Final Project Report

Group 7

Neil Pereira

Denali Relles

Yann Sartori

Introduction

GoLite is a (almost) subset of GoLang, where features that were removed include interfaces, packages, pointers, multivariate return functions, and any concurrency features, among other features. In effect, what we are left with is a c-like language, with c-syntax, type declarations, variable shadowing, and untagged types, to name a few. The following report documents our attempts and decisions at implementing a compiler for GoLite, where we first talk about our language decisions, the scanner, the parser, the AST and the structure of our specific node types, our weeder ; our symbol table including its structure, implementation, and our scoping rules, the type checker ; and finally, the code generator, where we discuss preliminaries functions needed for implementation, simple expression generation, how we record types in our codegen, complex expression generation, how we collect structs, our statement codegen, and our variable declaration codegen. We then close it out with our concluding remarks, as contributions each member made to the project, and the references we used in our project.

Language and Tool Choices

We decided to use Flex, GNU Bison, and C as our languages for the primary reason being we already had great familiarity with them. However, they also had the added benefit of speed and strong interoperability, with the first two tools being quite simple yet powerful, and C having the benefit of being versatile. The main drawback of using C was the lack of objects and algebraic data types.

We chose to target C mainly as it would offer us speed and low level memory manipulation which we thought would make it easier to implement our codegen.

Scanner

Most decisions regarding the scanner were straightforward. Reserved words and operators were each given their own token. Integer literals were represented by a single token, with hex and octal being converted to decimal integers. There were separate tokens for raw string literals and interpreted string literals. A decision of note concerns the handling of runes: we initially considered storing them as single characters, as they could fit into a single byte, however we decided that it was easier to store them as strings (char*) as we would need to reprint them during the pretty printing and codegen phases, which would be easier if they were stored as strings. For the optional semicolons, we followed the pattern on the course github page: a global variable recorded the last read token, and then a newline character determined whether or not we inserted a semicolon.

Parser

There were many issues we faced during the construction of the parser. The most trivial of them was enforcing precedence within expressions, which was done by using Bison's precedence directives in accordance with the GoLang specifications, along with using expression unrolling (with primary expressions mainly) to ensure they had higher precedence. Another issue we encountered was that blank identifiers couldn't be used in certain expressions and statements expecting a value such as the right hand side of assignments, function calls, switch statement expressions, etc. We decided to enforce correct usage of the blank identifier directly in the parser prior to the construction of the AST node as opposed to reflecting this requirement in the grammar which would increase its complexity.

Finally, we encountered an issue distinguishing between function call and type casts. However, at this stage of the pass, since both could be any type of identifier, it was impossible to distinguish the two, hence we treated both the same and resolved this at the symbol check phase, where we checked the genre of the identifier. Additionally, for our expression lists, we used left

recursion for LALR efficiency reasons. These lists were reversed when the AST nodes were created.

In the grammar, variable and type declarations were defined by their respective keywords followed by either declarations within parentheses or a list of identifiers and then possibly type information or value information. However, short declarations, due to parsing conflicts, were parsed as a list of expressions followed by the assignment token and then another list of expressions. We used weeding to ensure that those expressions were in fact identifiers, as well as to verify that the lists of expressions had the same length. Additionally, we wrote the parser in a way that permitted only one type to be declared at a time.

AST

Our AST had 7 main types of nodes: expression and statement nodes; variable, type, and function declaration nodes; and a type holder and top declaration node. We decided to have these nodes as we believed that it would strike a good balance between nodes being too specific and having too many structures to account for in our AST traversal, and too vague and losing a lot of information.

The root node has a special type, and it stores the package name, which is unique. It is always the root of the AST.

The program consists of top level declarations, forming the "backbone" of the AST. Each of these nodes points to either a function declaration node, a type declaration node, or a variable declaration node. Function nodes contain the expected information: name, argument information, return type, a pointer to the block of code within, a line number for error reporting, and symbol table information. Variable and type declarations are always expanded as much as possible. "Distributed" declarations with parentheses are broken apart into separate nodes, and declarations with more than one identifier separated by commas are also separated out. We decided that this would not affect performance, because while this requires that the type information is duplicated, expressions cannot be part of type information (at least in GoLite arrays must be created with integer literals) so this decision will have no side effects.

