Summary of Go Syntax

02-201 / 02-601

Variables

Variables are boxes that contain data. The data is of a particular *type*. The box is labeled with the variable's name.

Can declare 1 or more variables in same var statement

Can optionally provide initial values for all the variables (if omitted, each variable defaults to the "0" value for its type)

```
var Name1, Name2, … Type = InitialValue1, InitialValue2, …
```

The type of the variables (Can be omitted if you provide initial values from which the type can be inferred)

```
Name1, Name2 := EXPRESSION
```

Can abbreviate using the := syntax. At least 1 variable on the left-hand side must be new.

- Variables have types that never change
- Uninitialized variables start with their 0 value (i.e. 0, 0.0, "", false)
- You can't declare variables that you don't use

Scope

- Variables last from when they are declared to the end of the { } block that they are declared in
- Exception: variables declared as function parameters last for the entire function
- Global variables (those not in a {} block) last from when they are declared until the program ends.

```
func gcd(a int, b int) int {
    if a == b {
        return a
    }
    var c int
    if a > b {
        c = a - b
        return gcd(c, b)
    } else {
        c = b - a
        return gcd(a, c)
    }
}
```

variable c created

variable c destroyed

Types

Basic Numeric and logical types

```
int, int8, int16, int32, int64
uint, uint8, uint16, uint32, uint64
bool
float32, float64
```

List types: arrays (fixed size), slices (variable size), strings (like []int8)

```
[N]Type
[]Type
string
```

Maps: relate unique keys of Type1 to values of Type2. Any type where == is defined can be a key type.

```
map[Type1]Type2
```

Structures: a grouping of a small number of variables

```
struct {
    N1 T1
    N2 T2
    N3 T3
}
```

Pointers: a reference to a variable of type Type

```
*Type
```

Types, 2

- Types can be composed to create complex data structures:

```
map[int]struct{id int; n []*map[string]int}
```

a map from ints to structs, each containing an int field id and a field “n” that is a slice of pointers to maps from strings to ints

- New names for types can be provided via the **type** statement:

```
type Name Type
```

```
type S struct {id int; n []*map[string]int}
```

```
type EmployeeId int
```

Slices & Arrays

```
var slice []Type
```
[]Type declares a slice with elements of type Type

```
slice = make([]Type, Length)
```
slices start out as **nil**, you must call **make** to create the actual slice.

```
len(slice)
```
the length of slice S can be obtained with **len**(S)

```
var array [10]Type
```
Arrays are slices with an explicit length (that is known at compile time)

```
slice[I]
array[J]
```
access the I and Jth elements of slices an arrays. I and J can be arbitrary integer expressions (i.e. 3*a + 7*b)

Elements are numbered starting at 0

It’s an error to try to use an index greater than the size of the slice or array.

- It’s almost always better to use a slice instead of an array.

Maps

```
var Name map[KeyType]ValueType
```
Declare a map from KeyType to ValueType

```
Name = make(map[KeyType]ValueType)
```
Map variables start as **nil**
Must call make to create the map

```
Name[Key] = Value
fmt.Println(Name[Key])
```
Set and access elements in maps like arrays and slices

```
v = Name[Key]
v, ok = Name[Key]
```
Check if key present by trying to extract 2 values from the array; second value will be true if Key is in Name

```
delete(Name, Key)
len(Name)
```
Remove an item with **delete**
Get number of items with **len**

Structs

- Group together related variables to create a logical, composite object.

```
struct {
    N1 T1
    N2 T2
    N3 T3
}
```

```
type Name struct {
    N1 T1
    N2 T2
    N3 T3
}
```

- Nearly always used in a **type** statement to give a name to the struct.

- Access fields using the “.” syntax:

```
S.F1 = 3
fmt.Println(S.F2)
```

Composite Literals

- Specify initial values for composite types (arrays, slices, maps, structs) using the general syntax:

```
ListType{value1, value2, value3, …}
MapOrStructType{key1:value1, key2:value2, …}
```

- Examples:

```
[]int{3,4,1,5,6,-6}                    // slice literal
[5]uint{3,4,1,5,6}                     // array literal
map[string]int{“a”:3, “b”:7}           // map literal
Contact{name:”Eric”, age:37}           // struct literal
```

Pointers

- Pointers hold the addresses of other variables.

```
var pName *Type
```
Declare pointers by using type *Type

- Pointers start out **nil**

```
pName = &i
```
Use the & operator to get the address of a variable to store into a pointer

```
*pName
```
Use *PointerName to follow the pointer (called “dereferencing”)

*PointerName acts just like the variable it points to.

```
(*pStruct).Field
```
These are equivalent in the special case of a pointer to a struct

```
pStruct.Field
```

Type Conversions

`Type(Expression)` ⟵ Converts the result of Expression into type Type (if possible)

```
var v int
var q float64 = 3.14
v = int(q)
```

- To convert a string to a number or vice versa you have to use a library call; Package strconv provides a number of such functions.

Operators

```
+       addition, string concatenation
-       subtraction, negation
/       division, integer division
%       remainder

&&      logical AND
||      logical OR
!       logical NOT

==      equals
!=      not equals
<=      less than or equal to
>=      greater than or equal to
<       less than
>       greater than

&       address of
*       follow pointer (“dereference”)
```

- All the operands for an operator must have the same type.

