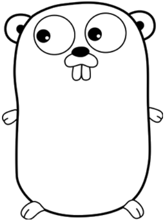
COMP-520 – GoLite Tutorial

Alexander Krolik

Sable Lab
McGill University

Winter 2019



Plan

- Target languages
- Language constructs, emphasis on special cases
  - General execution semantics
  - Declarations
  - Types
  - Statements
  - Expressions
- Implementation advice

Feel free to ask questions at any time.

Reference compiler

- ssh <socs_username>@teaching.cs.mcgill.ca
- `~cs520/golitec {keyword} < {file}`
- Codegen outputs C++ code (can be compiled with `g++ --std=c++11 {file}`)
- If you find errors in the reference compiler, bonus points!

Reminder

We know that previous year's submissions are available online. There are 3 requirements for this class:

1. You must come up with your own solutions; any inspiration that comes from other sources must be reported.
2. You must have permission to use any outside resources from the original authors.
3. No grading material may be used at any point, under any circumstance, nor may it be published.

Target language

For the project, carefully choosing the right target language is important. You should consider the following factors:

Target language

For the project, carefully choosing the right target language is important. You should consider the following factors:

- Low-level vs. high-level

Target language

For the project, carefully choosing the right target language is important. You should consider the following factors:

- Low-level vs. high-level
- Statically-typed vs. dynamically-typed

Target language

For the project, carefully choosing the right target language is important. You should consider the following factors:

- Low-level vs. high-level
- Statically-typed vs. dynamically-typed
- Similarity to Go

Target language

For the project, carefully choosing the right target language is important. You should consider the following factors:

- Low-level vs. high-level
- Statically-typed vs. dynamically-typed
- Similarity to Go
- (No C++ as this is used in the reference implementation)

Target language

Previous years

- C
- Java
- Swift
- JavaScript
- TypeScript
- Python
- Java Bytecode
- LLVM
- x86

Go execution

An executable Go program consists of:

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function
- Zero-or-more other top-level declarations

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function
- Zero-or-more other top-level declarations

During program execution, Go is:

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function
- Zero-or-more other top-level declarations

During program execution, Go is:

- Pass-by-value

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function
- Zero-or-more other top-level declarations

During program execution, Go is:

- Pass-by-value
- Return-by-value

Go execution

An executable Go program consists of:

- Zero-or-more `init` functions
- One `main` function
- Zero-or-more other top-level declarations

During program execution, Go is:

- Pass-by-value
- Return-by-value
- (Mostly) left-to-right evaluation order

Go execution

Special functions

Special functions are used as entry points into the program. When a Go program is executed, the control code:

Go execution
Special functions

Special functions are used as entry points into the program. When a Go program is executed, the control code:

1. Invokes the `init` functions in *lexical* order

Go execution

Special functions

Special functions are used as entry points into the program.
When a Go program is executed, the control code:

1. Invokes the `init` functions in *lexical* order
2. Invokes the `main` function

Go execution

Special functions

Special functions are used as entry points into the program.
When a Go program is executed, the control code:

1. Invokes the `init` functions in *lexical* order
2. Invokes the `main` function

You may assume our tests always include a main method

Go execution

Special functions

Special functions are used as entry points into the program.
When a Go program is executed, the control code:

1. Invokes the `init` functions in *lexical* order
2. Invokes the `main` function

You may assume our tests always include a main method

```
package main

func init() { ... } // init1

func main() { ... }

func init() { ... } // init2
```

In which order are the functions executed?

Go execution

Special functions

Special functions are used as entry points into the program. When a Go program is executed, the control code:

1. Invokes the `init` functions in *lexical* order
2. Invokes the `main` function

You may assume our tests always include a main method

```
package main

func init() { ... } // init1

func main() { ... }

func init() { ... } // init2
```

In which order are the functions executed? **init1, init2, main**

Declarations

Like most languages, Go has 3 kinds of declarations:

- Function declarations
- Variable declarations
- Type declarations

Declarations

Like most languages, Go has 3 kinds of declarations:

- Function declarations
- Variable declarations
- Type declarations

While it might seem easy, there are 3 common issues translating declarations:

Declarations

Like most languages, Go has 3 kinds of declarations:

- Function declarations
- Variable declarations
- Type declarations

While it might seem easy, there are 3 common issues translating declarations:

- Naming conflicts with keywords

Declarations

Like most languages, Go has 3 kinds of declarations:

- Function declarations
- Variable declarations
- Type declarations

While it might seem easy, there are 3 common issues translating declarations:

- Naming conflicts with keywords
- Scoping differences

Declarations

Like most languages, Go has 3 kinds of declarations:

- Function declarations
- Variable declarations
- Type declarations

While it might seem easy, there are 3 common issues translating declarations:

- Naming conflicts with keywords
- Scoping differences
- Blank identifiers

Declarations

Naming conflicts

Naming conflicts occur when an identifier is legal in Go, but a keyword in the target language.

```
var restrict int    // Conflict in C

func None() {}      // Conflict in Python
```

What approach avoids all possible keyword conflicts?