Structure of expression nodes

Our expression nodes contained the expressions's kind (which was an enum), then a union which contained relevant values for the more vague kinds of expression it could be-- binary (which had a left expression and right expression), unary (an expression), append (which had the base slice expression and the element to add), funcCall (which had the parameters as a list of expressions and the name of the function as an expression), and an expression for the body of built-in functions-- cap, len, etc. We also had the value for the literals and the identifier (which was a string). As remarked earlier, our runes were stored as strings. In addition, casts and function calls were represented by the same node; proper usage was checked at a later phase. A slightly novel field was a flag that recorded if this expression was parenthesised, which had one sole use: checking that the left hand side of short declarations had no parentheses. Otherwise, we stored line numbers for errors as well, as well as the symbol table information.

Structure of statement nodes

For the statement portion of the AST, the structure of the tree mostly followed the structure of the EBNF grammar as indicated on the Golang specification. The statement nodes consist of a line number, a field that tags each node with its kind, a union that has various structs to reflect the differences in various statement nodes and a pointer to the following statement as statements are represented as linked lists.

It was decided to weed out incorrect uses of blank identifiers during the parse phase as opposed to reflecting it in the grammar as it was easier to express and allowed us to give better error messages. If statements were designed to minimize the complexity of the tree; each if statement contained a pointer to an else statement (which was set to NULL to denote its absence). else if statements were not represented as their own node, instead they were represented as an if statement inside an else branch. Each of the three loop variants were represented by their own special statement node type. Switch case statements required the creation of a special struct type to represent switch case clauses. To be semantically equivalent with GoLite, we include special node types for increment and decrement statements as well as special node types for compound statements.

A new node type representing goto statements was introduced at the codegen phase. Though GoLite does not support goto statements, in the process of targeting the C language, appropriate continue statements inside loops were converted to goto statements. This new node variant greatly simplified the codegen process.

Weeder

The following language rules were enforced with a weeding pass. Continue statements could only occur inside a loop, break statements could occur inside a loop or a switch statement and return statements could only occur inside functions. The weeder function kept track of the scope of statements by using special arguments to determine whether a given statement was in a function, switch statement or a loop. Multiple default cases in switch case statements were also weeded at this stage.

Symbol Table

Decisions made regarding the symbol table structure

As 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 had 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-- we could shadow types with variables but we could not redeclare them. The reason we included functions in the symbol table was because they could 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 could 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 that would let us store both types.

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

An important decision here was that we immediately resolved types and stored them in the type node. This made our type resolution slightly quicker, but in addition, since we used strict type equalities (explained more later) in everything expecting a value, if we had "type A B", it wouldn't actually matter that A was defined in terms of B-- all that mattered was that type A got the base type of B. We reused context structs from the symbol table to keep track of struct fields as they allowed us to represent a namespace. Finally we stord symbol entries for functions because we needed both the ID and its type for each parameter for within the scope it introduced.

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 we need to be able to retrieve them later. A new scope is immediately placed on top of it to contain top level declarations. The following introduce new scope: (1) Function declarations : Functions declare new scope. Special handling of the body was required to ensure that the body of the function reused the parent scope. (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 and 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 recursively traversed the components of each expression, until we hit an expression with no more components (like literals). The only exception was 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 had to add a few flags to structs for one-time uses. 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 since they were functions. There was also another flag in the TTEntry struct for "comparable", which was set in the following fashion : only arrays, structs, and non-composites were comparable, and for arrays, an array was 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 wasn't too complicated: the function to create a TTEntry changed based on whether we were declaring a type or using an anonymous type, and if it was a type declaration, the name of the type we declared was passed down. There was also a flag to check if the recursive flag was in a slice.

