COMP 520 - Group 11 Final Report

Shafin Ahmed - 260782651

Nathaniel Branchaud - 260800960

Brendon Keirle - 260685377

May 1, 2020

1. Introduction

Go is a programming language intended for use in low-level systems but with access to high-level functionality. GoLite is a subset of Go which omits much of the more complex high-level features supported by Go in order to create a language which is robust enough to be able to write useful programs but which is simple enough that a compiler for it can be designed in a single semester. Among the features supported by GoLite are structs, functions, variable shadowing, custom types, slices, arrays, and the control-flow structures of Go consisting of If-statements, loops, and switch-statements. This report will outline our decision making process while designing a compiler for GoLite, as well as describe how we supported some of the more complex features of GoLite. A section is include for each of the main components of the compiler pipeline, as well as one covering the languages and tools we chose to work with.

2. Language and Tool Choices

Our group considered numerous languages such as C, Rust, and Java. I believe we were all a bit daunted by the task of such a large project in C, and did not feel we would have time to learn rust as the project went on. Due to these reasons and the opinions expressed on various internet forums, we decided to implement our compiler in Ocaml. One major benefit of going with Ocaml is it's native pattern matching combined with Ocaml's type checking system made it easy to traverse nodes of our AST in a uniform manner. Static type checking allows for faster execution times since the compiler knows more about the program to be ran and can thus do more aggressive optimization

compared to dynamically type checked languages such as python. This language feature also allows type verification errors to not remain hidden if the buggy code is simply never used as happens in python. Choosing OCaml also allowed us to maintain access to high-level data structures such as strings, lists, and hash tables without sacrificing the aforementioned efficiency.

Our compiler uses menhir as our build tool which provides informative error messages and to build a single executable for our project using a Makefile.

The target language we chose for the codegen portion of the compiler was Java. Java was chosen since it is a language that all of us are very familiar with, and because it is similar to Go in many ways. Both Java and Go are statically typed languages. They both use similar access points into the program, i.e. a function/method called 'main'. They both have convenient support for high-level operations on strings. Their control-flow structures are quite similar. Java also has built-in bounds checking for its arrays. Given all this it seemed to us that generating Java would be the most reasonable option for us.

3.1 Scanner

Much of the design of our scanner was more or less straightforward with the exception of a few particular features. One such feature was the implementation of optional semicolons to terminate lines. Go allows user to optionally terminate lines in a program with a semicolon by having the scanner insert a semicolon into the token stream depending on whether or not a line ends with a particular token. To implement this feature, we defined a variable to keep track of the last token encountered by the scanner. Each time the token is encountered, this token is stores in the variable, then when the scanner encounters a newline character, it checks whether the last token was one of the ones requiring a semicolon to be inserted, and if it was it accordingly inserts a semicolon into the stream. This did have one edge case however, if the file ends with a character other than a newline and that character is one that would normally cause a semicolon to be inserted then nothing would be inserted and the program would be considered invalid. The fix for this was tho check if a semicolon needs to be added when the EOF character is reached and if so manually feed a newline character to the scanner so that the standard semicolon handling could occur.

Another difficulty that came up in the scanner was how to recognize all special characters in interpreted strings. The primary source of this problem comes from our decision to use Ocaml. When Ocaml sees what resembles a escape sequence in a string, it replaces backslashes with two backslashes if and only if the escape sequence is not known to Ocaml. An example of such coercion can be seen

with the input string: "Hi! \z \t " which is coerced to: "Hi! \\z \t". Here, Ocaml recognizes '\t' and does not change the escape sequence, but '\z' is not an escape sequence in Ocaml and so a second backslash is automatically inserted in this case. The problem that arises from this is that we are meant to throw errors when an illegal escape sequence is found, but this coercion makes it impossible to know whether the user entered the valid escape sequence: "\\i" or the invalid sequence: "\i". In the end this was solved when we realized that moving the matching of string contents to the lexer from a function allowed us to match the raw input with no interference from Ocaml.

