GoLite Compiler Design Report

Angel Ortiz Regules
260660945

Joshua Inscoe
260659992

Zakaria Essadaoui
260776919

Table Of Contents

Introduction

Go is a statically typed, compiled programming language developed by Google which provides a syntax close to C while adding memory safety and garbage collection along with other features. In the context of our compiler class, our main task was to develop a full compiler for a language representing a considerable subset of Go, nick-named GoLite, which was designed by the course instructors. This project gave us hands-on, practical experience in compiler design, and allowed us to see what goes into the creation of a programming language.

Our project was separated into three milestones (originally four). Each milestone represented a significant portion of the compiler, which itself was divided up amongst the members of our team. The first milestone consisted of the compiler front end, which has the task of parsing the GoLite code into an internal tree representation. This was divided into the following components: The Scanner, Parser, Weeder, Pretty-printer, and the AST. The second milestone consisted of the symbol table and type-checker. Finally, the last milestone focused solely on code generation, which was the last piece of the puzzle.

In this report we will showcase the reasoning behind our choices for source and target languages, as well as the tools, of which we made use. We will then discuss in detail all components of our compiler, summarized above, focusing, in particular, on the major design decisions made.

Language and Tool Choices

Implementation Language

The choice of programming language used for the implementation of a compiler is inevitably one of the most influential ones since it is highly coupled with the design of the abstract syntax tree (AST). More specifically, the way in which the AST is designed and implemented is dependent on the abstractions and features that the programming language provides.

For the GoLite compiler implementation we considered two programming languages: C and C++. We decided to focus on these two languages given their ubiquity in the world of compiler design and their high performance capabilities. Furthermore, C and C++ are fairly similar, both offering manual memory management and access to other low level features. The decision of which of these two languages to use was largely motivated by the fact that an AST is nothing more than a data structure representing a program in the source language. When it comes to data structures and abstractions in general, C++ provides a much richer feature set including classes, inheritance and polymorphism. By contrast, C is significantly simpler, offering only structs, enums and unions to create arbitrary data structures. Furthermore, C++ includes a large range of common data structure implementations such as vectors and hashmaps, whereas C requires these to be manually implemented.

If we considered only the feature set on each language, C++ would have been the obvious choice to make, given that the performance differences between C and C++ are not large enough to be significant in this project. However, the increased feature set of C++ also meant an increased design space for the AST. Taking into account that the time frame for this project is rather constrained, we decided that we might not be able to design the AST correctly using C++. By contrast, C offered a simpler and somewhat safer AST implementation alternative, which is why we picked it. In conclusion, we acknowledged that C++ had the potential to allow for a better AST but we did not think that we could manage to create it within the project time frame.

Additional Tools

The main two additional tools that we used in the design of GoLite were Flex and GNU Bison (Bison). Flex was used to generate a lexer (or scanner) for the language, while Bison was used to create the parser. While these tools are convenient for generating the first two stages of the compiler, the lexer and parser they generate are not as efficient as a hand-written equivalent. We considered using a hand-written lexer and parser for performance reasons, however, given the time constraints we decided (for the time being) to stick with the lexer and parser automatically-generated by Flex/Bison. This will allow us to focus first on the other stages of the compiler and then later, if time permits, optimize the performance of our implementation (e.g. by rewriting the lexer and parser by hand).

Scanner

Our scanner handles the following use-cases:

- Matching of reserved keywords. These keywords were defined in the language specification for Milestone 1. It is important to note that base types as well as base constants (i.e. true, false) are not considered keywords since GoLite allows the user to overwrite these at any given time.
- Matching floating-point literals according to the format specified in the GoLite specification.
- Matching integers literals according to the format specified in the GoLite specification. Two extra functions have been added in order to convert strings from octal or hexadecimal format to integers.
- Matching strings and raw strings as defined in the GoLite specification. In addition to that, we convert raw strings to regular strings by escaping backslashes. We made this decision in order to facilitate later phases of the compiler since the two constructs have the same behavior.