Type Checker

In regards to expressions, we first declared helper functions that checked the genre of the underlying type (isNonComposite, isNumeric, isInteger, isBool, and typeEquality). For

comparisons, we 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 compare the pointer values. This was because two declared types were equal if and only if they *referred* to the same type. The only caveat to this was anonymous types (for instance, "var x struct {a int;}"), in which case we 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 type checking was relatively simple. The predicates for the type inference of unary and binary expressions corresponded directly to the helper functions (and accessing the isComparable field for comparison). Expressions that corresponded to indexing, length, capacity and append involved essentially checking the fields of our types to match the typing predicates, and returning the proper types in accordance to the typing rule. When we had literals, we returned the global base types (mentioned prior). And for identifiers, we returned the type in the polymorphic entry field. However, since types were not values, if the field was a type table entry, we actually threw an error. The reason for this was because the only place where a type could take part in an expression is a typecast, which we could handle in the function call expression case. This was a change from the previous implementation, as the previous solution led to sloppy code if we didn't want to allow illegal expressions in all cases.

The other kinds of expressions were slightly more complex, however. For field access, we verified the type of our access to be a struct. Then we looked within the context stored if there was a symbol entry that shared the ID of the assessor, in which case we returned the type of the entry. Since we did not set a parent for the context, we were able to reuse our lookup method from the symbol table phase. For function calls, we first checked that our base was an identifier--this disallowed for instance (1+1)(1+1). If that passed, we then typechecked our base. If our base had underlying type funcType, then we then traversed the STEntry of the function (c.f. symbol

table structure) and the arguments in the expression, and verified that those in the expression type checked to those of the type entry. If at any point we exhausted one list before the other, we knew we passed in an incorrect number of arguments. Otherwise, if our base was not of type funcType, then we verified that its polymorphic entry stored a type, and not a symbol. If so, we knew that we were attempting a type cast, so we then verified that we only had one argument, in which case we made sure that the conversion was valid as specified by the GoLite rules in which case we returned the new type.

We then stored the types in our expression nodes, as this made it easier and more efficient for the codegen phase (otherwise, we'd have to essentially re-typecheck each expression in our codegen). This wasn't done in the previous implementation, mainly because it wasn't needed (and we also forgot).

At the type checking phase of the compiler, in addition to type checking the body of functions, we checked that every function that returned a non void type ended 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)

Code Generator

Code Generation Preliminaries

The code generator was by far the most delicate portion of the project, and hence required many careful considerations. One of our first considerations was that the only defined types that actually mattered were struct types. This is because our type checker guaranteed type correctness, hence we were able to use base types and their compositions (arrays, slices) and their usages were legal according to GoLite, and by extension, C. This is where the fact that we propagated base types up into every symbol table entry came in very handy: There was no

“getBaseType” function, the base type was right there for us. This also made codegen for type declarations very easy: they could be skipped!

Our next important consideration was how we represented arrays and slices. For this, we used a sum type called “__GoLite_poly_entry” which could store the primitive types, a char pointer for strings, and a void pointer. The void pointer was quite important, as it allowed us to represent structs, as well as arbitrarily nested arrays and slices. Now an array was just represented as an array of the sum type. The actual type it stored could be retrieved whenever we encountered the expression, as in our type checker, we made all expressions store a reference to their type. As for slices, we did something similar except we declared a struct called__GoLite_slice which stored 3 fields: a size, a capacity, and a pointer to an array of __GoLite_poly_entry. This representation was chosen to mimic GoLite the most, along with its “quirks”. We then used a pointer to this struct whenever we wanted to use it for the option of storing a slice within a slice or an array under the void pointer field.

Another important consideration was name generation of variables and functions. There were three important criteria to consider: C does not have variable shadowing, we could declare GoLite variables that conflict with reserved words in C, and we could also have unlimited blank identifiers. In regards to the first two points, one really interesting property of the AST was that since we malloced a new PolymorphicEntry and STEntry/TTEntry for each variable, importantly whenever we entered new scopes, unique variable names could be generated using the pointer values of these fields. A solution we considered was using the hexadecimal address of the pointer as the variable name, as it would be legal, but that would be harder to debug. Instead, what we decided was to write a function that took in a polymorphic entry, and if it was an STEntry, we checked the identifier. If it was “main” (and the STEntry indicated it was a function), this was a special function, hence we returned the value “__GoLite_main”. This made declaring the C main function hassle free. If it was “init” (and the STEntry indicated it was a function), since we could declare any number of init functions, we returned __GoLite_init_i which corresponded to the ith init function. Since we generated code from top to bottom, i corresponded to the init ordering,

hence in our C main function, we could iterate over this value and write the functions in the proper order, staying consistent with GoLite init ordering.

