Final Project Report: GoLite Compiler

Erin Callow and Daniel Biggs

Comp 520

April 14, 2015

We're done. That much had to be said. Despite the handicaps of a smaller group size and relative inexperience as compared to other groups, we've completed the implementation of a compiler from GoLite to C++ to a standard of which both group members are genuinely proud. In this report, a general summary of work completed will be followed by a traversal of the programs where we will discuss design choices along with a detailed discussion of the purpose and implementation of all relevant 'helper' functions, particularly in the type-checking and code generation modules.

While C may often be irritating to work in, both of us had implemented our individual minilang assignments using flex and bison (as we'd learned it before SableCC in class), and so we decided to use C for the group project. Based on concerns in the code generation milestone (specifically handling aspects of a Go program that C does not accept [e.g., assignment of information that isn't a compile-time constant]), we changed our language of implementation to C++. As C is a proper subset of this language, 'changed our language of implementation' simply meant changing the extension of the file we generate and outputting the same 'C++' code we were generating previously.

The single biggest obstacle we encountered in this entire project is the way in which we developed the grammar: by hand, we decided on productions and hierarchy and then wrote the parser based on that grammar. We then spent the rest of the day fixing the emerging shift/reduce and reduce/reduce conflicts that naturally appeared in our grammar. It was only once we'd finished coding milestone 1 and began to test on programs we'd written in GoLite that we noticed flaws: certain cases of a production weren't accounted for, unfounded assumptions were made, and the base cases of 'lists' such as *identifierList* and *expList* required commas that GoLite did not require. We naturally fixed these problems as they emerged, but our workflow of writing a thousand lines of C and then debugging led to long-term difficulties: every change in the grammar now required corresponding changes in *tree* and *pretty* files towards the end of milestone 1; any changes made in advance of the submission of milestone 2 now required corresponding updates in *symbol* and *type*; towards the end of milestone 4, several final changes needed to be made so that the grammar would accept various valid programs: in the most important of these changes, the expression/addOp/mulOp/factor nature of the grammar had to be once again rewritten so that recursive booleans (i.e., a boolean expression where both children are themselves booleans) would be allowed by the parser. While these challenges may have helped us to better understand how a compiler works, the added difficulty could have been avoided.

Our compiler is run by invoking the (aptly-named) script *run* from the command line. Setting aside the use of flags to modify output, a GoLite program is passed as an argument to that script, which then runs the executable as created by the makefile with that same GoLite program as an argument. A call to *yyparse* is made, which in turn calls *yylex*, followed by an invocation of *weedProgram, mainTypeProgram, prettyProgram*, and *codeProgram*, which parse, scan, weed, type-check, pretty-print, and generate code, respectively. If the program is a valid GoLite program, the message 'Valid!' will be printed through *stderr*; if that program contains an error, the appropriate error message will be printed and the execution halted.

Scanner, Parser, Tree, and Weeder

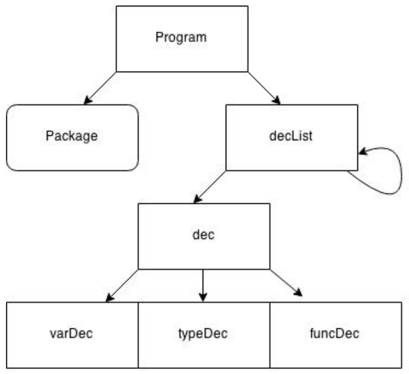
In our scanner, we define a variety of regular expressions to classify keywords, comments, numbers, identifiers, and the like. We first define the classes DIGIT and HEX to be any decimal and any hexadecimal digit, respectively. We then establish regular expressions for single and multi-line comments as well as regular expressions for keywords and operators (we're calling them regular expressions, but they're simply the exact thing we want to match, e.g, "<<" or "fallthrough"). In and among these regular expressions corresponding to keywords is one for a new line: because we're implementing the first case of

Go's semicolon insertion rule in GoLite, we need a way of knowing whether the last token seen before 'newline' is one that triggers semicolon insertion. The way we do this is by declaring and updating a global variable called *last* with the value of the last token seen; in the case where that last token is any of a laundry list of semicolon-insertion-triggering tokens, we return the token 'SEMICOLON' (while updating *last* to *newline* so that a second semicolon insertion won't be triggered should there be multiple newlines in a row). We also have a variety of regular expressions for various numbers and strings; in the case of an interpreted string, we iterate across its characters and replace any two-character representation of an escaped character (e.g., '\a') with the corresponding escape character (i.e., we replace something like '\' followed by 'a' with '\a') per the specification. Finally, in the case where the input matches absolutely no regular expression, we print an error message through stderr and halt execution; we accomplish this by having a program-final regular expression corresponding to 'anything' - if control reaches this expression, then the input isn't a part of the GoLite specification and must trigger that error.

Our grammar takes up the majority of the parser: we coded a union of all the levels of the grammar followed by token declarations of the leaves of our AST with their types and type declarations of the non-terminal nodes. Finally, we have precedence directives that correspond to Go's operator specification: multiply-level operations, followed by add-level operations, in turn followed by relational operators and then by *AND* and finally by *OR*. Finally, we have a *start* directive corresponding to *program*, the highest level in our grammar.

