Project Report - Milestone 2

Group 7

Neil Pereira

Denali Relles

Yann Sartori

Organization and division of work

We organized ourselves in the following manner : We first collectively created a tentative skeleton for how our symbol table would be organized, and then we would individually decide to update the symbol table to accommodate aspects we may have overlooked, or to make implementing certain tasks easier. Otherwise, Denali handled the declaration portions (all of 2), Neil the statement portion (all of 3) and Yann the expression portion (all of 4), as we did for the last milestone.

In fixing the previous errors involving parsing for semicolons after *all* statements, we had to consult the following source: https://stackoverflow.com/a/21631261 from user VolodymyrZubariev in order to perform actions on the end-of-scanning. We did not consult any other sources for this phase, except for our past work and the GoLang documentation.

Changes from the previous milestone

Our first changes addressed errors we committed in the previous phase. This included (1) Changing our regex for both types of strings to disallow the newline literal, and then to make sure that each / is followed by a valid character in an escape literal, (2) Doing a semicolon insertion check after scanning a multi-comment, (3) Decrementing our line count before doing an automatic insertion and incrementing it after, (4) Making our parser require semicolons after every declaration, (5) Ensuring that only one type could be declared at a time, (6) Checking for parentheses in the LHS of short declarations, (7) In the previous milestone, increment statements (x++) for example were represented as x += 1. As the reformulation was not equivalent from the perspective of type checking, the AST was extended to include special node types for increment and decrement statements. (8) Compound statements such as x += 1 were represented as x = x + 1 in the previous milestone. As the reformulation was not semantically equivalent, the AST was extended to include special node types for compound statements. (9) The grammar concerning increment and decrement statements in the previous compiler was incorrect (only identifiers could be incremented or decremented and not expressions). (10) "Distributed" variable declarations are now split up to allow "var (x = 0; y =x;)" without allowing "var x, y = 0, x"

In addition, we modified our AST to first include line numbers for all nodes. This allows us to provide more descriptive error messages. We also added a field of type PolymorphicEntry in expressions which is a pointer to an identifier's entry in the symbol table, null if the expression is not an identifier. This is covered in greater detail further on.

Explanation of the issues in each program

The following programs check the symbol table: Programs *duplicateType.go*, *duplicateVar.go*, and *duplicateTypeVar.go* check that, within a scope, we cannot redeclare types, redeclare variables, nor declare an identifier as a variable that has previously been declared as a type. Programs *undeclaredType.go* and *undeclaredVar.go* check to make sure when we reference an identifier as either a type or a variable respectively, they actually exist (this doesn't check for correct usage). Program *structFieldExists.go* checks to make sure that if we try to access a field α from a struct variable, then the struct has a field called α.

Program *duplicateFunctionParameters.go* tests that function parameters must have unique identifiers. Program *duplicateStructFields.go* tests that fields within a struct must have unique identifiers. Program *initWithArgs.go* tests that the special function init cannot be declared with arguments. Program *mainAsVariable.go* tests that "main" cannot be declared as a variable at the toplevel. Program *mainWithReturnType.go* tests that the special function main must have void return type Program *shortDeclNoNewArgs.go* tests that a short declaration must declare at least one new variable

The following programs check the type checker: Program *notMatchingType.go* checks for type equality in binary expressions. Program *notUnderlyingNumericType.go* checks to make sure that the - operation receives types that resolve to numeric types. Program *validIndexing.go* checks to ensure that we are only trying to use bracket indexing on variables that are either of a slice or array genre of type. Program *validStructAssignment.go* checks to make sure that if we assign a value α_1 to a field of type τ of some struct variable, α_1 has type τ. Program *emptySwitchExpressionDefaultstoBool.go* checks that switch statements with no expression condition default to bool. Program *failedLvalcheck.go* checks that the left hand side of an assignment statement is addressable (i.e. can be assigned to). Program *incrementNonNumerictype.go* checks that only numeric type expressions can be incremented. Program *invalidAssigntype.go* checks that the left and right sides of assignment statements must be of the same type.