If we encountered an unshadowed boolean (which can be checked by a flag set during the symbol check phase), then we wrote the ID directly. We imported stdbool.h in our generated file, so this worked. If we encountered a blank variable, we called a function tmpVarGen(), which returned __GoLite_temp_i, where i was a global current count of blank identifiers generated at that point. This guaranteed us a unique blank identifier name.

Finally, if it was a normal STEntry or TTEntry, we called a function where we passed the id and those respective pointers. As per the previous remark, we then stored variable names in a hash table where we hashed the id. Though we hashed on the ID, we stored the pointer value, due to the issue of declaring the same variable name in different scopes. Then we traversed the collision chain, where we kept track of the size of the bucket chain. If the pointer addresses matched, we returned __GoLite_decl_id_chainsize. As per the previous remark, since our pointer values captured the scope of the variable and we hashed based on the id, two variables with the same ID would always be in the same bucket, hence if they were in different scopes, they would have a different count, hence the variable names would be appropriate and readable. These prefixes prevented conflicts with C's reserved keywords.

These functions both created and/or retrieved the generated IDs for greater flexibility.

Easier Expression Code Generation

For the simpler cases: If our expression was an identifier, we called our idGen functions on the type of our expression and wrote the value. If it was an int, float, or rune literal, or interpreted string we retrieved the value from the AST node and wrote it. If it was a raw string, we called a special function which basically wrote the string character by character, and if we encountered a backslash, we wrote it twice to escape it, and if we encountered a quotation mark, we wrote an escaped-quotation mark . If we encountered a newline character, we wrote an escaped newline and escaped tab (\\n\\t), because that is what GoLite does. If we had any unary expressions, we wrote the corresponding C operations and recursed on the expression it stores, wrapping it in parentheses.

If we had a non-comparison binary expression, if it wasn't addition, we recursed on the left hand side of the expression, wrote the corresponding operation, and traversed on the right hand side, wrapping this all in parentheses. If it was addition, we did the same except if the type of one of the operands was a string. If it was a string, we generated a function call to concat which takes in as input two character arrays (the left and right hand side of the expression). This function created a new blank string, and then concatenated both strings to it using strcat from string.h. This was needed as C doesn't natively support a concatenation expression-- only concatenation statements.

If we had a comparison binary expression that wasn't equality or negated equality, then we called a function orderedBinaryGen, where if the operands weren't strings, we did the same as we did in the standard binary expression generation. If it was a string however, we generated a function call to strcmp (defined in string.h) and then did the comparison against 0 (so s1 <= s2 became strcmp(s1, s2) <= 0). This was necessary to comply with how C represents strings.

The first of the next challenging expression types were field selects on structs. We generated the base, wrote a pointer field access symbol ("->"), and then called our struct member id gen on the field id we wished to access. We didn't need to generate a cast, because we stored the actual type in the struct. The next was the built in length expression. For this, there were 3 cases: Case 1) We called it on an array. In this case, we inspected the type of the array, which stored the length of the array, and then wrote that value (so var a[10], len(a) wrote 10). Case 2) We called it on a slice. In this case, we generated the slice expression, then used the pointer field access on the field "size", because in our slice implementation, that stored the current number of elements. Case 3) We called it on a string. In this case, we generated a call to the string.h function strlen and then generated the string expression. This was cast to an int, because C returns an unsigned int, which would cause compiler warnings. The built in capacity expression was exactly the same, except we didn't have to deal with strings, and in the case of the slice, we accessed "capacity" instead of "size".

Concept of a Type Chain

Before we proceed to more complicated expressions, they all involve assignment or copying of some sort. The issue with this was in our implementation, due to how we represented certain types, we had to introduce functions that would allow us to have the same expected behaviour as GoLite. The first types were primitive types and slice types. C and GoLite both represent primitive types by value, hence we treated them identically in assignments and passing around. (This is actually a slight lie. C stores strings by reference, but since strings are immutable in GoLite, the fact that it is a reference type won't actually matter). For slices, in GoLite, they are references to arrays, so since we used pointers to represent slices in our generated code, the value was identical in C and in GoLite, and hence we treated them identically as well. However GoLite passes structs by value. Though C also stores structs by value, in our generated code, we actually used pointers, as mentioned earlier. Hence, we stored our structs by reference. Finally, with arrays, in GoLite, they too are stored by value. In C (and subsequently in our generated code), they are always stored by reference. Hence we took special care for this as well.