Declarations

Naming conflicts

Naming conflicts occur when an identifier is legal in Go, but a keyword in the target language.

```
var restrict int    // Conflict in C

func None() {}      // Conflict in Python
```

What approach avoids all possible keyword conflicts?

Renaming all identifiers with a unique prefix/suffix

Be careful, we must ensure that the renaming does not cause any further conflicts.

Function declarations

Function declarations consist of a name, set of parameters, and an optional return type.

- Nearly the same across all programming languages
- Beware: test all types!

Function declarations

Function declarations consist of a name, set of parameters, and an optional return type.

- Nearly the same across all programming languages
- Beware: test all types!

Is it valid to have untagged struct parameters in all languages?

```
func foo(a struct { a int; }) { ... }
```

Function declarations

Function declarations consist of a name, set of parameters, and an optional return type.

- Nearly the same across all programming languages
- Beware: test all types!

Is it valid to have untagged struct parameters in all languages?

```
func foo(a struct { a int; }) { ... }
```

No! In particular, this is not legal in C or C++

Function declarations

Function declarations consist of a name, set of parameters, and an optional return type.

- Nearly the same across all programming languages
- Beware: test all types!

Is it valid to have untagged struct parameters in all languages?

```
func foo(a struct { a int; }) { ... }
```

No! In particular, this is not legal in C or C++

How can we overcome the limitation of these languages?

Function declarations

Function declarations consist of a name, set of parameters, and an optional return type.

- Nearly the same across all programming languages
- Beware: test all types!

Is it valid to have untagged struct parameters in all languages?

```
func foo(a struct { a int; }) { ... }
```

No! In particular, this is not legal in C or C++

How can we overcome the limitation of these languages?

typedef the struct

Variable declarations

Variable declarations are relatively straightforward. In Go, there are 4 special cases:

Variable declarations

Variable declarations are relatively straightforward. In Go, there are 4 special cases:

- Implicit initialization

```
var a int // Implicitly initialized to 0
```

Variable declarations

Variable declarations are relatively straightforward. In Go, there are 4 special cases:

- Implicit initialization

  ```
  var a int // Implicitly initialized to 0
  ```

- Multiple declarations

  ```
  var a, b int
  ```

  We'll come back to this with assignment statements

Variable declarations

Variable declarations are relatively straightforward. In Go, there are 4 special cases:

- Implicit initialization

```
var a int // Implicitly initialized to 0
```

- Multiple declarations

```
var a, b int
```

  We'll come back to this with assignment statements

- Shadowing of `true` and `false` constants

```
var true bool = false
```

Variable declarations

Variable declarations are relatively straightforward. In Go, there are 4 special cases:

- Implicit initialization

```
var a int // Implicitly initialized to 0
```

- Multiple declarations

```
var a, b int
```

  We'll come back to this with assignment statements

- Shadowing of `true` and `false` constants

```
var true bool = false
```

- Scoping

Variable declarations

Scoping

Scoping rules vary widely and wildly between different programming languages.

```
var a int
{
    var b int = a
    var a int = a  // ‘a’ points to the parent scope
}
```

Can we directly translate the above code to C? JavaScript?

Variable declarations

Scoping

Scoping rules vary widely and wildly between different programming languages.

```
var a int
{
    var b int = a
    var a int = a  // 'a' points to the parent scope
}
```

Can we directly translate the above code to C? JavaScript?

No! C: declaration points to itself. JS: no block scoping

Variable declarations

Scoping

Scoping rules vary widely and wildly between different programming languages.

```
var a int
{
    var b int = a
    var a int = a  // ‘a’ points to the parent scope
}
```

Can we directly translate the above code to C? JavaScript?

No! C: declaration points to itself. JS: no block scoping

What is an easy solution to this problem?

Variable declarations

Scoping

Scoping rules vary widely and wildly between different programming languages.

```
var a int
{
    var b int = a
    var a int = a  // 'a' points to the parent scope
}
```

Can we directly translate the above code to C? JavaScript?

No! C: declaration points to itself. JS: no block scoping

What is an easy solution to this problem?

Renaming!

Type declarations

Do we need to generate type declarations (i.e. defined types) if our target language is:

- Dynamically-typed?

Type declarations

Do we need to generate type declarations (i.e. defined types) if our target language is:

- Dynamically-typed? **No!**

Type declarations

Do we need to generate type declarations (i.e. defined types) if our target language is:

- Dynamically-typed? **No!**
- Statically-typed?

Type declarations

Do we need to generate type declarations (i.e. defined types) if our target language is:

- Dynamically-typed? **No!**
- Statically-typed? **No!**

Defined types are only required for the purpose of type-checking. In terms of storage it makes no difference.

Declarations

Blank identifiers

Blank identifiers may be used in:

- Function names
- Function parameters
- Variable names (declarations/assignments)
- Struct fields

Blank functions and struct fields are easy to generate. Why?