- Matching runes and their escape characters (defined in the M1 specification).
- Matching single-line and block comments.
- Matching identifiers following the GoLite standard.
- Insertion of semicolons after new lines according to the use cases depicted in the Go specification.

Any GoLite code that does not comply with the above rules will cause the scanner to emit an error.

Parser

To create the parser stage of our GoLite compiler, we decided to use Bison as mentioned previously. As a consequence, we were required to define a grammar to describe the syntax of GoLite that Bison could use to generate parser code. We decided to focus on the following three objectives while designing this grammar:

- Generate a parser that is able to detect as many syntax errors as possible in order to decrease the load on subsequent phases of the compiler, such as the weeder, which are manually coded and, hence, more prone to bugs.
- Avoid any conflicts that a lookahead LR (LALR) parser cannot resolve, so that a generalized LR (GLR) parser with slower performance is not needed.
- Keep the design compact and easily maintainable given the large set of syntax rules included in the GoLite language and the collaborative nature of this project.

In the remaining part of this section we provide a brief description of the design approaches we used to make our grammar conform to these goals as much as possible.

We started by defining individual grammar units to tackle the different constructs that the language supports. To do this we exploited the fact that the GoLite specification already has separate definitions for each of these constructs. An example of this is the function declaration grammar, which is among the most complex constructs of the language syntax since it has to deal with different argument types as well as return types. In our design we created a single `func_dcl` grammar to compartmentalize the details of this grammar. Handling the details of complex grammars like this brings us to our next design approach, which consisted of the creation of small abstract helper grammars. These helper grammars are not explicitly mentioned in the GoLite syntax specification, but together they help create complex grammar definitions such as `func_dcl`. An example of this is the `identifier_list` grammar, which simply defines a comma-separated list of names that are valid identifiers according to the language rules. The `identifier_list` is not a real construct of the GoLite language on its own, but is a recurrent part of many other grammars such as `func_dcl` (function declaration) and `var_dcl` (variable declaration), which are defined in the GoLite specification. Finally, we used high level grammars to associate multiple constructs according to different contexts. High level grammars are like helper grammars in that they are often not explicitly mentioned in the GoLite specification, but rather than participating in the creation of larger grammars, they are instead made up of the constructs of the language. One example of this is the `simple_stmt` grammar which defines a subset of statement constructs that can be used in the context of a simple statement. One important remark about this organization is that helper grammars may be constructed from other helper grammars.

AST

Our AST revolves around the following main nodes:

- Declarations
- Expressions
- Statements
- Types

Declarations

This can be one of 4 type:

- Package declaration: which basically only contains the package name.
- Variable declarations: these can either for function or normal variables and we can differentiate by non array type and array types. In order to deal with the shift reduce conflict that arises from trying to define array type in the grammar, we add another step in these node's construction where we check if the end of the expression list is node of type `expression[expression]` and then we take some actions in order to restructure the node in our desired format.
- Type declarations: these can include identifier types or function types.
- Function declarations: only contain the list of input declarations, the return type and the core.
- Declarations sequences: implements a linked list of declarations.

Expressions

In our tree expressions are one of the following types:

- Identifiers.
- Literals.
- Unary expressions.
- Binary expressions.
- Function calls.
- Indexing expression (i.e. `array[index]`).
- Field selector expressions (i.e. `some_struct.element`).
- Sequence expression: This type is used to represent expression lists.

- Parentheses expressions: This type is used to refer to when an expression is put inside parentheses. We decide to add this node in order to give us a way during the weeding to detect declaration such as : `var (a) int = 2;`.

Statements

Statements can be one of the following:

- Expression statements.
- Block statements
- Assignment statements
- Operator assignment statements: this include statements such as `a+=1;`
- Declaration statements;
- Short declaration statements.
- Increment/Decrement statements
- Return statements.
- If statements.
- Switch statements.
- Switch case statements : basically stores all the statements and the conditions that come with the “case” statement.
- For statements.
- Statement sequences: implements a linked list of statements.