At the start of our grammar, *program* decrements into *package* and *decList*. *package* trivially decrements into the keyword 'package' followed by an identifier and a semicolon, while *decList* decrements into a list of *dec* (i.e., it has two productions: an empty base case and a *dec decList* case). *dec* is either a *varDec*, a *typeDec*, or a *funcDec*. Because GoLite supports 'single' and 'multiple' variable declarations, we have two productions for *varDec*: either the keyword 'var' followed by the grammatical category *var* or the keyword 'var' followed by the grammatical category *varList* surrounded by parentheses, where *varList* has a base case of *var* followed by a semicolon and a single recursive case of *var* followed by a semicolon and more *varList*: one way or another, *varDec* decrements to *var*. At this point, it bears mentioning how we handle types in our implementation of GoLite: because the user can define new types *ad nauseam*, any identifier could in theory be a type in this language, and so one might well try to make 'tIDENTIFIER' a production under *type* - unfortunately, this causes parser conflicts that we weren't able to resolve - the necessary steps we've had to take to allow user-defined types without including 'tIDENTIFIER' under *type* will be discussed at length, but, in the case of variables, suffice it to say we have five productions: a typeless declaration with initialization, a declaration with type and no initialization, a declaration with type and with initialization, a declaration with user-defined type and no initialization, and a declaration with user-defined type and with initialization.

As was the case with variables, a *typeDec* decrements either to *typeline* directly or to *typeline* indirectly through *typelineList*; *typeline* has two productions: an identifier and a *type* or an identifier and a second identifier (corresponding to a user-defined type) - this particular implementation has necessitated a copy of every production that uses *type* wherein we allow for a user-defined type in its place (i.e., a 'tIDENTIFIER'). It goes without saying that, were we to do this project over, we would make absolutely certain to write a correct and conflict-free grammar before doing anything else. Finally, *funcDec* has six different possible productions: three with parameters (i.e., that include *paramList* in between the parentheses) and three without; for both triplets, one of the three has no return type before the statement block (i.e., is of type void), while the second uses *type* and the third a user-defined type (i.e., a 'tIDENTIFIER').

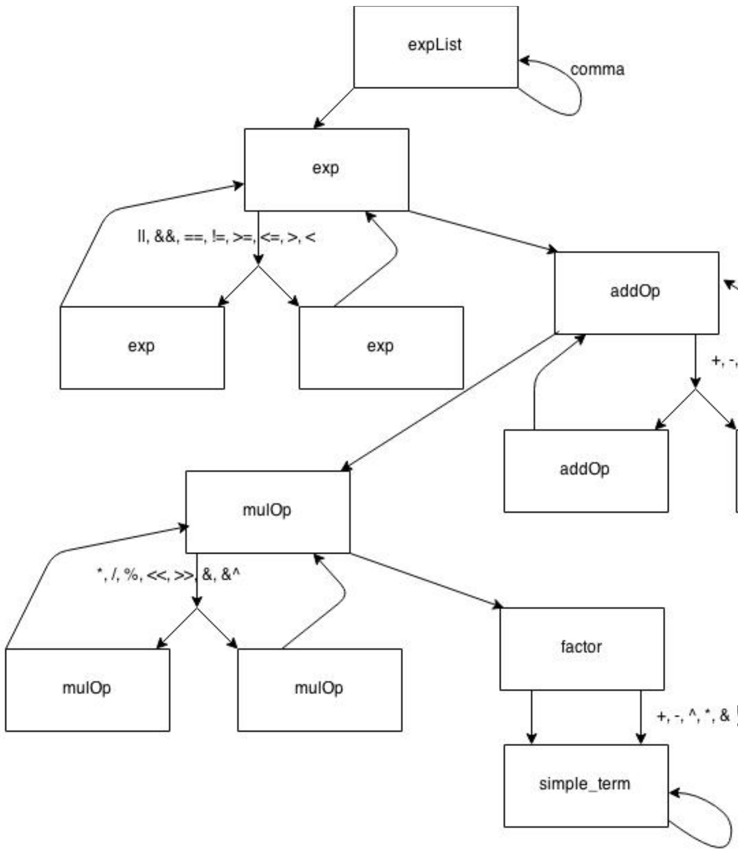


paramList, where referenced, is a list of parameters with a base case of a single *param*; *param* consists of either an *identifierList* and a *type* or an *identifierList* and a user-defined type (i.e., a 'tIDENTIFIER'); *identifierList* follows the same schema: a list of 'tIDENTIFIER' with a base case of a single identifier - note that a comma separates each 'tIDENTIFIER' in the recursive case. *block* is simply a *statementList* delimited by braces, and a *statementList* consists of zero or more *statement*s. A *statement* is either a *simpleStatement* or any out of *print, return, ifStmt, switchStmt, forLoop, varDec, typeDec*, a break/continue statement, or a *block statement list* (as distinct from a block, which contains a *statementList*). A *simpleStatement* is any out of *assignment, incDecStmt, stmtExp*, or a short variable declaration or empty statement. Since an empty statement consists of nothing more than a semicolon and since that semicolon is checked in *statement*, the corresponding *simpleStatement* production is simply nothing. As putting *identifierList* in place of *expList* in the fourth production corresponding to short variable declarations results in further conflicts, we use an *expList* and check that each *exp* is in fact a 'tIDENTIFIER' in the weeder (weedIsIDexp). An *assignment* is two *expList*s separated by an equals sign or two *exp* separated by a *selfOp*, which is any out of +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, and &^= (this language has a *lot* of operators). An *incDecStmt* is simply an *exp* followed by ++ or - -, while a *stmtExp* is simply an *exp* followed by a SEMICOLON (which, as in the case of an empty statement, is taken care of in *statement*).

