Song Yang, Charles Gingras, Gregory Theos Group 05
COMP 520

Milestone 3: Semantics of GoLite

1. Introduction

For this third milestone, we fixed bugs left over from the previous milestone and start analyzing the semantics of the GoLite language. We also did some codegen implementation. The chosen target language is C. SONG ARE YOU GOING TO SUBMIT THE REPORT?

Advantages of the C language:

- Enough proficiency in C among the team members
- Same language as the compiler tools used (*flex*, *bison*)
- Fast compilation and running-time
- Statically typed

Disadvantages of the C language:

- Poorly informative run-time errors, e.g. segmentation fault
- Absence of data structure libraries
- Absence of object-oriented features
- Major differences with the GoLite language

2. Pattern I: Scoping Rules

Go is scoped statically and does not allow redeclaration of identifiers within the same scope. However, if an identifier is redeclared in an inner scope, Go allows it as *shadowing* and all queries and modification of this identifier in this inner scope now refers to this new definition.

2.1. Functions

Function parameter bindings belong to the scope of the function. Hence, variables that share any of these bindings cannot be declared within the immediate scope of the function. The C language has the same behaviour.

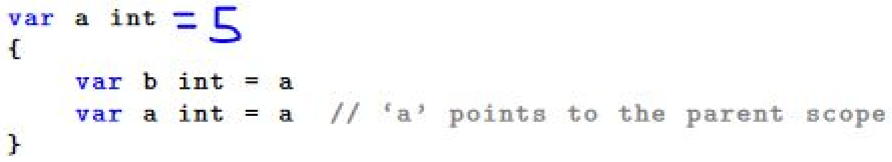
Function identifiers belong instead to the scope where the function is declared. The body can be used to shadow the function identifier. In C, the same thing applies.

2.2. Blocks

A block is syntactically surrounded by a pair of matching curly braces or within a case body. Examples of blocks are *if* blocks, *else-if* blocks and *for*-loop blocks. Every block contains a new inner scope. Since some blocks enclosed by curly braces may contain a simple statement, notably *if* blocks and *switch* blocks, an intermediate scope is created to hold the bindings the simple statement may create. This intermediate scope stands between the outer scope and the inner scope of the block body. Thus, shadowing of bindings created in the intermediate scope is possible inside a block body scope.

In our target language C, simple statements do not exist. Thus, an outer scope containing the simple statement has to be made to imitate the intermediate scope.

Furthermore, in Go, shadowing outer variables in blocks is possible and behaves as expected. In C, attempting to shadow variables from the inside of an inner scope is more complicated. For example, on a modified version of slide 13 from the tutorial,



```
var a int = 5
{
     var b int = a
     var a int = a  // 'a' points to the parent scope
}
```

the variable *a* inside the block will assume the value 5 of the outer declaration, before a shadowing attempt is made, *e.g.* on the line where *b* is declared. This is the correct behavior intended in Go. However, on the line where *a* is redeclared, *a* actually takes the value of 0, despite initializing it to the outer value of *a*. The 0 value is the default initialization value for type *int* in C. In general, the same variable to the right-hand side of a shadowing redeclaration takes the default value of the redeclared type, not the previous binding's value. In other words, C evaluates the declaration before the expression to the right-hand side of the assignment. This makes *a* in our example lose its previous binding of 5.

The proposed solution in C is to rename the redeclared variable. This way, the right-hand side of the assignment is still holding values of the previous binding. We need to be careful that the renamed variable does not clash with other variables in the scope, and that all subsequent uses of the shadowed variable are renamed as well. This requires appending a special suffix designed for shadowed variables and the suffix must be unique enough to not conflict with other variable names.

2.3 Control Flow

2.3.1 If Statements

In Go, every *if* block-like statement creates its own inner scope for its body, with the possibility of an intermediate scope as described above. If there is a following *else-if* statement after an *if* statement and this *else-if* statement contains a simple statement, the resulting intermediate scope of the *else-if* simple statement is created in addition to the previous simple statement's own scope. In this second intermediate scope, all bindings of the previous intermediate scope are reachable and can be shadowed, but the previous intermediate scope cannot access anything created in the second intermediate scope. Therefore, the second simple statement forms an inner scope within the first intermediate scope.

The body scopes of every condition statement are mutually unreachable. As such, the variables in the body of an *else-if* cannot shadow the variables in the related *if* statement body, and vice-versa.

In C, there are no simple statements and we use the approach described in the previous section if we are only mapping a single *if* statement. When there is a chain of *else-if*'s following an *if* statement, we can pretend that they are actually nested *if-else* statements, where the following *else-if* is contained in the *else* statement. As such, our C translation will not use C's own *else-if*, rather nested *if-else*. This implementation guarantees that every simple statement forms an intermediate scope nested deeper and deeper, which allows shadowing, and that all body scopes are mutually unreachable.

2.3.2 Switch Statements

Another control flow in Go is the *switch* statement. In Go, as described in the *blocks* section, a *switch* body may have an intermediate scope and every case statement creates its own inner scope. The conditions of the case statements are still in the parent scope, namely the *switch* body. Go supports a *default* statement inside a *switch*, which can be placed in any order among the cases, and like any case statement, it also creates its own inner scope. Because every case has its own inner scope, bindings in different case body scopes are mutually unreachable.

