5 3 // x from input, y from definition on line 9
BB 5 // x from main, y from global
AA 10 3 // x from input, y from definition on line 9
BB 5 hello // x from main, y from scope 22-25.
AA20 3 // x from input, y from scope 26 to 31.
BB 20 goodbye // x from scope 26-31, y from scope 26-31.

Part 2) Identify every scope (using the line numbers) and the variables that are accessible within that scope. If there are two variables of the same name that could be accessible within a scope then clearly identify the one that is visible.

Global scope: line 1 to line 32: Var x

Var y Func a

called on line 29, gets y from the inner scope defined from 26 to 31.
Func b

Func b: lines 13 to 15.
No local variables override-- can see x and y from its calling scope.
When called on line 20, gets x from main() and y from the global scope.

Func a: Lines 8 to 11
Variable y. One defined in scope func a y := 3 is the one that's accessible. Hides the global variable.
Hides the variable x it inherits from its calling scope with its parameter x. When called on line 23, gets y from the inner scope from 21 to 25, when

25.
from the same scope.

When called on line 24, gets x from main() and y from the inner scope from 21 to

When called on line 30, gets x from the inner scope defined from 26 to 31, and y

Func main from lines 17 to 32
Sets value of global variable x equal to 5

Inner scope: Line 21 to 25.
Variable y := "hello", hides global variable y

Inner scope Line 26 to 31
Variable x hides x defined in func main, x:= 20 Variable y hides the global variable y, y = "goodbye"

Example of this (small Go program portion illustrating only this statement):

```
var hh *int


func one() {
    aa := 1
    hh = &aa
}
```

This program illustrates this statement as the variable aa is a local variable created within the one function thus it SHOULD be reachable only within this function; HOWEVER, because the global variable hh is a pointer, pointing to aa's address and thus it's value, the variable aa now continues to exist even after its enclosing function (the one function) has returned.

3. VSC warns about recursion in line 6 because the fmt.Sprintf() function takes in len(item) and the item itself rather than an instance of the item or a casting of the item. If a function implements method String() string, that method will be invoked to convert the object to a string. Item is not a string but rather the object of the type that the function is calling, if we think in terms of object-oriented programming languages. The stack overflow occurs because the program is trying to use more memory space in the call stack than has been allocated to the stack. Each recursive call uses space on the stack, so if the recursion is too deep/infinite (which it is in this case because we did not provide a vase case as we did not intend for recursion to happen), then you get a stack overflow error.

The doNso() function does not create a stack overflow and doSo() does because doNso() creates a string instance of item rather than putting the item itself through fmt.Sprintf.

Missing from this answer is that the change has the effect because Go uses static method binding, so the cat over to string in nso change s the Strinf function being called. The Change does not work in Java because it uses Dynamic method binding. Hence, the cast to Object (in my example code) does not change the toString method that is being ised.

4. Tuple assignment in Go may seem like syntactic sugar, and in some cases it is, however in others it proves to be quite useful. It allows for quicker, more efficient coding with shorthand styles while also allowing for more flexibility, like Go is known for.
An example of tuple assignment usage that would be considered syntactic sugar, even with its efficiency and easy use would be when we want to switch/swap two variable's values with each other. For example, let's say:

```
x := 1
y := 2
x, y = y, x
```

This is of course a faster way to swap variable because it uses only three lines, it could use even less (two lines) by doing double the amount of tuple assignments:

```
x, y := 1, 2
x, y = y, x
```

This is extremely useful as it cuts downs lines of codes while also giving easy readability, but it isn't really needed as it can be achieved, albeit longer, by creating another variable to hold one of the two "to swap" variable's values:

```
x := 1
y := 2
z := x

x = y
y = z
```

However, tuple assignments are NOT fully/actually syntactic sugar since this "comma system" allows for multiple values to be returned from a function unlike other programming languages who only allow one return value (for example: Java). With values affecting others, in some instances, multiple return values may change in value if split into two function that return one value each. For example if we want to find the Fibonacci value starting with a new number (and not 1) while also calculating for what nth place the value is at, tuple assignment would make it easier. Without tuple assignment, we may lose the actual value of the nth number. For instance's where multiple return values can be calculated and returned, Go's tuple assignments proves to be far more useful than just "pretty packaging".