print has four productions: 'PRINT'/'PRINTLN' followed by an *expList* delimited by parentheses and then a semicolon and 'PRINT'/'PRINTLN' followed by (empty) parentheses and a semicolon. *return* is comparatively simple, with the keyword 'RETURN' and a semicolon - in one production, there is an *exp* between the two, and, in the other, nothing. The productions corresponding to *break* and *continue* consist of just the relevant keyword followed by a semicolon, and *varDec* and *typeDec* have already been described. A 'block statement list' is simply a *statementList* surrounded by braces and with a semicolon following - while simplistic, its main purpose is to allow the user to create a new scope on demand.

Finally, there are three more complex statements: *forLoop*, *ifStmt,* and *switchStmt*. In the case of *forLoop,* we have four productions: the keyword 'FOR' followed by a *block* and a semicolon; that same keyword followed by a *condition* (i.e., an *exp* that type-checks as a boolean), a *block*, and then a semicolon; 'FOR', a simpleStatement (with semicolon), a condition (another semicolon), a second simpleStatement, and then a block and semicolon; and a 'FOR', a simpleStatement, two semicolons, another simpleStatement, and then a block and a semicolon. *ifStmt* is either the keyword 'IF', a condition, and a block and semicolon, or the keyword, a condition, a block, the keyword 'ELSE', a block and a semicolon, or an 'IF', a condition, a block, an 'ELSE' and then another *ifStmt*. Because there can be a *simpleStatement*, each of those three cases has two productions, one with and one without that *simpleStatement*, and so there are six productions for *ifStmt*. Finally, *switchStmt* has both, one, or none out of a *simpleStatement* (and semicolon) and *exp* in between the keyword 'SWITCH' and the brace-delimited *caseList*. *caseList* is a list of *case*s with an empty base case; a *case* consists of either the keyword 'CASE' with an *expList* (corresponding to the expressions on which that case matches), a colon, and a *statementList,* or simply the keyword *default* followed by a colon and a *statementList*.

An *expList* is a comma-separated list of *exp* (which necessarily has a base case of *exp* so that the comma never appears list-finally). An *exp* is either an *addOp* or two *exp* separated by ||, &&, ==, !=, >, <, >=, or <=; an *addOp* is either a *mulOp* or two *addOp* separated by +, -, |, or ^; a *mulOp* is either a *factor* or two *mulOp* separated by *, /, %, <<, >>, &, or &^; a *factor* is either a *simple_term* or a *simple_term* preceded by a +, -, !, ^, *, or &. Finally, a *simple_term* is either a 'tIDENTIFIER', a 'tDEC', a 'tFLOAT', a 'tOCT', a 'tHEX', a 'tRUNE' (corresponding to identifiers, decimal integers, floats, octal numbers, hexadecimal numbers, and chars), an INTERPRETEDSTRING or RAWSTRING (self-evident), a function call (a *simple_term* followed by parentheses surrounding an *expList*), an empty function call (*simple_term* followed by empty parentheses), an append statement (the keyword 'APPEND' followed by parentheses surrounding a 'tIDENTIFIER', a comma, and an *exp*), a cast (a *type* followed by an *exp* surrounded by parentheses [note this means a user-defined type alias will show up as a func call; we fix this in the type-checker]), a parenthesized expression, the pseudo-identifiers 'true' and false', or an array/slice access (i.e., a *simple_term* followed by a bracketed *exp*).



In GoTree.h, we define a node in the concrete syntax tree corresponding to each level of the grammatical hierarchy; in GoTree.c, we have a constructor for each kind of each node in the concrete syntax tree (i.e., a unique function for every production in our grammar). The parser uses these files to create the concrete syntax tree that it passes along to the pretty-printer, type-checker, and code generator. Our second-biggest challenge as a group was creating a *concrete* syntax tree as opposed to an *abstract* syntax tree: while we again managed to complete the project on time despite this manufactured difficulty, more effort was required as a result.