Program *invalidIfCondition.go* checks that the condition of an if statement resolves to a bool type. Program *nonBoolLoopCondition.go* checks that the condition expression of a for loop resolves to a bool type. Program *nonComparableTypeSwitch.go* checks that the expression in a switch condition must be of a comparable type (in this case slice types were chosen). Program *nonPrintabletype.go* checks that only printable types can be printed (struct types cannot be printed) Program *switchExpressionMismatch.go* checks that the type of a switch condition expression and the types of expressions in switch case clauses are equal. Program *typeCastAreNotExpressionStmts.go* checks that type casts are not considered to be function calls and hence not valid expression statements,

Program *badAppend.go* tests that append cannot append an item of the wrong type to a slice. Program *mustReturn.go* tests that a function with a non-void return type must have a return statement. Program *nonCapacityType.go* tests that you cannot take the capacity of an int. Program *nonMatchingStructFields.go* tests that you cannot assign one variable to another, both structs, if the types do match exactly

Decisions made regarding the symbol table structure

Similar to in class, we stored symbols in a cactus stack of a struct called Context. Within a context, we stored the parent context, a hash table for types, and a hash table of symbols (variables and functions). We chose hash tables for their efficiency. The reason we separated types and symbols as opposed to storing them together in the same table was due to the structural differences of each type. This separation also has the added benefit of improving the clarity of the code. The reason we coupled the hash table for types and the hash table for symbols in each context was for scoping reasons-- you can shadow types with variables but you cannot redeclare them. The reason we included functions in the symbol table was because they exist only in the global scope, so we found it unnecessary to have a seperate table (as the specifications seemed to suggest). However, due to the separation of types and symbols, an identifier can refer to either a symbol table entry or type table entry. Hence, to be able to return either a symbol entry or a type entry when searching the context, we created a new struct called polymorphic entry (effectively a sum type) that would let us store both types.

A symbol table entry stores the id used for lookups, a pointer to its type, and a next pointer to handle hash table collisions. The type table entry stores an id and a next pointer for the same reasons, but also stores an "underlyingType", which, similar to the AST, stores if it is an array, struct, etc. An important difference however is that identifierType refers to a non composite type. In the AST creation, if we had "type A B", A would have a field with value identifierType, regardless of what genre of type B is (array, struct…). Now, in the symbol table construction, if we have "type A B", the value actually depends on that of B. Based on this, we have a union of the information needed for each type-- nonComposite is the base type to which it resolves, sliceType the type a slice stores, arrayType the type stores with its dimension, a context entry for structs, and a list of symbols for the arguments (we use the next pointer of STEntry that is normally for bucket collisions for the list) and a return type for the function type.

The most noteworthy decision here is that we immediately resolve types and store them in the type node. This makes our type resolution slightly quicker, but in addition, since we use strict type equalities (explained more later) in everything expecting a value, if we have "type A B", it doesn't actually matter that A is defined in terms of B-- all that matters is that type A gets the base type of B. In addition, we stored a context for structs for the convenience that, though structs do not introduce scopes, they have name spaces, so we can reuse the code. Finally we store symbol entries for functions because we need both the ID and its type for each parameter for within the scope it introduces. These decisions were made to facilitate and meet the requirements of the type checking phase.

Program scoping

At the lowest level, a scope is introduced to hold the base types: int, float64, string, bool, and rune; and the constants true and false. We also have a global pointer to the base types because they need to be retrieved later. A new scope is immediately put on top of it to hold top level declarations. The following introduce new scope: (1) Function declarations : Functions declare new scope. Special handling of the body was required so as to ensure that the body of the function reuses the parent scope. This is achieved through the use of a flag variable. (2) Block Statements. (3) Switch Statements : all switch statements introduce new scope to permit optional short declarations. The new context is used to symbol check switch condition expressions and all contained switch case clauses. (4) If, else Statements : If statements introduce new scope to permit optional short declarations. This new scope is used to symbol check the body of the if statement as well as any optional else blocks. (5) Loop Statements : Loop variants that permit optional declarations introduce new scope, the infinite loop as well as the while loop do not permit the declaration of new variables so the only new scope associated with the loop belongs to the body of the loop.

