GoLite Milestone 2

James Brace, Paul-Andre Henegar, Youri Tamitegama

Transitioning from Milestone 1

In this milestone, we were instructed to create a typechecker that feeds off of our lexer/parser that we implemented in the first milestone. We started off by first fixing the few errors we had in the first milestone, then transitioned into the second milestone by laying down some rudimentary boilerplate code for traversing the AST that was generated by our parser and constructing a symbol table that annotates the type of each declaration.

Design Decisions

Language

For this phase of the project, we continued to use Rust as our language of choice. The reason we chose to use Rust instead of another systems language such as C or C++ was mainly because of its memory safety. Rust provides a powerful compile-time error checking tool (cleverly called the borrow checker) that resolves a lot of data-race and dangling pointers errors that are rather common in languages that give users the power to oversee memory management. In fact, ever since we have started coding this project, I don't believe we have gotten a single runtime error (that wasn't logic-related that is). In addition to a rather insightful compiler, safety is guaranteed via immutable data and Smart pointers which are freed only when they go out of scope. This forces a strict concept of "Borrowing" that allows the compiler to nicely track which scope "owns" which values. Lastly, another reason we used Rust was for the experience to play around with a new language that is becoming widely used and loved in the software engineering community.

Type representation

In order to properly perform type-checking, we created a Rust module that defined the types specified in Go. Given that 'type' is a keyword in Rust we had to use the synonym 'kind'. After the fact, it was probably the best of ideas since we still ended up using both 'kind' and 'type' in our variable names. The loose standard we had was that we would use 'type' whenever we were referring to something that was explicitly defined in the specs, such as "type declaration" or "type definition" or "type-checking" and used 'kind' when we were referring to our own internal representation of Go types.

In our 'kind' module, we defined all the possible Go types using Rust structs and enums and defined some basic operations on them, like 'resolve', 'are_identical', 'is_numeric' etc…

The central type was **Kind**, which represented the possible values a Go type can have, as well as some helper values:

- **Basic(BasicKind)** - used for the GoLite basic types (int, float64, etc…)
- **Defined(Rc<Definition>)** and later **Defined(Rc<RefCell<Definition>>)** - used for types that are declared at runtime. It is a essentially a pointer to a heap allocated object representing a type definition.
- **Array(Box<Kind>, u32) -** used for arrays. It has a pointer to the Kind of the elements it contains, as well its size (u32).
- **Slice(Box<Kind>)** - used for slices. It's the same as for arrays, but there is no indication of size.
- **Struct(Vec<Field>)** - used for structs. It contains the vector of all all its fields.
- **Undefined -** the type we use for expressions that have not been typechecked yet.
- **Underscore -** is a special type that only the expression containing the blank identifier will contain.
- **Void -** used to signal that we were calling a function with no return value

BasicKind was an enum containing all basic Golite types -- Int, Float, Rune, Bool, String. The variants were capitalized simply because it's a convention in Rust.

Structure types (i.e. Arrays, Slices, Structs) were represented by enum variants containing other types. Boxes were used to indicate that these enums each owned a unique copy of another heap-allocated *Kind*. So for example an int array is *Array(Basic(Int)))* and array of int arrays is *Array(Array(...))*. This nicely allows us to be able to resolve types when type-checking.

In addition to defining the different *Kinds*, we added a handful of utility methods to allow us to quickly resolve, compare, and evaluate them. For example we added functions that checks if two *Kinds* are identical, and whether a *Kind* is comparable, ordered, numeric, etc. This formed a clean object-oriented approach to working with *Kinds* and made the rest of type-checking rather easy.

Symbol Table

The first design decision we had to establish regarding the symbol table was whether we want to construct the symbol table in it's own pass and then typecheck, or do it at the same time. We decided to do the construction and type-checking in parallel since it was the most intuitive, and we couldn't think of any heavily-weighted pros for performing two passes, since we would have needed to infer the types of certain variables during the second pass anyway. Another advantage of this approach (perhaps indicating our misunderstanding of the purpose of a symbol table) is that we only need to keep pointers to the parent scope of the scope we are currently in. That way, our symbol table can acts as a stack and we don't need to create any mutual references which is non-trivial in Rust.

An important design decision we had to make was what to use to reference the parent. Rust doesn't allow recursive structures and there are various way to overcome this issue. What we did was use a simple Rust reference and specify a *Lifetime* to the borrow. A *Lifetime* is a construct the compiler uses to ensure all borrows are valid. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. This essentially tells the compiler that the lifetime of the borrow cannot outlive the current symbol table. A useful side-effect of using lifetimes was that since a child scope was guaranteed to be destroyed before the parent scope in a deterministic way, we could use the Drop trait (which essentially specifies a destructor) to close the braces for our scopes.

Our symbol table consists of a unidirectional linked list of *SymbolTable* structs that holds a pointer to their parent *SymbolTable* (i.e their enclosing scope), a hashmap of *Symbol* structs, an optional *return_kind* and *in_function* boolean that depicting if we're in a function and what the return type of that function is. In addition we have the fields, *level* and *print_table*, that are used in order to pretty-print the symbol table for debugging purposes. Since we do everything in the same pass, we pass a "print_table" flag to differentiate whether we are in the typecheck or symbol mode. In addition to aiding in debugging, *level* is also used to make sure init() and main() are only defined in the top-level declarations (more on this later!).

In order to define *Symbol*, we used a Rust enum that was mapped to an identifier (or more strictly speaking, an identifier was mapped to a *Symbol)*. Symbols contained a line number to where they were declared and a *Declaration*, which was an enum that specified if the symbol was a Variable, a Constant, a Type, a Function or a "Dummy", and contained the type of the variable, constant, type, or function. We decided to have functions be separate from variables since in Golite we cannot treat functions as values.