One issue which did not appear until the codegen phase was handling octal integer literals. Go treats any integer literal consisting of just digits and starting with a zero as an octal literal. Both OCaml and Java treat such literals as standard decimal literals. As such, when we generated the token for such literals, OCaml would convert it to decimal automatically, and thus the output of our generated code would be incorrect. The solution to this was to check if any integer literals longer than one character start with a zero. If they do, we check to see if the second character is an 'x' specifying a hexadecimal literal or an 'o' specifying an octal literal. If this is case, the literal is handled as is. If not, then the sequence '0o' is prepended to the literal, forcing it to be treated as an octal as desired.

3.2 Parser

Our parser includes many recursively designed grammar elements. One of the most complex is the one covering variable declarations and assignments. Go allows users to declare or assign multiple variables in a single line by specifying them as a comma separated list. To implement this, we designed a recursive grammar element with a base case of a single identifier and corresponding expression, then allows identifiers separated by commas to be recursively added to the left-had-side of the element and similarly for expressions on the right-hand-side. This allows us to enforce the requirement that the number of identifiers on the left-hand-side equals the number of expressions on the right-hand-side without resorting to an additional check in the weeder.

Another example element supporting nested structs. Since Go allows structs to contain other structs as parameters, we needed a way for our grammar to support this. The solution was simply to consider the grammar element for a struct declaration as equivalent to any other uninstantiated declaration. This had the added benefit of also creating the grammar element to support anonymous structs. One problem with this however was that variable and type declarations use the same grammar element after matching their unique 'var' or 'type' keyword. In order to avoid making two AST nodes for struct declarations, we prepend a special character onto the identifier associated with the struct

declaration if the struct was declared as a type. The symbol table then assigns the appropriate type based on whether this special character appears at the beginning of the identifier. The grammar element to handle multi-dimensional arrays and slices works very similarly. The grammar elements for referencing these recursively defined variables, i.e. field and index expressions, are constructed in a similarly recursive way.

3.3. AST

Our main objective in designing our AST was to keep the number of node types low. In the end, we managed to keep this number down to three, statements, declarations, and expressions. This made sense since it (roughly) corresponds with the different kinds of grammar elements defined in our grammar, and because it (again roughly) corresponds with the way we divided the work among team members. The statement and expression types are fairly straightforward. One subtype exists for each kind of statement or expression supported in GoLite. The declarations are done similarly, with a subtype for each kind of declaration, i.e. typed declarations, inferred declarations, short assignments, type declarations, struct declarations, and so on. This eliminates the need for an explicit type node since type information is either unavailable, or stored as a string for the typechecker to decipher later. The decision to design the AST this way instead of using a type node was made in accordance with our goal of minimizing the number of node types in our AST.

We used lists extensively, with our AST being represented by a list of statement nodes, and block statements (e.g. If or For nodes) having statement lists corresponding to the statements contained in their scopes. Another place lists are used is in declarations. When the user declares multiple variables using the single line grammar element described above, it is converted into a list of individual variable declaration nodes. The downside of this is that the variables are assigned expressions in reverse order because of the recursive definition of the grammar element. This is addressed by traversing through a reversed copy of the list and re-assigning variables in the original list based on the new reversed order of the expressions. The use of lists in our AST allows much of the AST to be traversed tail-recursively, which is very convenient in OCaml.

3.4 Weeder

Our weeder simply goes through the AST recursively and when it enters a for loop or switch statement node, a reference variable 'loop' is incremented, and decremented when it exits the node.

This allows break and continue statement nodes to be checked against this variable, and an error is thrown if the loop variable is 0, i.e, if we are not in a loop. A similar thing is done in dealing with default case in switch statements by keeping track of the number of default cases encountered so far and throwing errors as necessary. Lastly, we also weed out any invalid uses of blank identifiers by matching the blank identifier string in incorrect nodes and throwing errors if there is a match.

To fix some of our parsing issues from milestone 1, we made use of the weeder. In particular, previously, our parser could not handle parentheses around identifiers in assignments. Adding a rule to allow left and right parenthesis before and after the identifier in the assignment would lead to shift/reduce errors. The solution we went with was to simply allow expressions in the assignment instead of just the identifier and weed out any expressions that were not identifiers or identifiers in parentheses. Other parsing issues of a similar nature, e.g. parentheses around types/function names/etc., were also fixed using the weeder in a similar way.