The "special care" alluded to simply involved making copies of our entries when passing them around. Additionally, we had to figure out a way to compare two polymorphic entries. However, there was a slight issue : Once we would have, say an entry representing an array, first of all, we wouldn't actually know that it is an array (all we would know would be that it is a pointer). Furthermore, we wouldn't even know what they store, or literally any type information. To mitigate this, we had to utilize the idea of a type chain.

In essence, in our codegen, we had a function that took in a type, and recorded the chaining of types (if we nested arrays). This type chain is used in knowing how to do copy and equality check, hence we recorded the appropriate information needed. If we had a non composite type or a slice type, we wrote the first and last letter of the type (so string becomes SG) and then we returned. The reason we returned is we didn't have to traverse the type chain anymore, either because there was nothing to traverse (non composite case), or we had a slice. In the case of slices, since type chains were used in equality and copying, we would never have a

slice in an equality check, and for copying, we would copy the slice by reference (because again, how they were stored was identical). For array types, we wrote “AY” and then the array size, and then recursed on the type it stored. The size was needed because we wouldn’t be able to retrieve it otherwise, and equality and copying would be done element wise, and the recursion was needed because this array would store variables of a type. For structs, we wrote “ST” and then the name of the struct in our C file. This is so that we could call the appropriate copy or equality method based on the name, as obviously each struct would not necessarily have the same fields. As an example, the type chain of [5][7]A, where A is a struct, would be AY5AY7STgeneratedID(A), and [5][]A would be AY5SE (since we stop at slices).

Secondly, as a helper method, we created a method called generateCopy, which took in a polymorphic entry and a type chain of that entry. It inspected the first type of the type chain. If it was an array, it recorded that fact and extracted the length. If it was a struct, it recorded that fact and extracted the struct name. If the type chain indicated the entry was a non-composite type or slice type, we returned the polymorphic entry, due to our previous discussion on how they were stored. If it was a struct type, we returned the result of the call of structCopy function mentioned later which took in a void pointer, which in this case, would be the polyVoid entry of our argument, as well as the name of the struct, which was retrieved from the type chain. With this name, structCopy was able to call the appropriate struct-copy method of that type. If it was an array type, we first retrieved the remaining portion of the type chain. Then we generated a call to a function, arrCopy, which took in a poly entry array, a string representing the type chain of the type the array stores, and the length of the array.

In arrCopy, we created a new array of the same size. Then for each entry of the new array, we called generateCopy on the corresponding entry in the array that needed to be copied, along with the type chain, hence creating these two mutually recursive functions.

Additionally, as a preliminary, we created helper methods that took in a variable of a type that polyEntry could store and returned the polyEntry representing it (createPolyInt took in an int, createPolyVoid took in an address etc). This allowed us to make conversions more easily.

With this knowledge, we are now ready to discuss quite complex expression cases

Complex Expression Cases

The first was indexing expressions. There were two cases: Case 1) We called it on an array. For this, in one of our prior generated functions, we had arrGet. This function took in a poly entry array of variable length, a position we wished to access, a length, and a line number. Then, if our position was not in bounds, we generated a run-time index-out-of-bounds error, citing the line number. Otherwise, we returned the value at that position. To actually generate the call now, we first generated a cast for the type our array stores, along with the function name. We then generated the array base expression, index expression, and then retrieved the length from the type, and the line number of this expression. Afterwards, since we would be returning a poly entry, we then called a function, generateUnionAccess, which extracted the proper union value based on the type the array stores. Case 2) We called it on a slice. This case was pretty much exactly the same, except we didn't take in a length since we stored that in the slice. Additionally, we returned the value stored at the array pointer that we stored in the slice. The generation followed the same pattern, minus again of course, the size.