To make type-checking easier, we implemented multiple methods to access the symbol table, such as getting a symbol, checking if a symbol is declared in the current scope or adding something to the table. The methods that add Symbols to the symbol table take care of checking if the Symbols are being redeclared in the current scope, printing them when in 'symbol' mode and also deciding whether or not to add them to the symbol table if appropriate (for example exclude "_" or "init"). In addition, we had a method to create a new scope that would create a new SymbolTable struct to which it would pass a borrow to itself.

Type checking

Type-checking for the most part was straightforward since it just required going through the specs and applying the rules as defined. To typecheck, our code traverses the AST using pattern matching and recursion, saving types in the AST and returning them when applicable. Our code simply traverses through all statements, finds the expressions that make up the statements, traverse them more and then evaluates them. The sub-expression types then bubble up to the original expression which can be declared as well-typed or not. Most of the design decisions made for type-checking were trivial since it just involves traversing our AST properly and following the type rules laid out in the specs. A few of the non-trivial use-cases are discussed in the following section.

Our code for our type-check module was broken up into 3 main functions with a handful of utility functions. The 3 main functions were: *typecheck_kind*, *typecheck_statement*, and *typecheck_expression*. The rest of the functions were essentially utility functions used for traversing through the AST and adding to symbol table. The function *typecheck_kind* maps AST kinds to our symbol table Kinds that we discussed before. In addition, it also checks to make sure that recursive definitions don't exist. The function *typecheck_statement* looks at a statement and see whether or not the expressions it is made up of also typecheck. This leads us *typecheck_expression* which is in charge of doing all the type-check grunt work. It makes sure that every possible expression it is given type checks and then return the type of the resulting Kind. This allows for a nice recursive bubbling up of the *Kinds* of sub-expressions into their parent expressions.

Some not-so-trivial typing rules

- **Comparison of defined types**: Since the identity of defined types is given by the place they were defined and not by their name, we could be using multiple different defined types in a single scope. Since the type of a variable is a pointer to a type declaration, type "t" could be declared in some scope as well as in a parent scope, but these two types are in fact different. In order to get around this, every time a type was defined, we created a new heap-allocated object *Definition* containing a line number, name and base type. We would then pass around a pointer to this object inside the Kind::Defined variant of our *Kind* enum. This way, two instances of the same defined type would point exactly to the same location in memory, and we could check if they are equal using a single pointer comparison. Because of Rust's safety enforcement, we had to use a reference-counted pointer to the Definition object, using a Rc (stands for reference count). Later on, when we realized that we might want to change the definition when declaring a recursive type, we had to also use a RefCell, which allowed what is called "interior mutability", so that it could stay mutable even if it did not have a single owner.
- **Void return type**: we had to handle the case when functions have no return type. When inside of the scope of a function, we would use the AST to define a return_type of the current SymbolTable. Using this field we were able to tell if the return type of a function matched its declared return type. When evaluating a function-call, the only situation in which it is valid for the function to return void is in an expression statement. In that situation, we passed a flag to the typecheck_expression function to indicate this. In any other situation, typecheck_expression would raise an error when evaluating a function returning void.
- **init() and main() functions**: we needed to make sure that init() and main() were only defined as functions in the top-level declarations. When we were to come across a function with the name init() or main() we would use our "level" indicator to make sure

that we were in the root scope. In addition, we had to make sure main was only declared once, but this rule doesn't pertain to init().

- **Terminating statements:** most checks for terminating statements were straightforward, but there was one subtlety to address: in "for" and "switch" statements, a break statement could be nested within a child scope, in particular it is within a "block" or "if-else" statement. These use cases were handled in our weeding phase.
- **Function name scope:** when declaring a function, there was a specific situation where a binding for the function is created before the parameters are added to the symbol table. For example, "func int(a int) {}" should throw a typecheck error as "int" in the parameters declaration should actually point to the function named int. To solve this, we temporarily add a Dummy symbol in our symbol table, then replace it by the function once all the parameters are parsed.
- **Function calls and casts:** Because we parse function calls the same way as type casts, we had to detect during the typecheck phase whether thit was one or another, and change the type of the expression appropriately. This was painful to do in safe Rust since we were passing a mutable reference to the AST node and we had use mem::Replace to temporarily swap out the subtree that the AST node contained so that we could put them back in as a TypeCast.
- **Recursive types:** Initially, according to the reference compiler, we were not allowed to have a type declaration reference itself at all. For example, "type T struct { a T }" or "type T []T" were forbidden. To solve this, we kept track of the name of the struct currently being defined when we typechecked an AST subtree representing a type (the *top_name* parameter in *typecheck_kind*).
- **Recursive types (again):** After having completely disallowed recursive types in our compiler, we heard that we were now to accept recursive types provided that a slice was used as indirection. This made things more difficult because we had to link to a type while we were constructing it. We therefore created a Definition containing an <undefined> type and added it to the symbol table before starting to evaluate the AST subtree referring to the type. That way, we could link to the type definition from inside the type. Finally, once we got the final type, we would go back to the Definition and actually set the correct type that we have computed.

Team Organization

It's hard to give a clear separation of each member's contribution as we were often working together on the same things, but the approximate work repartition was as follows:

James - symbol table & statement traversal boilerplate + unary/binary expression typecheck + filled in various holes for type-checking and Kind methods
Youri - expression, statement, kind type-checking boilerplate
Paul - statement, function & variable declaration type-check + Kind + finding edge cases

Youri and James wrote a majority of the test cases while Paul used them to fix bugs related to expression type-checking.

We discussed and agreed on the code structure together.