Declarations

Blank identifiers

Blank identifiers may be used in:

- Function names
- Function parameters
- Variable names (declarations/assignments)
- Struct fields

Blank functions and struct fields are easy to generate. Why?

They may never be accessed and can thus be ignored

Declarations

Blank parameters

If a function has blank parameters, they must still be generated as function calls will include the arguments.

```
func foo(_ int, a int, _ int) { ... }

func main() {
    foo(1, 2, 3)
}
```

What problem will occur in the above code?

Declarations

Blank parameters

If a function has blank parameters, they must still be generated as function calls will include the arguments.

```
func foo(_ int, a int, _ int) { ... }

func main() {
    foo(1, 2, 3)
}
```

What problem will occur in the above code?

Naming conflicts between parameters

Declarations

Blank parameters

If a function has blank parameters, they must still be generated as function calls will include the arguments.

```
func foo(_ int, a int, _ int) { ... }

func main() {
    foo(1, 2, 3)
}
```

What problem will occur in the above code?

Naming conflicts between parameters

What approach can we use to guarantee unique naming?

Declarations

Blank parameters

If a function has blank parameters, they must still be generated as function calls will include the arguments.

```
func foo(_ int, a int, _ int) { ... }

func main() {
    foo(1, 2, 3)
}
```

What problem will occur in the above code?

Naming conflicts between parameters

What approach can we use to guarantee unique naming?

Temporary variable names

Declarations

Blank variables

When assigning into a blank identifier, the value is discarded.

```
var _ int = ...
```

Can we therefore eliminate the declaration?

Declarations

Blank variables

When assigning into a blank identifier, the value is discarded.

```
var _ int = ...
```

Can we therefore eliminate the declaration?

No!

```
func foo() int {
    println("foo")
    return 0
}

var _ int = foo()
```

Expressions evaluated as part of declarations may have side-effects and should still be executed.

Types

Basic types:

- `int` (may be either 32 or 64 bit depending on the architecture)
- `float64`
- `bool`
- `rune`
- `string`

Composite types:

- Arrays
- Slices
- Structs

Arrays

What is an array?

- Data structure for homogeneous data
- Fixed number of elements
- Typically implemented as a contiguous section of memory

Arrays

What is an array?

- Data structure for homogeneous data
- Fixed number of elements
- Typically implemented as a contiguous section of memory

In Go they have two interesting properties:

Arrays

What is an array?

- Data structure for homogeneous data
- Fixed number of elements
- Typically implemented as a contiguous section of memory

In Go they have two interesting properties:

- Bounds checking

Arrays

What is an array?

- Data structure for homogeneous data
- Fixed number of elements
- Typically implemented as a contiguous section of memory

In Go they have two interesting properties:

- Bounds checking
- Equality

Go provides bounds checking for arrays, producing runtime error if the index is out of bounds.

```
var a [5]int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

Go provides bounds checking for arrays, producing runtime error if the index is out of bounds.

```
var a [5]int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

1. **Use a container with built-in bounds checking**

Arrays

Bounds checking

Go provides bounds checking for arrays, producing runtime error if the index is out of bounds.

```
var a [5]int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

1. **Use a container with built-in bounds checking**
2. **Wrap all indexes in a special "bounds-checking" function**

Arrays

Equality

Go also provides element-wise equality for arrays, returning true iff all elements are equal.

```
var a, b [5]int
println(a == b) // Ouputs true

b[0] = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement array equality?

Arrays
Equality

Go also provides element-wise equality for arrays, returning true iff all elements are equal.

```
var a, b [5]int
println(a == b) // Ouputs true

b[0] = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement array equality?

1. **Use a container with built-in equality**

Arrays

Equality

Go also provides element-wise equality for arrays, returning true iff all elements are equal.

```
var a, b [5]int
println(a == b) // Ouputs true

b[0] = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement array equality?

1. **Use a container with built-in equality**
2. **Implement helper functions for each kind of array**

Beware! Arrays can contain other arrays or structures - your helper methods must account for this.

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices in Go are implemented internally using two structures:

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices in Go are implemented internally using two structures:

- An underlying array storing the elements
- A *header* struct

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices in Go are implemented internally using two structures:

- An underlying array storing the elements
- A *header* struct
  - Pointer to the underlying array
  - Capacity and length

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices in Go are implemented internally using two structures:

- An underlying array storing the elements
- A *header* struct
  - Pointer to the underlying array
  - Capacity and length

Slices

What is a slice?

- Data structure for homogeneous data
- *Dynamic* number of elements

Slices in Go are implemented internally using two structures:

- An underlying array storing the elements
- A *header* struct
    - Pointer to the underlying array
    - Capacity and length

As the size of the slice changes, the header is updated and the underlying array reallocated if needed.

You will likely face a trade-off between correctness and efficiency.

Slices

Bounds checking

Go provides bounds checking for slices, producing runtime error if the index is out of bounds.

```
var a []int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

Slices

Bounds checking

Go provides bounds checking for slices, producing runtime error if the index is out of bounds.