In C, there is no simple statement for *switch* statements. We can use the same approach as for *if* statements here. Also, a C default statement must be placed at the end of the switch body to have its intended effect. Furthermore, all C case bodies, including the default, are contained within the same scope. Therefore, it is possible to redeclare the same variable in another case, which is a C compile-time error, if we naively translate Go case bodies. To solve this problem in C, we create an inner scope for each case and the default.

3. Pattern II : Single Variable Declaration

3.1 Implicit Initialization

In Go, all declared variables are initialized to a default value, unless given an explicit value to initialize. In C, variables are not always initialized, when not given an explicit value. To resolve this problem, we will explicitly initialize every variable in C, according to the following table.

Go Type	Default Value
int	0
float64	+0.000000e+00 (3 zeros to exponent, if in the Go playground)
string	“” (empty string)
rune	0
bool	false

In the case of Go arrays and slices, the variable does not have a default value. In the case of Go structs, every struct field is initialized recursively. The variable holding the structure itself does not have a value. In C, we do not need to worry about the value of an array or slice, since it does not exist in Go. Elements of these grouping types are another story (not covered in this section). In C, structs are continuous blocks of memory and its fields might not be initialized. To ensure its fields are initialized like in Go, we follow every struct variable declaration with an explicit initialization of its fields. Because the struct may contain another struct, we initialize its fields recursively.

For all user-defined types in Go, the resolved underlying type’s default value is used to initialize. In C, this means that we treat all user-defined types as a Go base type.

3.2 Shadowing of *true* and *false*

In Go, the boolean constants of *true* and *false* can be shadowed. In C, there is no shadowing for boolean constants. Since all boolean values in C are actually integers, it is not possible to shadow integer literals. To implement the behavior of shadowing *true* and *false*, new temporary variables for the shadowed values of *true* and *false* are used in all scopes affected by the shadowing. We will keep a pointer that is passed along traversals of the AST to indicate the new values of *true* and *false* in the inner scopes.

4. Pattern III : Identifiers

4.1 Blank Identifiers

In Go, a blank identifier is the underscore character sequence (_). Although it is grammatically equivalent to any identifier, its semantics are unique. This is already a major difference with C, where the blank identifier behaves just like any other C identifier.

In Go, a blank identifier can be used to ignore the assignment value of an expression. Since the blank identifier is not entered into the GoLite symbol table, it can be redeclared many times. As such, we cannot just map all blank identifiers to a unique C identifier. In addition, a Go expression can be a function call that introduces side-effects outside of the assignment to a blank identifier. So an assignment to a blank identifier cannot be ignore completely. To translate these usage of blank identifiers to C, we use a new temporary variable for every assignment to a blank identifier.

Blank identifiers in Go can also be used in a function's parameter definition. Semantically, this means that these parameters are ignored when the function executes. Like the assignment usage, these blank identifiers need each a separate C temporary variable to prevent naming conflicts.

In the cases where a Go blank identifier is used as the identifier of a function declaration and type declaration, the entire function and the type definition can be ignored, because blank identifiers can never be accessed. In C, this means that we don't translate these blank identifier usage.

All other usage of the blank identifier is syntactically invalid and thus weeded by our compiler.

4.2 Keywords

In Go, variables can be named after C keywords. Thus, *var restrict int* is syntactically valid in Go, but if translated directly into C by keeping the variable name, it produces a compile-time error. So we will rename the identifier in C by appending a prefix to every user-defined identifier in Go.

Since we said that we will append a suffix for shadowed variables, it is wiser to append a prefix for all identifiers. This makes the generated code more easily examinable.

5. Work So Far

We have implemented the part of the codegen which deals with declarations. Slices and arrays will be implemented as structs that contain size information (and capacity in the case of slices) along with a value array. Consistently naming the array as "val" should make it simpler to generate code that accesses the array by inserting a ".val". For instance, a[5] would become a.val[5]; b[3][5] would become b.val[3].val[5]; and so on.

Declarations are carefully analyzed so that if a variable, type or function declaration is encountered with the blank identifier, it is ignored. An exception to this is when the right hand side of the assignment is a function call. Function arguments denoted by the blank identifier are ignored as well. A declaration that would introduce a binding that is a reserved keyword, such as “int”, is modified to instead use a generated identifier. This information is kept for later stages by being stored in the structure of the symbols themselves.

Since functions can sometimes return slices or arrays, which are represented by structs in C, we need to create struct definitions in such cases. To keep track of these bindings, the symbol table was extended to keep track of the new bindings by mapping symbols to the struct name in C. These mappings are kept separate from those made during the symbol phase of the compiler.

We have also implemented basic assignments for primitive types as well as if/elif/else statements. Assignments work much the same way as in Go; the only major difference is that strings must be copied using strcpy() in order to achieve the desired semantics. If statements are handled similarly in both languages; however because of scoping issues with short statements that can be put in if clauses elif statements are represented as else{ if(condition){ }}, with the optional short statement occuring in the else scope but not in the if scope. In other cases, short statements are simply placed one line before an if or else.

Statement of Contribution

Song: Test files, report, some bug fixes left from milestone 2
Charles: Bugfixes from the typechecker, codegen of declarations, codegen of expressions and implementation of logic behind alternate binding generation
Gregory: Codegen for expressions and statements.