Type

We have used this node to hold the type of a declaration, whether it be for simple types or composite types such as arrays and structs.

Weeder

Since it was not possible to implement all the language functionality in the parsing phase a weeding was required. This pass handles the following checks:

- The balance between the left hand side (lhs) and the right hand side (rhs) of assignment.
- A valid identifier list in the lhs of an assignment. This check looks for incorrect identifiers in the lhs since the parser returns as an expression list. An example of these invalid expressions are literals, unary and binary expressions and function calls.
- A valid identifier in the lhs of a declaration. This check ensures that the lhs of a declaration (short and normal) is valid. Unlike in the previous check, field selector expressions and indexing expressions cannot be allowed.
- Ensuring that the lhs of an increment, decrement or an operator assignment (e.g. +=) statement is a valid identifier.

- Valid expressions on the rhs. This check only looks for whether the blank identifier is used on the rhs of an assignment.
- Making sure that the blank identifier is not used in any condition expression or function parameter.
- Break and continue are in the right scope. The weeding phase checks that these two keywords are used only inside switch or loop statements. This is done through global variables that enable and disable the use of these keywords.
- Proper expression statements. This check makes sure that expression statements can only be composed of function calls.
- One default case in switch statements. A count of the number of default cases is performed and an error is thrown if the count is greater than 1.
- No short declarations in post condition of 'for loop' statements. This check ensures that the post condition cannot be a short declaration.
- When dealing with function declarations of non-null types, the weeder ensures that the function terminates properly (following all the cases specified in the Go specifications and that are applicable to GoLite regarding Terminating Statements). However, we do not check that the type of the 'return' expression is compatible with function type in this weeding phase as this is done in the type checking phase

Symbol Table

Symbols

In our implementation, the symbols are represented by the following:

- The symbol name.
- The symbol type: Defines the type of the symbol, we will discuss it in the following paragraph.
- A linked list list of symbol types used to store information about the arguments types for functions.
- The symbol kind: can either be a function, variable, type or constant.
- A pointer to the next symbol, this is used to store symbols in the hashmap of the symbol table.

In order to implement our symbol types, we decided not to use the 'type' node that was used in the AST since the one linked to the symbol will contain more information and sometimes has a different structure. For example, the AST type stores structs as declarations but in the symbol table we will store them as a linked list of types. The 'symboltype' struct contains the following elements:

- A name: refers to the name of type. For variables, this refers to the name of it's parent type, but for types, this refers to its own name.

- A kind: this can be one of the base types (int, float64, etc), inferred, void, struct or defined.
- A pointer to the symbol of the parent type, this points to the parent when the type is a defined type.
- A pointer to the linked list of symbols (called ‘structelements’) that represents the elements of structs. These symbols are not pushed to the symbol table.
- Array information: Basically defines whether a type is an array and stores its dimensions. The dimensions are stored as a linked list of integers where an index value of ‘-1’ represents slices.

Whenever a new variable or type is created, if it is based on a defined type, then we link it through the parent pointer and we set the kind of the type to ‘defined’. Otherwise, if the type is a struct, then we process the struct elements and set the kind of the type to ‘struct’. Arrays are handled after this step and they are just copies of the array information that come from the types stored in the symbol table.

It is worth noting that we are aware that this design is suboptimal as it causes data to be copied unnecessarily. However, given the time frame allocated for this project we decided to focus instead on improving functional aspects of the compiler, such as correctness and performance. Alternatively, we could have changed the way we deal with arrays and slices so that each dimension has its own type, and the same information could then be conveyed through enums instead of linked lists of indices. Furthermore, having a function type would also have been a good idea since, in Go, variables can be functions. At this point function types are determined via their symbol kind.