One very interesting thing to note is this only worked for rvalues, not lvalues (for instance if we wanted to do a[0] = 5). For anything involving assignment, as will be mentioned later, we had another pair of functions, arrSet and sliceSet, which had the same function headers, except they took in additional arguments being the entry to add, along with the entry's type chain, and the return type was void. It then first called arrGet/sliceGet to determine if we were out of bounds, otherwise it accessed the memory location in the same way as their "get" parallel, and set it to the result of generateCopy on the element. This was needed because for instance, if you had var a [5][5]int; var b [5]int; a[0] = b, a[0] would store the actual value of b, (again, confer with the previous discussion).

With this in mind, the next expression type we had to handle was when we had a function call. As mentioned earlier in the typechecker, this could be both an actual function call or a typecast. If it was a typecast, we only had to worry about casts to a string. This is because C has automatic upcasting for primitive types. Hence in this case, we generated the argument. Now, if we were casting to a string (and our base wasn't a string), we generated a call to a function,

stringCast, which basically created a char array of size 2, put the argument in the first slot, and the null terminator in the second, and returned this string. We took in a char, which could accept ints as well in C. Once we generated the call, we generated the expression. If our base expression was a string, we wrote it (string to string has no effect). Now if we had a real function call, the thing to note is GoLite is call by value, as is C, but with their particular representation of values. Hence, for the call, we first retrieved the generated name of the function using the STEntry of the function as argument. Then for each argument, we first generated a cast for the type, a call to generateCopy, our expression represented as a polyEntry, the type chain of the type, then generated the appropriate union access. The cast and union access and wrapping were needed because generateCopy took in a poly entry. We could have done what generateCopy did directly and do away with all the conversions, but this allowed for code reuse.

Our next non-trivial expression type was append. We had a function we generate prior called append, which took in a slice pointer, an entry, and a type chain of the entry, and returned a slice pointer. The next decisions were all made to conform to the behaviour of slices in GoLite. We first created a new slice pointer. Now, if the slice to which we were appending had capacity 0, this meant it was uninitialized, so we set the size of the new slice to 1, the capacity 2, instantiated the underlying array (and its pointer) of the new slice, and then added a copy of the passed in element to the underlying array pointer. If the slice to which we were appending had a size less than capacity, then we do not need to resize. In this case, we updated the underlying array of the slice we passed in to include the copy of this new element, updated the size of the new slice, and then made the array to which it pointed the same as the slice we passed in. A thing to note is *we did not update the size of the original slice, yet updated its array*. This is because this was consistent with the GoLite behaviour-- though two different slices may share the same array reference, they might not necessarily have the same access throughout. Finally, if we were at capacity, we made the capacity of the new slice double that of the slice passed in, updated the size, and then created an entirely new array. For this array, we called the arrCopy function described earlier. Once we copied it, we added a copy of the new element to the new array. With this method, the append expression involved generating append function calls, generating the

code to get the slice, generating a type chain, generating the proper function call for the proper poly entry container, and the element to be added.

The final expression was equality. If it was a C primitive type, then we generated the left side, generated a "==" symbol, and generated the right hand side. If it was a C string, similar to the ordered case, we generated the strcmp function from string.h with the LHS and RHS as arguments, then generated a check of the return value to 0 ("== 0"). If it was a struct type, we generated a call to that specific struct's equality function (retrieved by outputting the generated ID of the type stored by one of the expressions) with the LHS and RHS as arguments. For arrays, we generated a call to a function called arrEquality. As inputs, we generated first the LHS, then the RHS, then a type chain of the type the array stored, and then the size of the array. The way this function worked was similar to arrCopy-- it first extracted the first type, inspected it, extracted the relevant information from the type chain, and set the proper comparison mode. Then we went element by element and did the appropriate equality check. For GoLite primitive types, we did it in the same fashion as before, except calling the appropriate union type (so for instance, ints would be arr1[i].intVal == arr2[i].intVal, strings strcmp(arr1[i].stringVal, arr2[i].stringVal) == 0), emulating a forAll function. If it was an array type, we checked the recursive call with each entry's polyval (casted to __GoLite_poly_entry *), the remaining type chain, and the array length extracted from the type chain. For structs, we checked the result of calling the function structEquality on the polyval of each entry of both arrays, passing in the name of the struct extracted from the typechain.