The above answer makes many good points but misses that you can return multiple values from a function in go by returning a struct. This might be wildly inconvenient, but it is certainly possible.

5. Byte-compiled languages are languages that are compiled into bytecode, which is then interpreted by a virtual machine. This allows byte-compiled languages to be portable across platforms because the bytecode can be run on any platform that has a virtual machine for the language. Byte-compiled languages also have the advantage of being easier to debug because the bytecode is easier to understand than machine code. The main disadvantage of byte-compiled languages is that they are usually slower than compiled languages because the bytecode has to be interpreted by the virtual machine.

Compiled languages are compiled into machine code, which is then run directly by the processor. This allows compiled languages to be very fast because the machine code can be run directly by the processor without interpretation. They also have the advantage of being more flexible, because the source code can be easily modified. The main disadvantage of compiled languages is that they are not portable across platforms because the machine code can only be run on the platform for which it was compiled.

6: fun: 48 // the first call to fun(16) at line 22 will call the function named fun declared at line 17, which will activate the print statement at line 18, printing the value of x + a, which is 32 + 16, because x is defined in fun's closure as 32 from line 16, and a is defined as fun's input parameter.

fun: 48 // the call to subFun on line 23 calls fun(16) in the process, on line 10 when printing aFun(x), which is fun(x) printing 48 to the screen (see prev. For why 48)

fun: 48 // repeat of last, calls y on line 10, which is stored as aFun(x), which calls fun with the value of 16.

Sub: 112 // the call to subFun will run the function returned from fff. This means that running subFun will run the function defined as qq in function fff on line 9, which will print the value ("Sub" + (aaa+aFun(x) +y)), which at this time is defined to be 16+fun(16) +fun(16), which is 16 + 48 + 48, which is ("Sub: " + (112))

fun: 48 // fun is called a third time with the input 16 on the return statement within fff on line 11., see 1, 2, and 3 for why it prints out 48.

fun: 96 // the value of x changes (within a scope visible to fun) on line 24, therefore when fun evaluates with an input variable of 32 from line 25, the print statement on line 18 prints 64 + 32, which is Fun: 96.

fun: 80 // subFun(32) is called on line 26, which runs fff's returned subfunction qq with the parameter 32. On the print statement in qq, aFun(x) evaluates with it's input variable as 16. When fun gets run, x is 64 within the closure of fun, because x's value is changed to 64 in the same scope as where fun is defined, meaning it prints out on line 18 the value 80.

Sub: 160 // when subFun(32) is called on line 26, it prints out on line 10 the value of aa + aFun(x) + y. We established previously aFun(x) is 80. Aa is 32, its input from line 26. However-- y's closure does not include the change in variable x to 64 at line 24, because y is not defined within the same scope as that variable change. Therefore, y stays as 48. This print out 80 + 32 + 48 = 160.

fun: 80 // When subFun(32) is called, it evaluates aFun(x) again, printing x + a = 64 + 16 = 80.

```
7
package main

import (
	"fmt"
	"math/rand"
	"time"
)

func makeDeck() []int {
	rand.Seed(time.Now().UnixNano())
	var a []int
	for i:=0; i<10; i++ {
		a=append(a, rand.Intn(4)+1)
	}
	return a
}

func main() {
	deck := makeDeck()
	fmt.Println(deck)
	for i:=0; i<(len(deck)-3); i++ {
		if deck[i]==deck[i+3] {
			deck = append(deck[0:i+1], deck[i+3:]...)
			fmt.Printf("Compressing %v : %v\n", i, deck)
			i--
		}
	}
	fmt.Println(deck)
}%
```

8. If this were true, functions could be used to make the code look neater to the programmer and abstract sections of code. However, if I am understanding this correctly, if a function has a single static location in the compiled representation of the program, it can only be called for as many times as is predicted and preallocated at compile time. For example, recursion would not be possible, as it relies on the ability to add functions to the stack and then trace the functions called before as they are returned.