Scoping Rules

New scopes can be pushed and popped from the scope stack using the functions ‘_scope_sym_table()’ and ‘_unscope_sym_table()’. In our implementation we follow the following scoping rules:

- In the base scope (the initial scope), we push the base types (int, string....) and the boolean constants (true and false). Then we create a new scope where all our top level declarations are stored. This is done this way since any of the base types and constants can be shadowed and this cannot happen unless the base types and constants reside in a different scope than the rest of the program.
- When a function declaration is encountered, the function is added to the current scope and a new scope is created. In this new scope, the function arguments are first added to the scope before processing the function’s statement. When no more statements are left in the function, this new scope is popped off the stack.
- When an if statement is encountered, a new scope is created where the variables declared inside the ‘init’ part of the if statement are added (we will call it the ‘init scope’), then another scope is created where all the statements residing inside the

‘then block’ are processed. Once that step is done, we pop one scope off the stack. Then we check if there is an ‘else block’ and in that case, we create a new scope and process the statements within the ‘else block’ and then pop the new scope off the stack. Once these steps are done, we pop the ‘init scope’ off the stack. We are handling the scopes this way so that both the statements inside the ‘then block’ and the ‘else block’ have access to the variables declared within the init statement (these variables reside in the ‘init scope’) while each having a separate scope of their own (i.e. the ‘then block’ does not know about the scope of the ‘else block’ and vice versa).

- When a block statement is encountered, a new scope is created and the statements within the block are processed. Once those statements are processed, the new scope is popped.
- When a switch statement is encountered, an ‘init scope’ is created where the variables that are declared inside the init statement of the switch are stored. From that point, with each case statement, a new scope is created and the statements residing within the case are processed and the scope is popped after no more statements are left inside the case. Once no more cases are left, the ‘init scope’ is popped off the stack. This strategy follows a similar scoping rule to what was done with the if statement since we wanted each case statement to have its own separate scope while they can all access the ‘init scope’.

Typechecker

Our typechecker makes use of several helper functions, which will be described, below, in the next few subsections. These helper functions are used extensively throughout the implementation of the type checker.

Helper Functions: Miscellaneous

`_resolve_type( … )`

- Return the resolved type of a given symbol.
- E.g.

```
type A int // _resolve_type( A ) -> int
type B A   // _resolve_type( B ) -> int
type C []A // _resolve_type( C ) -> []A
```

`_is_lvalue( … )`

- Return true if the expression is an “lvalue” (i.e. addressable), and false otherwise.

- Valid lvalues in GoLite include variables (non-constant), slice indexing expressions, array indexing expressions (of an addressable array), and field selection expressions (of an addressable struct).

Helper Functions: Type Classification

`_is_comparable_type( … )`

- Returns true if the type is "comparable", and false otherwise (i.e. can be used in expressions with binary operators "==" and "!=").
- Comparable types include:
    - `int, rune, float64, string, bool`
    - Array types, as long as the elements' type is comparable
    - Struct types, as long as each struct member's type is comparable

`_is_ordered_type( … )`

- Returns true if the type is "ordered", and false otherwise (i.e. can be used in expressions with binary operators "<", "<=", ">", and ">=").
- Ordered types include:
    - `int, rune, float64, string`

`_is_numeric_type( … )`

- Returns true if the type is "numeric", and false otherwise.
- Numeric types include:
    - `int, rune, float64`

`_is_integer_type( … )`

- Returns true if the type is "numeric", and false otherwise.
- Numeric types include:
    - `int, rune`

Helper Functions: Type Equality

`_types_are_identical( … )`

- Returns true if two types are identical, and false otherwise.

Type Checking: Expressions

The implementation of expression type-checking is divided into several subroutines -- one subroutine for each type of expression. These subroutines are called from a switch statement in the main `_typecheck_expr( … )` function. Each subroutine determines whether or not the sub-expressions of their main expression are type-correct (as well as the resolved types of those sub-expressions) via mutually recursive calls to `_typecheck_expr( … )` itself on the sub-expressions.

All code external to expression type-checking (e.g. the implementations for declaration and statement type-checking) need only call `_typecheck_expr( … )` to (a) determine if an expression is type-correct, and (b) to get the resolved type of that expression.