In the weeder, we check for a variety of possible context-sensitive errors that (necessarily) can't be weeded in a context-free grammar. In package, we check that the identifier designated as the package isn't the blank identifier. In weedvar, we then check that any initializations have lists of equal length on both sides, per the specification, just as we later do in weedassignment: in both cases this is accomplished by having weedexp return a 1 each time it's called; weedexpList will then return the sum (i.e., the length of the *expList* on which it is called), which can then be compared with that of the other side of the assignment. weedLH checks that an *exp* is an lvalue, while weedLHlist calls weedLH on each *exp* in an *expList* and weedRH checks that the right-hand side of an assignment is not the blank identifier ('_'). weedID and weedIDlist perform similar tasks to weedLH and weedLHlist. In the same vein, weedIsIDexp, as mentioned above, verifies that the left-hand side of a short variable declaration is in fact an identifier. weedStatement checks that break and continue don't appear in the wrong place - this is accomplished by passing a flag each time weedstatement is called - that flag will be set to 1 when inside an appropriate context and will remain 0 otherwise. weedSimpleStatement checks that any lists used in short variable declarations are of equal length, and weedforloop checks that the 'post' simple statement of a loop is not a short variable declaration. weedswitch checks that no more than one 'default' statement appears in a switch statement by using weedcaselist, which adds up the total number of 'defaults' it sees. Finally, weedfactor uses a flag to determine whether to allow pointers or blank identifiers (i.e., to determine whether it's looking at the left-hand side of an assignment/initialization or not).

Symbol Table and Type Checker

In GoTree.h, we created an enum called SymbolType with an enum representing every single type as well as one called noTYPE.

```
typedef enum SymbolType{intTYPE, floatTYPE, boolTYPE, runeTYPE, stringTYPE, sliceTYPE, arrayTYPE, structTYPE, noTYPE}
SymbolType;
```

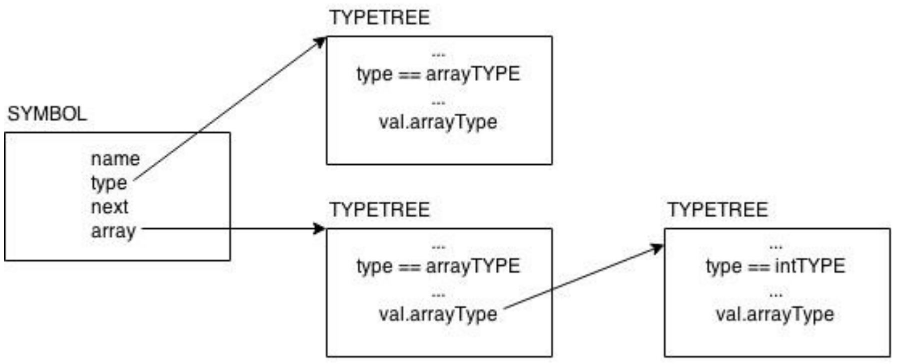
The structures for storing type information in the symbol table are as follows:

```
typedef struct SYMBOL {
    char *name;
    struct TYPETREE type;
    struct SYMBOL *next;
    struct TYPETREE *array;
} SYMBOL;

typedef struct TYPETREE{
    char *idType;
    SymbolType type;
    int typeDec;
    int funcDec;
    int lineno;
    union{
        struct TYPETREE *arrayType;
        struct TYPETREELIST *structType;
        struct TYPETREELIST *funcType;
    }val;
} TYPETREE;

typedef struct TYPETREELIST{
    int lineno;
                char *idName;
    union{
        struct{struct TYPETREE *tytree; struct TYPETREELIST *tytreelist;}typetreelist;
    } val;
} TYPETREELIST;
```

The SYMBOL structures are stored in the symbol table. The TYPETREE structures store the type information. “typeDec” is set to 1 if the symbol is a type declaration, and “funcDec” is set to 1 if the symbol is a function declaration. Depending on the type, the TYPETREELIST pointed to from the TYPETREE stores the name and associated TYPETREE of, for example, parameters of a function, or declarations in a struct. In hindsight, our structures are overcomplicated in certain ways, which made accessing types slightly more challenging. Suppose we have an array of arrays of ints. The first arrayTYPE value is stored in the TYPETREE pointed to by “type” in SYMBOL, and the subsequent type for array of ints is stored in the TYPETREE pointed to by “array” in SYMBOL. This was entirely unnecessary. We should have had everything accessible from the initial TYPETREE pointed to by “type” and not even had a TYPETREE pointer called “array” in the SYMBOL structure.



In GoType.c, we create an extern symbol table that will later be accessed in code generation. In the main, the AST is passed in along with an integer that is set to 1 if we call dumpsymtab in the command line. We initialize the symbol table using *initSymbolTable,* and true and false are pre-declared as booleans using *boolPutSymbol*. The type checker traverses down the tree inputting the types that correspond with identifiers into the symbol table whenever there is a variable declaration, type declaration, or function declaration.