We also use the weeder to check whether functions that have a return type have a terminating statement at the end. This was fairly simple to implement as all we had to do was traverse through the AST to the tail of the statement list in the function and check whether it satisfies the conditions of a terminating statement. If, on traversing to the tail, we find a terminating statement and we still haven't reached the tail, then that means there is unreachable code and we raise an error.

3.5 Symbol Table

Our symbol table consists of a simple cactus-stack of hash-tables where in addition to a table for the current scope and a parent pointer each stack node has a value representing the return type of the current function (or that the current scope is not in a function). We decided that typechecking should happen alongside the building of the symbol table. Therefore, mappings in the symbol table are only added or modified if the associated AST node passes typechecking. The information stored in the symbol table is the type corresponding to the identifier mapped to it. This type symbol varies in complexity from a simple constant type for built-in types to more complex structures for functions and structs which also store the list of parameters/arguments and in the case of functions, the return type. The symbols for arrays and slices store the type stores within the structure as well as how many dimensions it was defined with. Arrays also store the size of each dimension. In terms of printing the symbol table, we simply print each new mapping as it is encountered. This means that the printed symbol table will be incomplete if an error is encountered, but given that our symbol table is implemented as a cactus-stack it is the simplest solution available to us.

The scoping rules we used are fairly straightforward. Each block (if/else, loop, function body, etc.) is treated as its own scope. The init statement in if/else and for loops are treated as their own inner scope. This allows us to shadow variables in the init statement properly, and the parent pointer of the body block simply points to this scope letting us re-declare variables from the init statement if needed. The parent pointer of the post statement in a for loop also points to the init statement's scope. When checking if an identifier has already been declared, only the immediate scope is checked allowing local variables to shadow global ones. When looking up the value of an identifier however we look through all preceding scopes before deciding the identifier has not been declared. For short declarations, we look at all identifiers in the current scope and keep track of whether there is at least one new identifier being declared. A function's arguments are added to the function's scope before the statements in the function's body are considered.

We decided not to augment the AST since the symbol table and typechecker from Assignment 2 we were using as a template did not and changing it to do so seemed like too much work for not enough benefit. Instead, once typechecking is complete we rebuild a more rudimentary symbol table during codegen since we know that the program typechecks. Though this is slightly inefficient, it suited our purposes well.

3.6 Typechecker

The typechecker's most important function is verifying type equality. Since GoLite only allows variables to be assigned to each other if their types match exactly, this is the notion of type equality our typechecker verifies. For built-in types this is trivial. For custom defined type it is more difficult. The symbol for custom defined types includes the name associated with the type and well as its underlying type, which can itself be a custom type. The two types are considered equal if they have the same name and underlying type. However, this does not take shadowing into account. If a type is shadowed a higher scope, GoLite requires that it be considered a different type to the one it shadows. To account for this, we generate a number for each identifier in an assignment based on the scope it was declared in. If two identifiers have equal types by the previous definition, but one is found to have been declared in a higher scope than the other, we know that if one was declared in a higher scope that the one associated with its type that it must have a type which is being shadowed in a lower scope, meaning the types should not be considered equal and an error should be raised.

Our typechecker verifies much more than if the type of a variable's specified value matches its declared type. For one, it checks that a struct or set of function arguments do not contain multiple

parameters with the same name. This is done by creating a list of the parameter names and associated types and ensuring that the names list contains no duplicates. Since checking membership in a list is very easy in OCaml, the implementation of this is fairly simple.

Another task our typechecker performs is ensuring that the specified type of a variable or type declaration actually refers to a type and not a variable. To do this, we needed to differentiate between declared custom types and variable declared as a custom type. The typechecker then simply needs to check that any specified type is either the former or a built-in type and raise an error if it is the latter.

For inferred declarations, the typechecker needs to do a bit of extra work in order to obtain the appropriate type for the symbol table. If the expression being assigned is a literal, then it simply returns the associated type of the literal. If it is an identifier, then it looks it up in the symbol table and if it is found, returns its associated type. Of course not all types can be assigned, such as the aforementioned declared types. Functions are another type which cannot be assigned, so if the type of the identifier turns out to be a function, the typechecker raises an error. This process gets complicated if the expression being assigned is a struct field. As mentioned above, the symbol for structs stores the list of their parameters including the names of the parameters and their type. In the case of a field reference, the typechecker looks us the first identifier, i.e. the one before the period and ensures it is indeed a struct. If it is, then it goes through its list of parameters looking for one with a name matching the second identifier, i.e. the one after the period. If it finds one, it returns its associated type, otherwise it raises an exception. This process is performed recursively in order to account for nested structs.