`_typecheck_expr( … )`

- Return the type of the expression, or exit with an error code of 1 if the expression is not type-correct.

Type Checking: Declarations

The implementation for declaration type-checking followed a similar approach to expression type-checking. For each type of declaration, a separate subroutine was written to verify its correctness. All these functions were then put together in the main `_typecheck_decl( … )` function, through the use of a switch construct which looks at the type of each particular declaration (package, variable, type or function) to decide with subroutine to call.

`_typecheck_decl( … )`

- Check if the declaration is well typed, if this is not the case, exit with an error code of 1.

Type Checking: Statements

In the same fashion as expression and declaration type-checking, separate subroutines were created to check that all the different statements in the GoLite specification are well typed. These functions were put together under `_typecheck_stmt( … )` using the same switch construct approach mentioned before. Moreover, statement type-checking makes extensive use of `_typecheck_expr( … )` and `_typecheck_decl( … )` to verify the correctness of the parts that make each statement construct.

`_typecheck_stmt( … )`

- Check if the statement is well typed, if this is not the case, exit with an error code of 1.

Type Checking: Top Level

In order to enable type-checking in the compiler project, a single main `typecheck( … )` typecheck function was exposed from the type-checking module. This function takes the root of the parsed AST and forwards it to `_typecheck_decl( … )` since declarations are the only top level constructs allowed in the GoLite specification.

`typecheck( … )`

- Check if the parsed AST is well typed, if this is not the case, exit with an error code of 1.
- The function expects the root of the AST which should be a declaration. This is enforced by the grammar.

Code Generator

Target Language

The first key decision in our code generator design was the target language. Originally, when we first laid out the details of this project, we decided to set the x86 assembly language (x86) provisionally as our target language while we worked on the other phases of the compiler. The reasoning behind this decision was that generating x86 code would introduce the smallest overhead between the generated code and something that the CPU could actually run, which in turn would give an overall better performance. Of course, this expectation was dependent on the fact that our compiler would generate high quality x86 code. In spite of x86 having the greatest performance potential, it was one of the most challenging choices for target language given that it lacks any kind of high level construct support or similarity with the GoLite programming language. Therefore, before starting the design of the code generator we discussed the current state of the compiler project and the time we had to finish it and we determined that using x86 as our target language was not a feasible goal within our time frame. Moreover, we decided to opt for a much higher level target language which would provide us with high level constructs similar to those supported by GoLite in order to ensure that we could carry out the development of the compiler to completion before the deadline set by the course. This of course, came at the expense of a much worse runtime performance of the generated code, but we decided it was the safer choice. Given that we wanted a language that would provide enough high level constructs and that all of our team members were already familiar with, we chose Python 3.

Although Python 3 (Python) is not the most similar language to GoLite, syntax- or feature-wise, it does provide a lot of the same high level functionality which makes the task of code generation much simpler. These are some of the features that make Python a good target language for our GoLite compiler.

Data Operations

Python presents a number of advantages when emulating the data operation behavior provided by GoLite. Firstly, Python uses a dynamic but strong typing system, which in turn implies two things for our purposes:

1. A variable can be assigned or reassigned objects of any data type.
2. Operations done on data must still conform to the rules of the type of the data.

These two properties remove the necessity to port any types defined in GoLite to Python. The assumption is that the typechecker will deal with any inconsistencies in the source code so that the generated Python code can run entirely on dynamically typed variables.

Secondly, Python supports all the data operations provided by GoLite as part of the core language. This includes boolean, integer, float and string operations. In particular, the emulation of string concatenation is significantly simplified by the fact that Python comes with a garbage collector. Moreover, because Python is strongly typed it can make the distinction between integer and float arithmetic which further facilitates the implementation of the behavior provided by GoLite.