For the not equal case, we basically emulated the equality expression, except we negated the result.

Struct codegen

As structs were already a part of C, it was natural to represent GoLite structs as structs in C as well. There were two main issues that we encountered with regards to structs. (1) We could define structs within functions in GoLite, but not in C. (2) Anonymous structs were valid in GoLite, but not in C. We believed that the easiest solution to both problems was to do a first pass traversal of the AST and to collect the different struct variants after having type checked the

program. For the purposes of codegen, two struct variants were considered to be identical in type iff they had the same number of fields, identical names for corresponding fields and equal types for corresponding fields. It is worth noting that this approach doesn't differentiate between anonymous and named structs. With this notion of equality in mind, we decided that a data structure called a trie would best suit our needs. A trie is effectively a tree in which each node is permitted to have an arbitrary number of children. The internal nodes conceptually correspond to a field name - type tuple. The leaf nodes correspond to a label that serves as a unique identification number. Thus, every root-leaf path in such a trie would correspond to a struct variant. So, given a field name list and a type list, we could traverse the data structure and end up with the unique identifier for that struct type. We used this information to generate relevant struct definitions, type names and functions that could test for struct equality. We constructed the trie by traversing the AST and finding all struct occurrences. We could introduce a new struct variant in a GoLite program in the following ways. (1) A standard struct type definition, (2) An anonymous struct that was an argument to a function, (3) A variable declaration.
Since struct definitions could be nested, we had to traverse the contents of a struct looking at the types of the field names and decompose the structure of composite types (struct or slice types) to make sure that no type was missed. The main drawback of the approach in its current form is that it does not handle recursively defined types (attempts to encode the trie would result in an infinite loop)

Statement Codegen

The code generation of statements required subtle consideration, with most solutions being relatively straightforward. Block statements in Go were represented as block statements in C. Expression statements were generated without modification. Print and println statements were relatively straight forward. We only had to specially handle the cases corresponding to bools for which we used the ternary operator to print either "true" or "false", and runes in Go were casted to ints prior to printing. We generated if statements by enclosing them in a block to permit optional declarations. Return statements did not require any special consideration. We could not translate switch statements in GoLite to switch statements in C directly due to language

differences. We instead decided to translate the switch conditions into nested if/else statements, making the appropriate conversions. It is worth noting that we stored the value of the switch expression once prior to the if statement to ensure that each expression was computed only once. We then used the stored variable for the relevant comparisons. This if statement, any optional switch statement declarations and the declaration of the temporary variable were all enclosed inside an infinite loop which was terminated with a break statement so as to ensure that the break statements that belonged to the switch statement functioned correctly. We handled infinite loops and while loops without having to make any changes as these loop variants did not permit optional declarations. The three part for-loop required special attention. We firstly created a block statement; the initialization statements were then generated inside the block. We created a while loop whose expression condition was the expression supplied by the loop. In the event that no expression was supplied, we made the loop run forever. The post increment condition of the loop required some thought. As continue statements would have to directly move to the increment condition, the easiest solution was to just generate a label and substitute continue statements that belonged to the loop with goto statements. It is for this reason that the statement node definition was extended to include a goto node. We traversed the AST and converted relevant continue statements into goto statements. So, putting this all together, inside the while loop, we generated the modified body of the loop, followed by a label which was then followed by the post increment statement after which we end the loop. We did not have to change break and goto statements, and we appropriately transformed continue statements if necessary. Assignment statements were originally generated incorrectly but later fixed. The right hand side of assignments were generated and stored in respective temporary variables. These temporary variables were then assigned to corresponding variables on the left hand side. There was a subtlety involved with assignment and opAssignment (such as +=) statements for which the lvalue was the indexing of an array or slice. This was a consequence of our choice of representation which was previously described. These assignments were expressed through the use of specialized get and set functions for arrays and slices. Apart from this issue, the only other implementation detail we had to concern ourselves with was the plus operator as it could also represent string concatenation. We resolved these issues by using special concatenation functions

as well as careful pointer manipulation. The other opAssign statements did not require as much work and for the most part were equivalent to C's built in operators. We handled increment and decrement statements similarly, with slice/array indexing lvalues being treated differently than other lvalues.