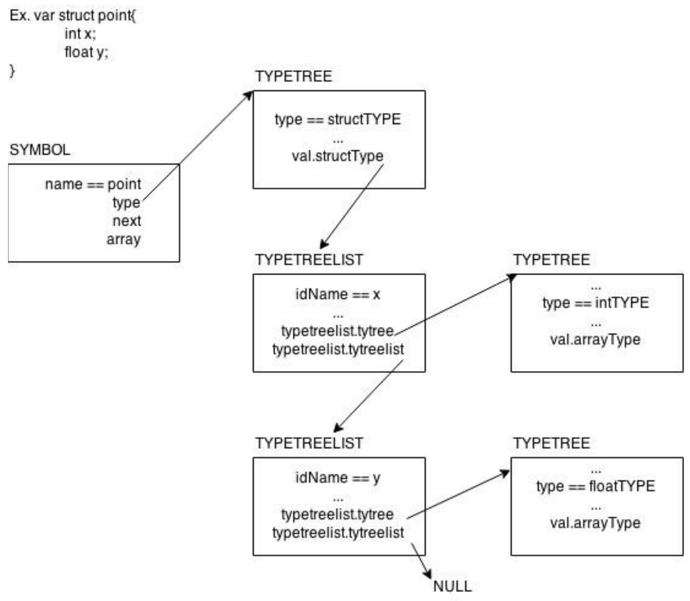
When a variable declaration is reached (symVAR), depending on whether the declaration is given a normal type or a user-defined type, as well as whether the declaration is set to equal one or more expressions, different helper functions are used to check if the variable being declared already exists. If not, it is inputted into the symbol table.

symTYPEIDLIST traverses through an identifier list and puts each identifier into the symbol table with the type information from the TYPETREE that was passed from a call to symTYPE. *symIDIDLIST* does this with a user-defined type. *symIDEXPLIST* puts each individual identifier from an identifier list into the symbol table with the type of the corresponding expression in the expression list; This is done by traversing each list one by one at the same time, and accessing the type of the expression by calling symEXP. *symIDTYPEEXPLIST* compares the expressions against the type in the TYPETREE passed from a call to symTYPE, and then the individual identifiers are put into the symbol table with that type. *symIDIDTYPEEXPLIST* does this with a user-defined type.

When a type declaration is reached (symTYPELINE), the type is either a normal type or a user-defined type. An error occurs if a type has already been defined in the scope or if it has already been defined as a variable. Otherwise it calls symTYPE and puts the type along with its corresponding name and TYPETREE into the symbol table. “typeDec” is set to 1.

When a function declaration is reached (symFUNCDEC), there are six different cases for declaration format that are checked. In all cases, an error is thrown if the function is already declared. In cases where a type is defined, symTYPE sets the type in the TYPETREE; otherwise the type is set to noTYPE. If the function has any parameters, they are type checked and their TYPETREEs are then linked to the function's TYPETREE through the TYPETREELIST structure, using *symPARAMTYPEIDLIST* and *symPARAMIDIDLIST*. These helper functions do not put the parameters into the symbol table. We want to store the type information for the parameters in the function declarations' TYPETREELIST but we want to store the function in the global scope of the symbol table. Therefore, we scope the table after putting the function declaration into the symbol table and then symTYPEIDLIST and symIDIDLIST are used to put the parameters into the symbol table in the proper scope. When the symbol for the function is inputted in the symbol table, the type information for the function return type along with the parameters are accessible from the single symbol. "funcDec" is set to 1. A new scope for the symbol table is created and the parameters are added to this new scope as individual symbols so that they can't be re-declared. The traversal then moves into the new block, which contains a list of statements. We pass the name of the function we are currently in down the tree, which is used to type check when we reach a return statement.

symTYPE is called whenever a type is read. It checks each case to determine what type is being passed and returns a TYPETREE with the corresponding SymbolType. For the case of arrays and slices, there is a recursive call on symTYPE, which sets "arrayType" to point to the next TYPETREE that stores the subsequent types. If there is a user-defined type, the helper function *idSet* is used to look up the symbol and then set the type in the TYPETREE. In the case of a struct, a new scope is created and all of the declarations are type checked. Their type information is stored in TYPETREEs that are linked together by a TYPETREELIST, which is pointed to by the "structType" pointer in the first TYPETREE that contains the base type structTYPE. This is done using the helper functions *symSTRUCTTYPEIDLIST* and *symSTRUCTIDIDLIST*, which are variations of *symTYPEIDLIST* and *symIDIDLIST* that store all of the TYPETREES corresponding to each declaration in the struct into a TYPETREELIST. They also put the declarations into the current scope of the symbol table to (type) check that something isn't declared twice.



Whenever an expression occurs, it is passed to symEXP, and a TYPETREE that stores its type information is made accessible at the expression level. The expression is type-checked at the expression level, if it has a comparison operator, using *comparable,* which returns 1 if the expressions on both sides of an operator are the same type, or comparable (e.g. comparing int to rune) and *ordered* which returns 1 if the expressions are of type int, float, rune or string. We traverse down through addOp, where it's checked that both sides of an addition, subtraction, etc. operator are numeric (int, float or rune), using the helper function *numericAddOp*. We then pass through mulOp, checking for numeric values on either side of multiplication, division, etc. operators, and finally pass into typeFactor where the type information of the expression is set in the factor TYPETREE and effectively passed all the way back up to the expression level. In setting primitive types, the TYPETREE "type" is set to the corresponding enum. In the case of identifiers, we check to see if the identifier has been declared, and then use *getSymbol* to access its type information. We then set the factor TYPETREE equal to the TYPETREE given by the SYMBOL when accessing the symbol table.