Lists

Python provides lists as a core data structure of the language, which are dynamically sized arrays that can contain any type of data in them, including other lists. Python lists are very useful for implementing GoLite arrays because they provide index checking, thus emulating the behavior of GoLite. Moreover, lists behave almost identically to GoLite slices, with the exception of the underlying buffer management. This can be fixed by using a wrapper class, which is discussed in a later subsection.

Dictionaries

Python also provides an associative array data structure called a dictionary out of the box. Dictionaries greatly simplify the implementation of structs in GoLite. This is largely due to Python's typing system. A dictionary can contain any number of keys mapped to any type of data, which is enough given that any type checking was already done in the typechecker phase. At the same time dictionaries provide useful operations such as comparisons.

Although Python has several advantages for generating GoLite code, it also comes with its disadvantages.

Scoping

Because Python lacks variable declarations enritely, the scope of variables works differently as that of most programming languages including GoLite. For instance, in Python a variable that is created inside of an if statement is still available one the if statement ends, which is not the case in GoLite. To deal with this discrepancy, we used a numbering system for the scopes of the symbol table. Whenever a scope is created the compiler assigns a unique numeric variable to it, then when a symbol from this scope is generated, the number of the scope is appended to it (i.e. a variable 'x' would be generated as '__scope<num>_x'). This effectively mitigates all potential conflicts in the scoping behavior of Python and GoLite.

Furthermore, Python requires that global variables be declared as global inside a function before they can be used. In order to deal with this issue, we perform an analysis at the symbol table level to see which global variables are used inside each function and we store them inside in a linked list that will be used when the function is generated.

Reference Types Assignment

Another difference in the behavior of Python and GoLite is the behavior of reference types assignment. This difference arises from the fact that when a list or a struct is assigned to a variable, GoLite effectively makes a new copy or the data for the variable receiving the value. Python on the contrary, passes a reference to the same object. If a program then makes changes to this variable the behavior of GoLite and Python will not be the same. To fix this issue, we use a deep-copy operation on the data being assigned on Python. Although this fixes the problem, deep-copying a data structure is an expensive memory operation which degrades the performance of the generated code.

Loop Constructs

Loop constructs are one of the main differences between Python and GoLite. Python supports iterator and while loops, whereas GoLite supports three-part loops, and while loops. The main problem is emulating three-part loops using Python constructs. This can be done by using Python's while loops and inserting precondition and postcondition statements in the right locations of the code.

Switch Statements

A major inconvenience of Python as a target language is that it does not support switch statements at all. To get around this limitation, we used a combination of a while loop and an if statement. The if statement allowed us to separate the logic that was expected to run in each case of the switch. The while loop provided the functionality of the `break` keyword which is missing from simple if statements. Furthermore, the condition expression of the switch was only evaluated once and stored in a variable. This was done to prevent potential issues when the expression is a function with side effects, which would produce erroneous behavior if the function was evaluated when checking for each case.

If Statements

If statements are fairly similar between Python and GoLite with the exception that python does not support init statements. In order to deal with this problem, we append the scope number to each variable using the same technique we described previously. Using this, init statements can just be printed out before the 'if' construct. It is important that new variables declared in the init statement reside in their own scope (as explained in the Scoping Rules section), so that their generated name does not interfere with other variables. It is also important to note that we implement `'if{} else if{}'` as `'if{} else{if{}}'`.

The Slice Class

In order to emulate Go's behaviour with regard to slices we have decided to encapsulate slice operations inside a class. This class stores the usual slice information (length, capacity, and the underlying array which in our case is a Python list) as well as a list of one element called 'duped'. These are the attributes of our Slice class:

- 'len' and 'cap': Used to define the slice dimensions and the growth strategy.
- 'arr' : A python list that holds the slice elements.
- 'duped': A python list of one element used to define how many slice objects have access to the same underlying array. We have added this field in an attempt to optimize our slice class by removing duplication of the 'arr' when only one instance of the class is used.