Base codegen

Our biggest function, the one called in src/main.c, is called totalCodeGen. It set up the basic elements in our C code. The first thing to say, which wasn't a difficult design choice but it's worth mentioning, is that we did not map the "main" function in Go to the "main" function in C. Instead, we renamed the main function (as described above), and created a C "main" that did a few things: first, it initialized all the top level variable declarations, since those couldn't always happen the same way Go declarations do at the C toplevel. Second, it called all the GoLite init functions in order, which was easy since the number of init functions was stored in a global variable. Then, and only then, did it call the GoLite main function.

Variable declarations were handled basically like assignments. We probably could have skipped a step in real variable declarations, since we aren't allowing anything like

"var a,b = b,3"

But we standardized everything, because short declarations could have shenanigans like that. Declarations have the following steps: first, initialize any variables that are actually being declared. Save the names (I forgot that this part the first time around, which caused a problem with blank identifiers). Second, declare a bunch of temporary variables and assign them the correct values. Third, assign the temporary variables to the real variables. With toplevel declarations the only difference is that the first step happens in place, while the second and third steps happen in the main function before anything else.

When variables are initialized in Go, they need to be set to zero. This is annoying, but it can be solved with more coding than thinking. There's a recursive function that initializes ints, chars, and floats to zero, strings to the empty string, and slices with the correct structure, whatever they described above. For structs and arrays, We had to build up the correct name that had to be printed, including casts, and then call itself recursively.

> In passing I (Denali) found it amusing that, for arrays, I could choose between a for loop that printed out all the necessary lines of code or printing out a for loop. We chose the latter.

Conclusion

Denali:

I thought this was all a good time. This final really taught me how complex and finicky language design could be. The minilang thing was fairly straightforward, in terms of everything lining up with the basic concepts. But Go has all these layers. Short declarations? Blank identifiers? They just require all these extra passes. Or maybe there were easier ways to do it that we didn't think of. But I would never leave C. I'm a stick-in-the mud like that.

I think, if I had to do it again, I would try to give more input on the structure of the structures. I did of course have input, but in particular with the type checking and symbol table, I had some ideas that I didn't come up with until we were already halfway done and it was too late to change them. Maybe the solution could be to plan everything before writing the code. Then again, that sounds like a lot of extra work.

Neil:

I quite enjoyed the project. The only thing that I might have done differently is to change the language in which we wrote the compiler. While exploring the various language features of C was interesting, I found myself trying to reinvent the functional wheel. I don't think that I would use Haskell due to it being a bit too restrictive, but something like Ocaml would have definitely simplified many aspects of the project.

Yann:

I very much enjoyed this project. I felt as if the decisions we made generally held throughout later portions of the development, without having to write incredibly messy code (for instance, we never really had to drastically reformat our AST, nor inorganically force our current design and write really poor and hard to maintain code on top of a poor infrastructure). Of

course, we did have to change some things as we entered later phases (for instance, changing how we represented a list of short declarations), but these were generally easy to change, and didn't really affect the rest of our existing code.

If I had to change one thing, I feel like maybe generating Java code would be easier, at least in certain regards. For one, having polymorphism would've been much easier and cleaner for, for instance, copy and equality methods. In addition, we believe that our biggest slowdown was the fact that we had literally no memory management, and especially when we were doing a ton of copying, this added up quite quickly. However, if we had targeted Java, this memory hogging wouldn't have occured. That said though, there would be definitely difficulties in using Java that could easily be solved with C.

In short, very fun class! Thank you Alexander, Adrian, and Jason!

Contributions

Denali wrote the scanner, and then for each part, the code involved with declarations; Neil wrote, for each part, the code involved with statements; and Yann wrote, for each part, the code involved with expressions. In regards to structures (AST, symbol table, composite types in the codegen), this was all done on a collaborative basis. Additionally, if our code relied heavily on the code of someone else in the group, we would influence it in that way, as we saw fit.

References

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 for an edge case in semicolon parsing. We did not consult any other sources, except for past work, GoLang documentation, the GoLite tutorial slides, and the GitHub examples (for automatic semicolon insertion).