```
var a []int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

1. **Use a container with built-in bounds checking**

Slices

Bounds checking

Go provides bounds checking for slices, producing runtime error if the index is out of bounds.

```
var a []int
a[10] = 0 // Runtime out-of-bounds error
```

What approaches can we use to implement bounds checking?

1. **Use a container with built-in bounds checking**
2. **Wrap all indexes in a special "bounds-checking" function**

The special function is trickier for slices - it must use the dynamic size from the slice header.

Struct

What is a struct?

- Data structure for heterogeneous data
- Fixed structure

Languages like C already provide this data structure. How do we implement this in other higher-level languages?

Struct

What is a struct?

- Data structure for heterogeneous data
- Fixed structure

Languages like C already provide this data structure. How do we implement this in other higher-level languages?

Objects, records, etc.

We will not check nor implement any low-level details such as alignment or padding.

Structs

Equality

Go provides field-wise equality for structs, returning true iff all *non-blank* fields are equal. Empty structs are trivially equal.

```
var a, b struct {
    f int
    _ float64
}
println(a == b) // Ouputs true

b.f = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement struct equality?

Structs

Equality

Go provides field-wise equality for structs, returning true iff all *non-blank* fields are equal. Empty structs are trivially equal.

```
var a, b struct {
    f int
    _ float64
}
println(a == b) // Ouputs true

b.f = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement struct equality?

1. **Use a container with built-in equality**

Structs

Equality

Go provides field-wise equality for structs, returning true iff all *non-blank* fields are equal. Empty structs are trivially equal.

```
var a, b struct {
	f int
	_ float64
}
println(a == b) // Ouputs true

b.f = 1
println(a == b) // Ouputs false
```

What approaches can we use to implement struct equality?

1. **Use a container with built-in equality**
2. **Implement helper functions for each kind of struct**

Beware! Structs can contain other structs or arrays - your helper methods must account for this.

Statements

- Assignments
- Short declarations
- Increment/decrement
- Ifs
- For loops
- Switches
- Returns
- Prints

Assignments

An assignment statement:

Assignments

An assignment statement:

- Copies the value of the expression to the variable
- Ignores assignments of blank identifiers
- May assign multiple values *simultaneously*

```
var a, b int
a = 5          // ``Copies'' 5 to the variable `a'

_ = 5          // Ignored

a, b = b, a    // Swaps the values of `a' and `b'
```

Assignments

Are the copying semantics different for composite types?

```
var a, b [5]int

b = a
a[0] = 1

var c, d []int
c = append(c, 0)

d = c
c[0] = 1

var e, f struct { f int; }

f = e
e.f = 1
```

What are the values for `b[0]`, `d[0]` and `f.f` respectively?

Assignments

Are the copying semantics different for composite types?

No!

```
var a, b [5]int

b = a       // Copies the contents of 'a'
a[0] = 1    // Does not change 'b'

var c, d []int
c = append(c, 0)

d = c       // Copies the *header* of 'c'
c[0] = 1    // *Does* change 'd'!

var e, f struct { f int; }

f = e       // Copies the contents of 'e'
e.f = 1     // Does not change 'f'
```

What are the values for `b[0]`, `d[0]` and `f.f` respectively? **0, 1, 0**

Assignments

Blank assignments

Can we eliminate blank assignments altogether?

Assignments

Blank assignments

Can we eliminate blank assignments altogether?

No! The expression must still be evaluated

How can we implement the swapping semantics of multiple assignments?

How can we implement the swapping semantics of multiple assignments?

Use temporaries to store old values of all RHS expressions before assigning

```
int tmp__0 = b;
int tmp__1 = a;

a = tmp__0;
b = tmp__1;
```

Short declarations

Short declarations are a cross between assignments and declarations.

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign
- If the variable is not declared, define

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign
- If the variable is not declared, define

Otherwise, they follow the same logic as assignment:

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign
- If the variable is not declared, define

Otherwise, they follow the same logic as assignment:

- Copies the value of the expression to the variable

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign
- If the variable is not declared, define

Otherwise, they follow the same logic as assignment:

- Copies the value of the expression to the variable
- Ignores assignments of blank identifiers

Short declarations

Short declarations are a cross between assignments and declarations.

- If the variable is already declared, assign
- If the variable is not declared, define

Otherwise, they follow the same logic as assignment:

- Copies the value of the expression to the variable
- Ignores assignments of blank identifiers
- May assign multiple values *simultaneously*

Increment/decrement

Increment/decrement statements change the value of a numerical variable by 1. This is valid for:

- `int`
- `float64`
- `rune`

Increment/decrement

Increment/decrement statements change the value of a numerical variable by 1. This is valid for:

- `int`
- `float64`
- `rune`

Most languages support this functionality. If not, you can carefully generate another equivalent operation.