The Slice class allows the following operations:

- 'dup': Duplicates the slice by creating a new object with the same fields and increment the 'duped' count by 1.
- 'append': Add a new element to a copy of the slice without forgetting to duplicate the underlying array and len equals to cap and if slice has been duped previously (i.e. duped[0] != 1). It returns the slice copy.
- 'fast_append': Does the same operations as 'append' except that it does not copy the slice but instead appends the element directly to itself. This function is used in cases where we have the following pattern: x = append(x, element), where x can be any slice variable.
- The magic methods '__getitem__' and '__setitem__' are in order to simplify indexing and thus allows us to index slices the same way we would index arrays while keeping the correct behaviour for bound checking. This comes in handy when two slices have the same underlying array and one slice's length is greater than the other one's.
- The magic method '__del__' is used to decrement the 'duped' count so that we can keep track of which slices are linked to which underlying array. This method is directly called by the garbage collector when a slice reference is no longer needed by the program.

Note: The ‘fast_append’ and ‘duped’ functionalities were not part of our original design (as of Milestone 4), we added them later as explained in the next section.

Implementation

General Structure

The generated code has the following structure (in order):

- The helper methods and Slice class used throughout the program.
- Generate all the code structures from the go program (with renaming as explained in the ‘Scoping’ chapter).
- Generate a Python main function that calls all the go init functions in order, then calls the main function. When no init or main functions are present, this step is ignored.

Slice Runtime Issues And Optimizations

The Milestone 4 programs shed light to huge runtime delays mainly due to use of the slice class which made some of the benchmarks fail. In order to deal with this problem with this problem we tried three solutions:

- Change the original slice class implementation to inherit from the Python list class. This resulted in a speed-up of around 300%, but some of the semantic tests for GoLite began to fail due to this change.
- Adding extra information in order to prevent from copying the slice’s underlying elements when not needed (this is done through the ‘duped’ field explained previously).
- Optimizing appends checking at compile time if the append result is assigned to the same object and if so we use the ‘fast_append’ method that removes the need to create a new object and the need for the garbage collection to collect the dead instance of the same slice.

Although the strategies described above did improve our runtime, we were still not able to meet the benchmark’s requirements and so we have decided to go with a more radical approach. For each slice variable, we need to determine if using an actual slice representation is necessary, if not, we generate the slice as a normal Python list. We have followed the following rules to determine if a slice variable needs to be used as a slice:

- If it is used as the expression inside of a return statement.
- If it is used with the ‘len’ or ‘cap’. (We could remove this restriction and determine the length and capacity using other techniques than reading them the slice class fields. However, given our benchmark targets and time limitations, we decided to do it this way).
- If it is passed as a function argument to a function that requires that exact argument to be a real slice. For example, let’s assume we have function ‘foo(x []int)’ where x is

required to be a real slice foo is called from another function using ‘foo(y)’, then y has to be a real slice as well. (Note: This requirement again is not necessary as we could have had found some other way to ‘fake’ slice into a real slice without losing the target behaviour)

- If append is used with a function parameter, the function parameter is required to be a real slice. For example:

```
foo(x []int){
      x = append(x, 3)
}
```

- In cases of multidimensional slices (i.e. `var x [][]int` and similar), if one of dimensions is required to be a real slice, then all the other dimensions have to be real slices. This is enforced by running the processing routine twice and checking that when using append, both parameters to append are either both real slices or both fake slices. Since some information does not always backpropagate properly.
- When append is used but the source slice is not the same as the destination slice (i.e. `y = append(x, element)`).

Once we have made this distinction between fake and real slices, we can easily generate real slices by using the slice class explained previously and usual python lists for fake slices.

Notes:

- In the above explanation, we were using x and y for slices, however, our program does not require the two slices to be identifiers, they can be any valid slice in our context. For example: `x[0] = append(x[0], element).`
- We were not able to fully test this functionality so there might be cases where we failed to determine the slice behaviour. However, we were able to pass all the test cases at our disposition using the above mentioned restrictions.