One special case the typechecker needs to account for is short declarations. Go allows variables to be both declared and re-assigned in a short declaration. However, it requires that at least one of the variables in the statement be new. To implement this, we defined a flag in the code handling declarations which keeps track of whether there has been a new variable declared. Since our parser transforms Go's multiple short declarations into a list of individual assignments, the code goes through this list and checks if each identifier exists in the symbol table yet. If one is reached that does not, then we consider it a new variable declarations and change the flag accordingly. If the end of the list is reached without the flag being changed, then an error is raised stating that no new variables were specified in the short declarations, as required. Expressions are type-checked by recursively extracting the types of all sub-expressions and verifying that their types fit into the overall expression. This method of evaluation was convenient for simple expressions such as binary and unary expressions,

but added complexity for appends, indices, and field expressions in particular. Index expressions which are arguments of an append expressions must be validated differently. Such index expressions must be indexed using 1 less dimension than the declaration of the object being indexed. Another consequence of this caveat is that although normally the types of expressions on either side of an assignment operator are computed without knowledge of the other.

3.7 Code Generator

Our code generator needed to handle the various differences between Go and Java. One such difference is that the main method in Java must be static. This means that any functions or variables referenced within it must also be static. Thus, the code generator needed to prepend the 'static' keyword onto any declaration made at the toplevel. Whether or not the current scope is the toplevel is determined by checking whether the symbol table for the current scope has a parent or not.

Another tricky case to handle is the difference in control-flow statements between Go and Java. Specifically, Go allows variables to be declared for use exclusively in the scope of the control structure while Java only allows this in 'for' loops. Luckily, Java allows for an arbitrary scope to be specified by enclosing the statements in braces. Thus, we could replicate the scoping of these variables by placing both the variable declaration and the appropriate control-flow structure in braces.

One big difference is that Go allows variables to be shadowed while Java does not. This meant we needed a way to differentiate between variables with the same name declared in different scopes. The solution we ended up using was to append a number to the end of the variable's name based on the scope it was declared in. Since variable names within scopes must be unique even in Go we can be certain that there would be no naming conflicts created because of this solution. However, since we did not augment our AST and thus had no stored data about a variable's scope beyond the symbol table itself, special care needed to be taken to ensure that the correct number was appended in the case where a variable was being referenced outside its initial scope.

Relatedly, both Go and Java allow variables to be declared without being initialized. However, Go implicitly initializes its variables with default values based on what type the variable is. Since Java initializes all uninstantiated variables to null, this simply meant that any uninstantiated declarations in our AST needed to be translated into declarations using the appropriate default value. These default values were determined by testing in the Go playground to see what variables of different types were initialized with.

Another difference to handle was the difference in output when printing in Go and Java. Go's

print and println statements have very close approximations in Java in its System.out.print and System.out.println statements. However, there are two key differences in how they output. In Go, runes are printed as integers and floats are printed in exponential notation. Since we were using Java's char primitive type in place of Go's runes, this meant that the solution to the first of those differences was to cast any rune typed variable to an integer in the print statement. The other was slightly more complex. It required converting floats into string using Java's String.format method, which allows floats to be converted into strings in standard or exponential notation. Thus, replacing floats with calls to this method with the float as an argument solved the problem.

One particularly subtle difference is that Go allows statements after break or continue statements while Java does not. The Java compiler outputs an error if the program contains unreachable statements. Luckily, even though Go allows unreachable statement to exist in the program, they are still unreachable and can therefore be disregarded. Therefore when printing a list of statements, the code generator checks to see if the current statement is a break or continue and if it is the rest of the statement list is not printed since it will be unreachable.

Java's lack of support for blank identifiers did not pose a huge challenge. We simply kept a global counter to keep track of how many blank identifiers had been encountered so far and generated the name 'blankId' followed by the value of the counter for any variable declared with the blank identifier. Blank functions and struct parameters were ignored since they could never be referenced. A similar fix allowed us to avoid any conflicts between Go variable names and Java keywords. All variables names were given the prefix '__goLite__' to ensure that no matter what name a variable was given in Go, it could not create a conflict.