Beware! The following statements are *not* equivalent.

```
a[foo()]++  // foo() called once
```

```
a[foo()] = a[foo()] + 1  // foo() called twice
```

If statements

If statements in Go consist of:

If statements

If statements in Go consist of:

- Optional init statement

If statements

If statements in Go consist of:

- Optional init statement
- Condition expression

If statements

If statements in Go consist of:

- Optional init statement
- Condition expression
- True branch

If statements

If statements in Go consist of:

- Optional init statement
- Condition expression
- True branch
- Zero-or-more else-if branches
  - Optional init statement
  - Condition expression

If statements

If statements in Go consist of:

- Optional init statement
- Condition expression
- True branch
- Zero-or-more else-if branches
  - Optional init statement
  - Condition expression
- Optional else branch

If statements

If statements in Go consist of:

- Optional init statement
- Condition expression
- True branch
- Zero-or-more else-if branches
  - Optional init statement
  - Condition expression
- Optional else branch

The conditions are evaluated lexically until one evaluates to true and the branch is executed. Otherwise, the else branch is taken.

If statements

Init scoping

Be careful of scoping when translating to your target language - `init` statements are visible to all subsequent branches.

```
if a := false; a {            // Branch 1
    ...
} else if a := true; !a {    // Branch 2
    ...
} else if a {                 // Branch 3
    ...
} else {                      // Branch 4
    ...
}
```

Which branch executes?

If statements

Init scoping

Be careful of scoping when translating to your target language - `init` statements are visible to all subsequent branches.

```
if a := false; a {            // Branch 1
    ...
} else if a := true; !a {    // Branch 2
    ...
} else if a {                 // Branch 3
    ...
} else {                      // Branch 4
    ...
}
```

Which branch executes?

Branch 3

If statements

Init scoping

Be careful of scoping when translating to your target language - `init` statements are visible to all subsequent branches.

```
if a := false; a {            // Branch 1
    ...
} else if a := true; !a {     // Branch 2
    ...
} else if a {                 // Branch 3
    ...
} else {                      // Branch 4
    ...
}
```

Which branch executes?

Branch 3

What approach easily implements this functionality?

If statements

Init scoping

Be careful of scoping when translating to your target language - `init` statements are visible to all subsequent branches.

```
if a := false; a {            // Branch 1
    ...
} else if a := true; !a {     // Branch 2
    ...
} else if a {                 // Branch 3
    ...
} else {                      // Branch 4
    ...
}
```

Which branch executes?

Branch 3

What approach easily implements this functionality?

Decompose “else if” into “else { if”

If statements
Init scoping

Also note that the init statements are not visible outside of the if statement context.

What two approaches can we use to solve this?

If statements

Init scoping

Also note that the init statements are not visible outside of the if statement context.

What two approaches can we use to solve this?

1. **Renaming (again)!**

If statements

Init scoping

Also note that the init statements are not visible outside of the if statement context.

What two approaches can we use to solve this?

1. **Renaming (again)!**
2. **Nesting the entire if structure in another scope**

The above is valid for `for` *and* `switch` *init statements as well*

For loops

Infinite loops

Easy! Implicitly, the condition is always `true`.

```
for {
    ...
}
```

Still easy! The condition is a simple expression evaluated every iteration.

```
var a, b int
for a + b == 0 {
    ...
}
```

For loops
3-part loops

Very hard! We now have optional `init` and `post` statements.

```
for a, b := 0, 1; a < b; a, b = b, a {
    ...
    if (a > b) {
        continue
    }
    ...
}
```

What issues are present? How can we correctly translate the above code?

For loops
3-part loops

Very hard! We now have optional `init` and `post` statements.

```
for a, b := 0, 1; a < b; a, b = b, a {
    ...
    if (a > b) {
        continue
    }
    ...
}
```

What issues are present? How can we correctly translate the above code?

1. **Initialization may be several target statements**

For loops

3-part loops

Very hard! We now have optional `init` and `post` statements.

```
for a, b := 0, 1; a < b; a, b = b, a {
    ...
    if (a > b) {
        continue
    }
    ...
}
```

What issues are present? How can we correctly translate the above code?

1. **Initialization may be several target statements**
2. **Post may be several target statements**

For loops

3-part loops

Very hard! We now have optional `init` and `post` statements.

```
for a, b := 0, 1; a < b; a, b = b, a {
    ...
    if (a > b) {
        continue
    }
    ...
}
```

What issues are present? How can we correctly translate the above code?

1. **Initialization may be several target statements**
2. **Post may be several target statements**
3. `continue` **may conditionally execute**

For loops

3-part loops

In most languages, representing the 3-part loop as a `while` loop is natural. For `continue` we can use labels and jumps.

```
{
    int tmp__0 = 0;
    int tmp__1 = 1;
    int a = tmp__0;
    int b = tmp__1;
    while (a < b) {
        if (a > b) {
            goto continue__lbl;
        }
      continue__lbl:
        int tmp__2 = b;
        int tmp__3 = a;
        a = tmp_2;
        b = tmp_3;
    }
}
```

Beware! You must be *very* careful of scoping issues when placing the post-statement in the loop body.