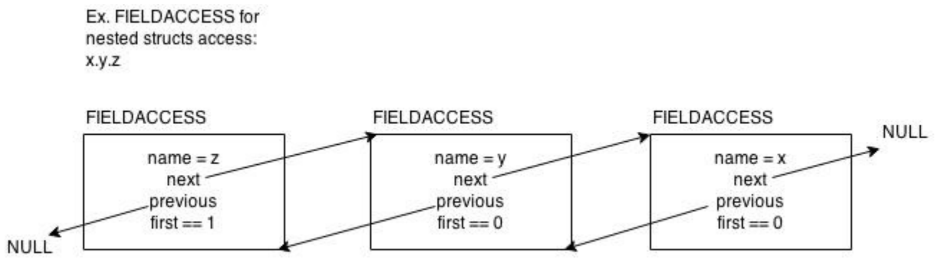
In factorType, we deal with casts in two different locations. Because of the way we structured our grammar, we realized that, when trying to cast to a user-defined type, it is read as a function call, and an error would appear saying the function did not exist. To handle this, we implemented a check where, if the name would be undefined as a function and if the typeDec flag of the corresponding TYPETREE is equal to 1, this factor must actually be a cast. We treat it as such by accessing the symbol of the user-defined type and setting the TYPETREE to the SYMBOL's type information. If cast is a normal type, we call symTYPE on the type passed and then set the TYPETREE at the factor level equal to the TYPETREE returned by symTYPE.

When append occurs, we first check to see if the identifier being appended exists, and then, using *getSymbol*, we access the corresponding symbol stored in the symbol table. symEXP type checks the expression that's trying to be appended, and then the type of that expression is compared to the primitive type of the array or slice , which is accessed using the helper function *toEnd*. *toEnd* takes in a TYPETREE linked list, traverses it to the very last TYPETREE, and returns the type.

When a function call occurs, it is either empty or has parameters. First, we check to see that the function has been defined. With parameters, we call *getFunc* to access the symbol that stores the type information. We then call *expToParams* to make sure that the expressions being passed into the call match the types expected by the function. When there are no parameters, we just get the symbol and store its information into the typeFactor TYPETREE.

When trying to access an array, we first type-check to make certain that the expression inside the brackets has SymbolType intTYPE. Then we check on the left hand side to see if the factor is an identifier. If so, we get the symbol corresponding to that identifier and set the factor's TYPETREE type by calling the helper function *toEnd*. If the factor is not an identifier, we traverse further down the tree.

Structs are accessed by looking at the factor on the left hand side and checking if the "kind" is an identifier. If it is, then we've reached the top level of the struct. We use *getSymbol* to get the type information of the identifier and then set the type of the factor TYPETREE equal to the type returned by a call to *structFieldAccess*. This takes the right hand side identifier of the struct access, searches through the TYPETREELIST corresponding to the left and side and if a match is found, the type is returned. To handle nested structs, we created a structure called FIELDACCESS. As we move from right to left across the struct access call, the identifiers are stored in a FIELDACCESS list, effectively saving the pathway necessary to follow through the structs to get to the stored primitive type. Because of the way we move down the tree, the list is stored backwards so we use a function *printAccess* to traverse through the list and set pointers to the previous structure. We use *getBase,* which returns the pointer to the FIELDACCESS structure corresponding to the left most identifier from a struct access (e.g., if the access is x.y.z = 9; getBase returns FIELDACCESS for x), to pass the last link in the list to *getStructType*. Using *structRecursion,* which returns the TYPETREELIST that corresponds to the identifier name found in the FIELDACCESS structure passed, *getStructType* traverses back through the FIELDACCESS list, following the path through the TYPETREEs of the nested structs to return the type of the element we're trying to access.



In simple statement, an assignment with one identifier and one expression is checked by comparing the types of both side's TYPETREEs. If we have a list of identifiers and expressions, this is type-checked with the helper function *symASSIGNEXPLIST*. Increment and decrement statements pass type-checking if expressions being incremented or decremented are integers, floats, or runes. A statement expression type checks if symEXP passes without error. A short variable declaration type checks using the helper function *symSHORTEXPLIST*, which traverses through lists on both sides of the declaration one by one, type checking the expressions and inputting the identifiers and corresponding types into the symbol table. *atLeastOne* is used to make sure that at least one identifier being declared has not been declared previously, per the specification.

A print statement type checks if the expression passed to symEXP passes without error. When a return statement is reached, if there is no return type, we check to make sure the return type of the function is noTYPE. We access the function by calling *getFunc* on the name of the function that we passed down from the initial declaration of the function. If there is a return type, the expression being returned is type checked and then we compare the type of the expression with the return type of the function.

When an if-statement occurs, a new scope is created immediately. If a condition and/or a simple statement are given, they are type-checked and a new block is created. In the case of an if/else (i.e., multiple new scopes), we initialize a second symbol table and store the original symbol table before it gets scoped for the opening of the if block. We scope the second symbol table before the else block as well as pass over the simple statement (if one was declared) to store it into the new scope of the symbol table and make it accessible inside the else block as well as the if block. We then pass the second symbol table into the else block.

When a for-loop occurs, a new scope is created immediately. If a condition and\or any simple statements are declared, these are type checked. Originally, we created the new scope directly before the new block. This caused no issues in accessing the simple statement type information inside the block, but we realized that it was then also accessible to the scope before the block. This would cause an issue in the case where a simple statement declared x as an int, and there was a variable declaration for x past the for-loop. The type checker would think we were trying to declare a variable twice. To correct this, we moved the call to *scopeSymbolTable* in front of the simple statements and condition.