Another Go particularity is that it allows multiple 'init' functions which are executed in lexical order before the execution of the main function. Java allows for such functionality by declaring a static block at the toplevel. This allowed us to mimic the behavior of Go's 'init' functions exactly.

GoLite's structs are fairly easily converted into Java classes. The only snag here is that Go allows element-wise struct equality to be verified using its equals operator. The equals operator for classes in Java verifies whether their memory locations are the same. To rectify this, a simple equals method needed to be added to each class which checked element-wise equality. A call to this method then needed to replace the equals operator whenever two classes were being compared. Similar calls to equals methods needed to replace to equals operator for other types not properly supported by the Java equals operator, such as String.equals for strings and Arrays.equals from the util package for arrays. Furthermore, Go allows strings to be compared lexicographically using the greater than and

less than operators. Therefore, strings additionally needed calls to String.compareTo where the result of that method is compared to zero using to appropriate operator to replace such comparisons.

Perhaps the biggest difference between GoLite and Java is the slice type. In order to mimic the behavior of the slice type in Java, we needed to create our own Slice class. This allowed us to control the growth rate of the object, and implement methods to get its length and capacity as well as to get the value at a given index and append. Thus, call to the built-in cap, len, and append functions in GoLite could be replaced with calls to their appropriate equivalent in this Slice class. One problem was that in order to avoid issues with Java's generic class functionality, values were stored in the Slice class as the Object supertype. This simply meant that values needed to be casted to the appropriate type when being indexed from a slice object.

4. Testing

Our group frequently tested our compiler using the provided subset of test go programs, this was beneficial since each test file focused on one syntax element. We frequently made adjustments to the test suite in order to add complexity to increase the robustness of our testing as well as increase the test suite's syntactical coverage. Various bash scripts such as `test-mode.sh` and `compare_generated.sh` were created to facilitate testing. The test-mode.sh script can be ran as: `bash test-mode.sh mode` or `ctest mode` if the script is sourced in the execution environment. This script helped by testing our subset of valid go programs all at once while providing success or failure messages per file tested. One of the reasons this script was created was to easily identify the programs which failed and why they failed. To accomplish this, the script outputs colorized success/failure messages where the only information provided for successfully tested files is the file name, but displays all program output in the case of a failure. Further, the last line of output from this script displays the names of all failed files which allowed us to easily identify which syntactical elements were not working for the mode tested. For testing code generation we created the script: `compare_generated.sh`. We found this to be very beneficial since the script displays the most recently tested go file along with the corresponding generated java code on standard output. Looking back, a simple improvement to this script would be to only display the files if the generated code does not compile or to display both files side by side by launching terminals.

4. Conclusion

Overall the experience of building this compiler was positive. Our biggest problem was organization. The problems in our code were often pretty simple to solve and might have been avoided if we had communicated better. If we had been organized enough to schedule meetings to work on this all at the same time we probably would have done much better. That said, this project taught us so much about not just compilers, but also time management, working as part of a team on a large software project, and the advantages and disadvantages of various programming languages. It was a unique experience absolutely worth going through even if the final result wasn't perfect. In terms of decisions we wish we could remake, the decision not to make an augmented AST during typechecking is the only one that comes to mind. The fact that we make a new symbol table for codegen is inefficient and meant we often needed to use functions from the typechecker that required information we weren't keeping track of, meaning we needed to use placeholder values for them instead. Having all of the type information stored in an AST node potentially alongside unique names for variables would have made the codegen phase much simpler. If we could do it over again, we would definitely make an augmented AST.

5. Contributions

In examining the GoLite specifications, we noticed that the language divided roughly evenly into three categories: declarations, statements, and expressions. With this in mind, we decided that if each of us was responsible for one of these sections throughout the duration of the project, we would each have roughly the same amount of work. It was then decided that Nathaniel would cover the declarations, Brendon would cover the expressions, and Shafin would cover the statements. Nathaniel also contributed the bulk of the reports as well as the bulk of the bug fixes for the final submission.

6. References

We worked alone.