Switch statements

Switch statements in Go consist of:

Switch statements

Switch statements in Go consist of:

- Optional init statement

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases
  - List of one-or-more non-constant expressions

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases
  - List of one-or-more non-constant expressions
  - Body

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases
  - List of one-or-more non-constant expressions
  - Body
  - Optional break(s)

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases
  - List of one-or-more non-constant expressions
  - Body
  - Optional break(s)
- Optional default case

Switch statements

Switch statements in Go consist of:

- Optional init statement
- Optional switch expression
- Zero-or more cases
  - List of one-or-more non-constant expressions
  - Body
  - Optional break(s)
- Optional default case

Phew! Likely the hardest statement kind to implement correctly.

Switch statements

We want to codegen the following Go program fragment in C.

```
switch foo() {
    case a, baz():
        if (b > c) {
            break
        }
    default:
}
```

Switch statements

Proposal 1: Implement switches using `switch` from C.

Does it work?

Switch statements

Proposal 1: Implement switches using `switch` from C.

Does it work?

No!

```
switch (foo()) {
    case a:
    case baz():  // Problem: illegal in C
        if (b > c) {
            break;
        }
        break;
    default:
}
```

Switch statements

Proposal 2: Implement switches using `if-elseif-else`.

Does it work?

Switch statements

Proposal 2: Implement switches using `if-elseif-else`.

Does it work?

Mostly! Two smaller issues

```
// Problem 1: foo() is evaluated twice
if (foo() == a || foo() == bar()) {
    if (b > c) {
        break;  // Problem 2: illegal in C
    }
} else {
    // Default branch
}
```

Switch statements

Proposal 3: Implement switches using `if-elseif-else` from C using:

- Temporary for the condition
- Labels for break

Does it work?

Switch statements

Proposal 3: Implement switches using `if-elseif-else` from C using:

- Temporary for the condition
- Labels for break

Does it work?

Yes!

```
int tmp__0 = foo()
if (tmp__0 == a || tmp__0 == bar()) {
    if (b > c) {
        goto break__lbl;
    }
} else {
    // Default branch
}

break__lbl:;
```

Return statements

Go is a return-by-value language (i.e. the return value is copied into the calling function's stack frame).

Return statements

Go is a return-by-value language (i.e. the return value is copied into the calling function's stack frame).

- Easy for basic types

Return statements

Go is a return-by-value language (i.e. the return value is copied into the calling function's stack frame).

- Easy for basic types
- Trickier for composite types

Return statements

Go is a return-by-value language (i.e. the return value is copied into the calling function's stack frame).

- Easy for basic types
- Trickier for composite types

```
var a [5]int
var b []int // b = append(b, 0)
var c struct { f int; }

func foo() [5]int { return a; }
func bar() []int { return b; }
func baz() struct{ f int; } { return c; }

func main() {
    var d, e, f = foo(), bar(), baz()

    d[0], e[0], f.f = 1, 1, 1
}
```

What are the values for `a[0]`, `b[0]` and `c.f` respectively?

Return statements

Go is a return-by-value language (i.e. the return value is copied into the calling function's stack frame).

- Easy for basic types
- Trickier for composite types

```
var a [5]int
var b []int // b = append(b, 0)
var c struct { f int; }

func foo() [5]int { return a; }
func bar() []int { return b; }
func baz() struct{ f int; } { return c; }

func main() {
    var d, e, f = foo(), bar(), baz()

    d[0], e[0], f.f = 1, 1, 1
}
```

What are the values for `a[0]`, `b[0]` and `c.f` respectively? **0, 1, 0**

Print statements

Print statements in Go output zero-or-more printable expressions to `stdout`. In the case of `println`, they also:

Print statements

Print statements in Go output zero-or-more printable expressions to `stdout`. In the case of `println`, they also:

- Separate expressions by spaces

Print statements

Print statements in Go output zero-or-more printable expressions to `stdout`. In the case of `println`, they also:

- Separate expressions by spaces
- End with a newline

Print statements

Print statements in Go output zero-or-more printable expressions to `stdout`. In the case of `println`, they also:

- Separate expressions by spaces
- End with a newline

```
println(5, 4)  // 5 4    [newline]
print(5, 4)    // 54     [no newline]
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)
print(0377)

// Floats
print(0.12)

// Booleans
print(true)

// Runes
print('L')

// Strings
print("hello\n")
print(`hello\n`)
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)     // 255
print(0377)    // 255

// Floats
print(0.12)

// Booleans
print(true)

// Runes
print('L')

// Strings
print("hello\n")
print(`hello\n`)
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)     // 255
print(0377)    // 255

// Floats
print(0.12)    // +1.200000e-001

// Booleans
print(true)

// Runes
print('L')

// Strings
print("hello\n")
print(`hello\n`)
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)     // 255
print(0377)    // 255

// Floats
print(0.12)    // +1.200000e-001

// Booleans
print(true)    // true

// Runes
print('L')

// Strings
print("hello\n")
print(`hello\n`)
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)     // 255
print(0377)    // 255

// Floats
print(0.12)    // +1.200000e-001

// Booleans
print(true)    // true

// Runes
print('L')     // 76

// Strings
print("hello\n")
print(`hello\n`)
```