Increment & Decrement Statements

- For integer variables i :

```
i++     means increase i by 1
i--     means decrease i by 1
```

Assignment Operators

```
a += Expr      means a = a + (Expr)
a -= Expr      means a = a - (Expr)
a *= Expr      means a = a * (Expr)
a /= Expr      means a = a / (Expr)
a %= Expr      means a = a % (Expr)
```

Constants

```
const Name Type = Value
```

Type is optional.

If it's omitted, the constant doesn't have a type and can be used anywhere "Value" could have been typed directly

Can group together several constant definitions

```
const (
    Name1 Type1 = Value1
    Name2 Type2 = Value2
    // …
)
```

- Best practice: declare constants for any non-trivial numbers you have in your code.

Import, main()

- You code starts by running the main() function.
- You can import packages to access functions and types they include via:

```
import “packageName”

import (
    “packageName1”
    “packageName2”
    //…
)
```

- The functions from package P are accessible via P.FunctionName

Flow Control: Conditionals

```
if BooleanExpression {
    // statements T
} else {
    // statements F
}
```

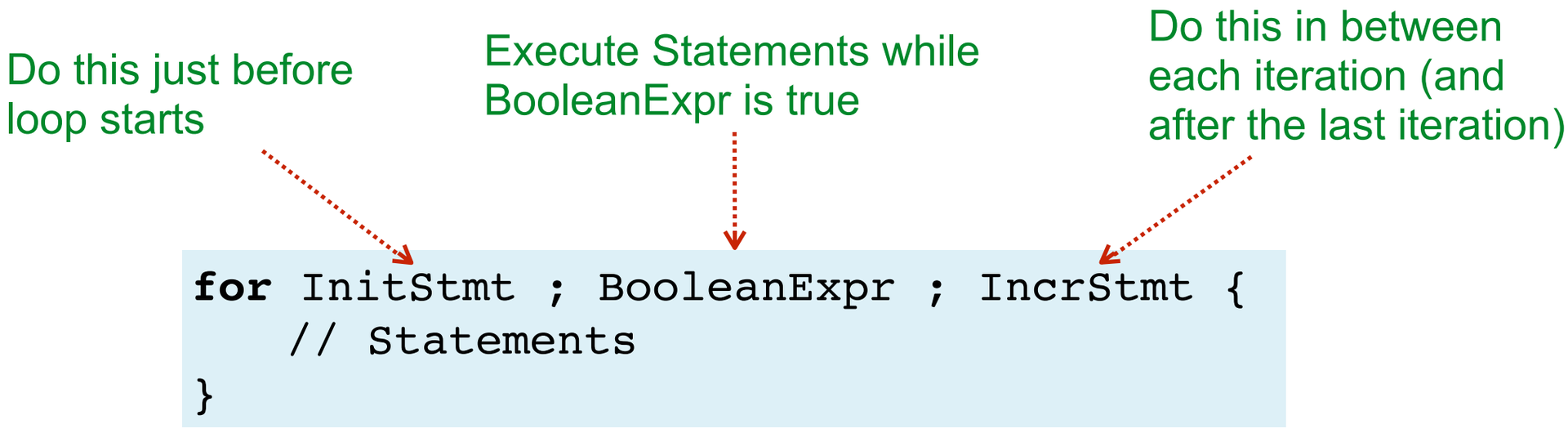
If BooleanExpression is true do statements T otherwise do statements F

```
switch Value {
case V1:
    // statements
case V2:
    // statements
case V3, V4:
    // statements
default:
    // statements
}
```

Do the first set of statements with a case value (e.g. V1) that matches the switch Value

Do the (optional) default statements if no other case matches.

Flow Control: Loops



- Any variables declared in the InitStmt have scope of just the for loop
- One or both of InitStmt and IncrStmt can be empty
- If both are empty, you can omit the “;”
- If BooleanExpr is empty, it means “true”

Flow Control: Looping Over Lists and Maps

The index or key of the current element

The value of the current element (this can be omitted to just loop over the indices)

A map, array, slice, or string variable

```
for i, v := range ListVar {
    // Statements
}
```

- Use the blank identifier “_” if you don’t need i
- If ListVar is a map, the order of the elements is not defined

Functions

The receiver type: if present, function must be called using C.Name where C is of type T0

The parameters and their types (if the type is the same as the previous parameter, it can be omitted)

The return types (if only 1, the parentheses can be omitted)

```
func (m *T0) Name(param1 T1, param2 T2, …) (RT1, RT2, …) {
    // Statements
    // Including a return statement if any return types were
    // declared
}
```

- Functions can return ≥ 0 values and can have ≥ 0 parameters
- All possible paths through a function must return a value if a return type is declared
- Variables of a simple type (int, string, bool, etc) are copied when passed into a function.
- Arrays are **copied** when passed into a function
- Maps and slices are passed by reference: you can change the values in a map or slice within the function.

Design Strategies

- Top-down: write a function to solve your problem, “creating” (but not writing) functions for smaller problems as needed.

- Object-oriented: create types for your real-world entities (users, files, cars) and then write methods that manipulate those entities.

- Data structuring: choose a way to arrange your data into variables (maps, lists, structs, etc.) that make answering your desired questions easy.

- Small-to-large: start with a small, simple case, and then add complexity incrementally after you have worked out bugs at each stage.