When a switch statement occurs, a new scope is created immediately. The simple statements (if any) are type-checked, and the expression (if there is one) is type-checked. The type of the expression is passed down the tree and compared to each case using the helper function *caseExpCheck*. If no expression is declared, noTYPE is passed down the tree and the cases aren't compared to anything. If a statement list occurs inside a statement list (i.e., a block), a new scope is created.

GoSymbol.c contains all the helper functions used to initialize, create a new scope for, input symbols into, or access a symbol table**.** There are six different functions that deal with inputting new symbols into the symbol table. They account for each case where the symbol has a primitive type or a user-defined type, and is a variable declaration, type declaration, or function declaration. *getSymbol* returns a symbol if it exists at any scope in the symbol table. *getFunc* returns a symbol if the symbol is a function declaration (i.e., "funcDec" == 1) and will only check the global scope of the symbol table. *symbolDefined* and *typeAlreadyInScope* return 1 if a symbol or type, respectively, are defined in the current scope. We use these functions to check whether a variable or type can be re-declared. *symbolAccess* returns 1 if a symbol has been defined in any scope at all and *typeDefined* and *funcDefined* check specifically for type and function declaration symbols. *getCurrScope* returns the symbol table of the current scope only. *printSymbolTable* prints the symbol table passed to it with the help of the functions *arrayPrint* and *structPrint*. *printCurrScope* only prints the current scope of the symbol table. *getPrimitive* returns the primitive SymbolType of a variable with a user-defined type; it takes in the name of the user-defined type, and searches through the symbol table for the type declaration until it finds the original type declaration with a primitive type. *getTreePrim* finds the symbol corresponding to a name and returns the TYPETREE pointer "array". *getStructTreePrim* finds the symbol corresponding to a name and returns the TYPETREE pointer called type which stores all the type information of a symbol.

If at any point an error condition is met, the appropriate message is printed out and execution is halted immediately. Otherwise, the program type-checks and the pretty-print begins.

As mentioned previously, in order to accommodate the fact that user-defined alias types are not included under *type*, we created 'TYPETREE' in GoTree.h, which would contain either a *type* or a string corresponding to a user-defined type - in the case of a 'recursive' type, such as a struct, slice, or array we also implemented a union containing structs as we did for every other node. We include this TYPETREE in each level of our expression hierarchy so that we can easily find out the type of any expression, addOp, mulOp, factor, or simple_term when printing out code. It's in prettytypepptype in GoPretty.c where we first make use of this feature to accommodate the -pptype flag: given a TYPETREE when it's called from prettytypepp, prettytypepptype prints out the type associated with that expression - in the case of 'recursive' types, prettytypepptype is called on the TYPETREE contained within the union within TYPETREE. Any pretty function that would call prettytypepp (or prettytypelistpp, which iterates over an *expList* and calls prettytypepp on each *exp*) has a flag passed as a parameter, as we do in many other functions in many of the files - if the call to prettytypepp is in fact made, then comment syntax (i.e., /*… */) is printed out. The fact that a valid program will run no matter how many times it's been 'pptype'd is something of which we're especially proud.

Code Generation

In the code generation files, we traverse the tree in a similar fashion to *pretty*, and quite a bit of the code generated is the same in both cases. Unfortunately, there are several ways in which

C++ differs from Go, particularly in typing and allowing multiple instances of some statements on one line where C++ emphatically does not. As such, a variety of helper functions was needed to gather information and handle cases outside the scope of 'normal' tree-traversal functions.

Code generation starts at codeprogram, where <stdio.h> and "golite.h" are included, the latter being a file written in C++ to implement resizable arrays for slices. If a type declaration in codetypeline is a struct, a special helper function called codestruct is called: first the general structure of a struct is printed out, and then the call is made, during the scope of which codestructlist is called to repeatedly print out declarations via codeparamstruct, which in turn calls codeparamlisttype and codeparamlistid for 'normal' and user-defined alias types, respectively. In codeparamlisttype, anonymous structs in declarations are handled by giving them the temporary name "name" to define them as a type and then defining the variable name in terms of that "name" type. Back in codetypeline, if the type declaration is simply a declaration of a (non-struct) type in terms of another type, "typedef " is printed out followed by the old type and new type. For variable declarations, structs are handled as just described, while calls to codetype are made to find the type of a variable and calls to dummyassignment are made to take care of initializations. codetype switches over a type passed to it and prints out the appropriate type in the case of a primitive type; in the case of arrays and slices, a call is made to another helper function: array. array takes in a type, a name, and a flag 'init' that determines whether to initialize as well as declare. array uses a variety of functions to create and update Array_int, Array_float, Array_char, and Array_string structures defined in "golite.h" - this way, resizable arrays can be used throughout the generated C++ code.