Print statements

What are the printing formats for basic types?

```
// Integers
print(255)    // 255
print(0377)   // 255

// Floats
print(0.12)   // +1.200000e-001

// Booleans
print(true)   // true

// Runes
print('L')    // 76

// Strings
print("hello\n") // hello [newline]
print(`hello\n`) // hello\n
```

Binary expressions

Binary expressions are the same throughout most languages.
Two possible exceptions:

Binary expressions

Binary expressions are the same throughout most languages.
Two possible exceptions:

- Integer vs. float division

Binary expressions

Binary expressions are the same throughout most languages.
Two possible exceptions:

- Integer vs. float division
- Bit clear (`&^`) may be missing

Binary expressions

Binary expressions are the same throughout most languages. Two possible exceptions:

- Integer vs. float division
- Bit clear (`&^`) may be missing

You should also implement string concatenation and comparisons.

```
var a string = "apple"
var b string = "Apple"

println(a + b)
println(a < b)
```

What does the above program print?

Binary expressions

Binary expressions are the same throughout most languages.
Two possible exceptions:

- Integer vs. float division
- Bit clear (`&^`) may be missing

You should also implement string concatenation and comparisons.

```
var a string = "apple"
var b string = "Apple"

println(a + b)
println(a < b)
```

What does the above program print?

appleApple
false

Call expressions

Go is a pass-by-value language (i.e. function arguments are copied into the new stack frame).

Call expressions

Go is a pass-by-value language (i.e. function arguments are copied into the new stack frame).

- Easy for basic types

Call expressions

Go is a pass-by-value language (i.e. function arguments are copied into the new stack frame).

- Easy for basic types
- Trickier for composite types

Call expressions

Go is a pass-by-value language (i.e. function arguments are copied into the new stack frame).

- Easy for basic types
- Trickier for composite types

```
func foo(a [5]int, b []int, c struct{ f int; }) {
    a[0] = 1
    b[0] = 1
    c.f = 1
}

func main() {
    var a [5]int
    var b []int // b = append(b, 0)
    var c struct { f int; }

    foo(a, b, c)
}
```

What are the values for `a[0]`, `b[0]` and `c.f` respectively?

Call expressions

Go is a pass-by-value language (i.e. function arguments are copied into the new stack frame).

- Easy for basic types
- Trickier for composite types

```
func foo(a [5]int, b []int, c struct{ f int; }) {
    a[0] = 1
    b[0] = 1
    c.f = 1
}

func main() {
    var a [5]int
    var b []int // b = append(b, 0)
    var c struct { f int; }

    foo(a, b, c)
}
```

What are the values for `a[0]`, `b[0]` and `c.f` respectively? **0, 1, 0**

Append expressions

Recall: Slices are dynamically sized containers of homogeneous data implemented using a header and an underlying array.

The `append` built-in function adds data onto the end of the underlying array, and updates the header.

- If `len < cap`, the same underlying array is used
- If `len == cap`, a new underlying array is allocated and the data copied

Beware! This creates very unnerving behaviour if you're not careful (and of course we test it).

Append expressions

Slice growth

How does the capacity/length change over time?

```
var a []int

for i := 0; i < 10; i++ {
    println("Cap:", cap(a), ", len:", len(a))
    a = append(a, 0)
}
```

Append expressions

Slice growth

How does the capacity/length change over time?

```
var a []int

for i := 0; i < 10; i++ {
    println("Cap:", cap(a), ", len:", len(a))
    a = append(a, 0)
}
```

Cap: 0 , len: 0
Cap: 2 , len: 1
Cap: 2 , len: 2
Cap: 4 , len: 3
Cap: 4 , len: 4
Cap: 8 , len: 5
Cap: 8 , len: 6
Cap: 8 , len: 7
Cap: 8 , len: 8
Cap: 16 , len: 9

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
b = a

// ‘a’ and ‘b’ headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 1)
```

What are the length and capacity of `a` and `b`?

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
b = a

// 'a' and 'b' headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 1)
```

What are the length and capacity of `a` and `b`?

a: len=2, cap=2
b: len=1, cap=2

Interestingly, `b[1]` is out of bounds.

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
b = a

// 'a' and 'b' headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 1)
b = append(b, 2)
```

What are the values of `a[1]` and `b[1]`?

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
b = a

// 'a' and 'b' headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 1)
b = append(b, 2)
```

What are the values of `a[1]` and `b[1]`?

Both 2

Yes, we can overwrite data if we're not careful!

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
// a = append(a, 1)
b = a

// ‘a’ and ‘b’ headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 13)
a[0] = 1
```

What are the values of `a[0]` and `b[0]`?

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
// a = append(a, 1)
b = a

// 'a' and 'b' headers: len=1, cap=2, ptr=0xDEADBEEF

a = append(a, 13)
a[0] = 1
```

What are the values of `a[0]` and `b[0]`?