Decisions made regarding the symbol table creation and check

The symbol checking for expressions was relatively straightforward. If our expression was an identifier, we checked that it existed either in the current scope or in a higher one, in which case we set that expression's context entry pointer to the retrieved value. Otherwise, we just recursively traverse the components of each expression, until we hit an expression with no more components (like literals). The only slight exception was for field selects. Since the accessor was weeded to be an identifier, we did not perform any action on it and deferred to the type checker.

We actually had to add a couple flags into structs for one-time use things. We added a "constant" flag in the STEntry struct that was turned to 1 for "true" and "false" at the base scope to help later when we were checking if you could assign to a variable. This flag is also set to 1 for functions, which can't be reassigned, although this would also have been caught because they are functions. There is also another flag in the TTEntry struct for "comparable", which is set in the following fashion : only arrays, structs, and non-composites are comparable, and for arrays, an array is comparable if and only if the type it stored was comparable. For structs, they were comparable if and only if all their fields were comparable. This came directly from the GoLang specifications.

Recursive types required some thought, but the solution isn't too bad: the function to create a TTEntry changes based on whether you're declaring a type or just using an anonymous type, and if it's a type declaration the name of the type you're declaring is passed down. There's also a flag to check if the recursive flag is in a slice.

Decisions made regarding the type checker

In regards to expressions, we first declared some helper functions that checked the genre of the underlying type (isNonComposite, isNumeric, isInteger, isBool, and typeEquality). For comparisons, we just checked the field 'comparable' of the type, which was set when we created the symbol table. For type equality, since we did not use type aliasing, in almost all cases, we could just compare the pointer values. This is because two declared types are equal if and only if they *refer* to the same type. The only slight caveat to this was when we had anonymous types (for instance, "var x struct {a int;}"), in which case we just checked if the identifier of the type pointed to null for both and then recursively checked to make sure sub-components were equal (size of arrays, underlying types for arrays and slices, underlying members of structs). Structs posed a sizable challenge: since we had stored the struct members in a context, we would have to iterate over the whole hash table to compare the members. To avoid this, we also stored a linked list with the names of the members for quicker checking.

With these helper functions, the majority of the type checking was relatively simple. The predicates for the type inference of unary and binary expressions correspond directly to the helper functions (and accessing the isComparable for comparison). Expressions that correspond to indexing, length, capacity and append involve essentially just checking the fields of our types to match the typing predicates, and returning the proper types in accordance to the typing rule. When we have literals, we just return the global base types (mentioned prior). And for identifiers, we just return the type in the polymorphic entry field. However, since types are not values, if the field is a type table entry, we change the value of a boolean pointer that is accessible in all recursive calls to true. This is used to ensure that (1) we are not trying to do invalid operations with the type, such as int + 5 and (2) for verifying type checks. This boolean gets reset when we actually use a type in the way it should-- only the case for casts.

The other types of expressions were slightly more complex, however. For field access, we verify the type of our accssé to be a struct. Then we must look within the context stored if there is a symbol entry that shares the ID of the assessor, in which case we return the type of the entry. Since we do not set a parent for the context, we can actually reuse our lookup method from the symbol table phase. For function calls, we first check that our base is an identifier-- this disallows for instance (1+1)(1+1). If that passes, we then typecheck our base. If our base has underlying type funcType, then we then traverse the STEntry of the function (c.f. symbol table structure) and the arguments in the expression, and verify that those in the expression type check to those of the type entry. If at any point we exhaust one list before the other, we know we passed in an inappropriate number of arguments. Otherwise, if our base is not of type funcType, then we check our boolean type pointer directly after type checking the base. If it is true, we know we are attempting a type cast, so we then verify that we only have one argument, in which case we make sure that the conversion is valid as specified by the Golite rules in which case we return the new type.

At the type checking phase of the compiler, in addition to type checking the body of functions, we check that every function that returns a non void type ends in a terminating statement as outlined in the GoLang specification.

Assignment statements required two additional checks. The left hand side of an assignment statement needed to be checked for addressability and the right hand side of the statement need to be check for assignability (for example to prevent a function from being assigned to a variable)