dummyassignment takes in an *identifierList* and an *expList* as well as a flag; it prints out assignments of each element of the former to the latter and prints out the type beforehand if that flag was equal to 1. codetype takes in the name of the variable in codevar (for handling arrays and slices), but it was necessary during debugging to make sure this variable name wasn't used for primitives, and so the string "error message" is passed to codetype for variables of primitive types - if the name of a primitive would be printed out by mistake, the error message is printed instead (note that this will simply not happen - we're just explaining why the error message is there from debugging). Finally, in terms of declarations, we have codefuncdec, which, as the name would suggest, handles code generation for functions: we print out the return type or 'void' (note that, in the case where a function named 'main' would be of type void, we use type 'int' instead), the name of the function, the parenthesized list of parameters (with appropriate typing information), and then the body of the function via a call to codeblock; the parameters are printed through a call to codeparamlist and then to codeparam, which handles primitive, array/slice, struct, and user-defined types.

codeblock and codestatementlist perform as expected, and the continue, break, and 'sblock' cases of codestatement are straightforward, while variable and type declarations are handled by codevardec and codetypedec. return simply prints out the return statement (or omits it in the case of a void function), while print is considerably more complicated: in the case of printing an empty list of expressions, "printf()" and "printf(\"\n\")" are printed, while a call to the helper function codeprintcontent is made for a non-trivial print and println command, with a flag of 0 and 1, respectively (the flag indicating whether to insert a newline into the printed printf). codeprintcontent simply switches over the type of the expression and puts the corresponding escape sequence into the printf printed out (e.g., %d, %f).

Back in codestatement, for loops and if statements are relatively similar to their c counterparts - brackets must obviously be placed where appropriate, but little else changes with the exception of the (perennially irritating) optional *simpleStatement*. Where these are present in the *ifStmt*, it is printed out on its own line before the 'if' keyword to conform with C++ syntax. *switchStmt* looks nothing like its GoLite counterpart when generated: GoLite allows switching over virtually anything, while C++ does not have this freedom: where the optional *simpleStatement* exists, it is taken care of as in *ifStmt*, but codecaselist is entirely unlike its *pretty* counterpart: it eventually decrements to codecaseif, which compares the expression being switched over with the expression in that case - where there are multiple expressions corresponding to a given case, these equality comparisons are linked together with an 'or' operator; in the case where there is no expression to switch over, we call a similar set of functions with identical names save the addition of 'empty' at the end, which perform essentially the same tasks save one exception: they don't perform an equality check between the expression being switched over and the expressions on each line - rather, they treat them as conditions in their own right (which, for the program to type-check, must be the case).

Finally, in terms of *statements*, there remain *simpleStatement*s: the case of empty statement and statement expressions are trivial (with nothing being printed out and a simple call to codeexp, respectively). For assignment, self-ops are handled by printing out each side of the expression with the operator in between, except where that operator is &^=, which gave us some trouble in implementation: we have a special case in both code assignment and codeselfop whereby the assignment is rewritten in simpler C++ that compiles properly. Both non-self-op assignments and short variable declarations make use of the helper function dummyassignmentexp: as in dummyassignment, when the flag (which indicates declaration as well as assignment) is equal to 1, codetypetree is called to print out the correct type of the left-hand-side expression, while both expressions are then printed out with an equals sign between them. As in other such helper functions, this function iterates over the *expList*s such that each *exp* is handled as just described and each pair of *exp* is printed out on its own line.

Finally, in terms of *simpleStatements*, codeincdec prints out the relevant expression followed by either "++" or "- -", as appropriate. codeexp, codeaddOp, codemulOp, and codefactor, are much like their counterparts in *pretty* with a few small exceptions: where an operator differs between GoLite and C++, obviously the C++ version is printed out; in the case of addition, if both expressions are strings, a helper function 'concatenate' is called from "golite.h", which smoothly concatenates the strings; array access is modified slightly, as the array is a field of the Array_int, Array_float, etc. struct; since the cast case of code factor requires a *type* (and a user-defined type would be a string), it only handles casts that don't use aliased types - when the 'name' of a function is in fact a user-defined type and not a function, that function call is treated as a cast - this solution makes the top three list for 'things we're proud of in this project', particularly since what would otherwise have been an error was noticed the day before we tagged our submission on git; finally (finally), as append has no counterpart in C++, its implementation is entirely unique: depending on the type of the 'slice' being appended to, the appropriate function out of insertArray_int, insertArray_float, etc. is called with the expression to be appended - in "golite.h", the next spot of the array is filled or the array is reallocated with a larger size if necessary.

In summary, we're done (we just really like being able to say that). Our compiler scans, parses, etc., and properly generates code in nearly all test cases made available to us. Throughout this course and in particular throughout this project, we've learned exactly what goes into building a working compiler and the importance of generating optimizable target code; we've also developed a newfound appreciation for the features that our favourite languages offer and a sense of respect for the people responsible for writing the compilers we use. In addition to what are presumably the fundamental objectives of the course, we learned several important practices to take into consideration the next time we develop any large software projects, the most important of which is undoubtedly 'debug periodically' - while there may be a sense of satisfaction associated with overcoming manufactured difficulties that didn't need to be there in the first place, we now know enough to plan out our code, debug periodically, and, in the case of writing modular code, complete each module to specifications before moving on to the next.