Both 1

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
a = append(a, 1)
b = a

// 'a' and 'b' headers: len=2, cap=2, ptr=0xDEADBEEF

a = append(a, 2)
a[0] = 13
```

What are the values of `a[0]` and `b[0]`?

Append expressions

Edge cases

```
var a, b []int
a = append(a, 0)
a = append(a, 1)
b = a

// 'a' and 'b' headers: len=2, cap=2, ptr=0xDEADBEEF

a = append(a, 2)
a[0] = 13
```

What are the values of `a[0]` and `b[0]`?

a[0] = 13, b[0] = 0

Yes, we can change the underlying array of one header but not another!

Length expressions

The length built-in supports the following types:

- Strings
- Arrays
- Slices

Given an expression, it returns the current number of elements. For strings and arrays this is easy.

The length of a slice uses the header information and not the size of the underlying array.

Capacity expressions

The capacity built-in supports the following types:

- Arrays
- Slices

Given an expression, it returns the allocated number of elements - again easy for arrays.

The capacity of a slice uses the header information and returns the size of the underlying array.

Cast expressions

Easy! But be sure to correctly implement string casting.

```
var a int = 65
println(string(a))
```

What is the output of the above progam?

Cast expressions

Easy! But be sure to correctly implement string casting.

```
var a int = 65
println(string(a))
```

What is the output of the above progam?

A

Order of evaluation

Go uses left-to-right order of evaluation in *most* instances.

Implementing the correct order of evaluation if your language is different (e.g. C or C++) is very hard, so it is **not** required.

```
var a int = 0

func foo() int {
    a++
    return a
}

func main() {
    var b, c, d int = foo(), a, foo()
}
```

What are the values of `b`, `c` and `d`?

Order of evaluation

Go uses left-to-right order of evaluation in *most* instances.

Implementing the correct order of evaluation if your language is different (e.g. C or C++) is very hard, so it is **not** required.

```
var a int = 0

func foo() int {
    a++
    return a
}

func main() {
    var b, c, d int = foo(), a, foo()
}
```

What are the values of `b`, `c` and `d`?

1, 1, 2

A nice, simple, understandable outcome which is perfectly left-to-right. But then...

Order of evaluation

```go
var a int = 0

func foo() int {
    a++
    return a
}

func main() {
    b, c, d := foo(), a, foo()
}
```

What are the values of `b`, `c` and `d`?

Order of evaluation

```go
var a int = 0

func foo() int {
    a++
    return a
}

func main() {
    b, c, d := foo(), a, foo()
}
```

What are the values of `b`, `c` and `d`?

1, 2, 2

Go decomposes the expressions and evaluates all function calls *before* other operations in assignments and short declarations.

Order of evaluation

We can also look at the order of operation with logicals.

```
var g int = 0

func bar(a string) int {
    println(a)
    g++
    return g
}

func main() {
    var a, b, c = bar("lhs1") == 2 || bar("rhs1") == 3,
g, bar("call3")
}
```

In which order are the functions called, and what is the value of `b`?

Order of evaluation

We can also look at the order of operation with logicals.

```
var g int = 0

func bar(a string) int {
    println(a)
    g++
    return g
}

func main() {
    var a, b, c = bar("lhs1") == 2 || bar("rhs1") == 3,
g, bar("call3")
}
```

In which order are the functions called, and what is the value of `b`?

lhs1, rhs1, call3, and **2**

A nice, simple, understandable outcome which is perfectly left-to-right. But then...

Order of evaluation

```
var g int = 0

func bar(a string) int {
    println(a)
    g++
    return g
}

func main() {
    a, b, c := bar("lhs1") == 2 || bar("rhs1") == 3, g,
bar("call3")
}
```

In which order are the functions called, and what is the value of `b`?

Order of evaluation

```
var g int = 0

func bar(a string) int {
    println(a)
    g++
    return g
}

func main() {
    a, b, c := bar("lhs1") == 2 || bar("rhs1") == 3, g,
bar("call3")
}
```

In which order are the functions called, and what is the value of `b`?

lhs1, call3, rhs1, and **3**

Go decomposes the function calls on the LHS of logical operators, and leaves the RHS untouched.

Recursive types

Recursive types are also quite tricky depending on the language - C++ being hard. We will not evaluate this feature.

Useful addresses

- `http://golang.org`
- `http://play.golang.org`
- `http://golang.org/ref/spec`

References

- Gopher: `http://golang.org/doc/gopher/frontpage.png`
- Vincent Foley-Bourgon
- David Herrera
- Classes of 2015-2019

Advice

- This is a project that takes a lot of time: start early!
- Pick an target language that you know well enough to not get painted into a corner.
- Don't be afraid of asking questions and using the Facebook group.
- Build a test set of semantics programs using the slides and test often!

Gophers!

Thanks Google :)