Conclusion

Overall, we thoroughly enjoyed this project, although we are walking around from it with mixed feelings towards the Go language. Angel likes the simple syntax of the language and feels neutral towards Go as a whole. Zak finds slices to be quite tricky, but doesn’t consider it to be a dealbreaker. Josh considers Go to be a terrible language for terrible people who hate babies, puppies, and software developers.

Our main takeaways with regards to this project were that a language’s grammar grows exponentially (not linearly) with the number of features supported by the language, and that the difficulty of code generation is very dependent on the chosen target language. For example, we personally found the hardest part of the project to be symbol table construction, whereas code generation was fairly easy to implement. We attribute this to the fact that we chose Python to be our target language. However, we did experience difficulties generating

Python code that was fast enough to pass the benchmarks. After trial and error, we were able to resolve this issue and pass all given tests.

If we had more time to work on this project, we would have prefered to target a lower-level language, such as C, x86, or LLVM IR. We suspect that this would have made code generation more difficult, but passing the benchmarks would have been less of an issue, and we generally agree that this would have given us a greater sense of accomplishment in the end. Additionally, we wish that we had spent more time planning out the proper design for this project, rather than jumping straight into the implementation. One example of how our code was negatively impacted by our eagerness to start the implementation is evident in our "type" data structure, which could have been used in both the symbol table and the type-checker, but was not.

In conclusion, this project gave us great practical experience in compiler design, and we look forward to making use of what we learned in our own future projects.

Contribution

The work for the GoLite compiler was distributed as follows:

- Lexer: Zakaria Essadaoui
- Parser: Angel Ortiz Regules
- AST (First version): Joshua Inscoe
- AST (Second version): Zakaria Essadaoui
- Weeder: Zakaria Essadaoui and Angel Ortiz Regules
- Pretty Printing: Zakaria Essadaoui
- Symbol Table: Zakaria Essadaoui
- Typechecker: Joshua Inscoe and Angel Ortiz Regules
- Code Generator: Zakaria Essadaoui, Angel Ortiz Regules and Joshua Inscoe
- Final Report: Zakaria Essadaoui, Angel Ortiz Regules and Joshua Inscoe

References

We worked alone.

Appendix 1: The Slice Class

```
class __codegen_helper_slice:
    def __init__(self, slice_object=None):
        if slice_object is not None:
            self.len = slice_object.len
            self.cap = slice_object.cap
            self.arr = slice_object.arr
            self.duped = slice_object.duped
            slice_object.duped[0] += 1
        else:
            self.len = 0
            self.cap = 0
            self.duped = [1]
            self.arr = []

    @classmethod
    def dup(cls, rhs):
        new_slice = cls()
        new_slice.len = rhs.len
        new_slice.cap = rhs.cap
        new_slice.arr = rhs.arr
        rhs.duped[0] += 1
        new_slice.duped = rhs.duped
        return new_slice

    def append(self, element):
        copy = self.dup(self)

        if copy.cap == 0:
            copy.cap = 2

        if copy.len == copy.cap:
            if copy.duped[0] != 1:
               temp = []
               for x in copy.arr:
                   temp.append(x)
               copy.arr = temp
               copy.duped = [1]
            copy.cap = copy.cap * 2
        if copy.len >= len(copy.arr):
            copy.arr.append(type(element)(element))
        else:
            copy.arr[copy.len] = type(element)(element)

        copy.len = copy.len + 1
        return copy

    def fast_append(self, element):
        if self.cap == 0:
            self.cap = 2

        if self.len == self.cap:
            if self.duped[0] != 1:
               temp = []
               for x in self.arr:
                   temp.append(x)
               self.arr = temp
               self.duped = [1]
            self.cap = self.cap * 2
        if self.len >= len(self.arr):
            self.arr.append(type(element)(element))
        else:
            self.arr[self.len] = type(element)(element)

        self.len = self.len + 1

    def __getitem__(self, index):
        if (index >= self.len):
            raise IndexError(
                'Slice index out of range.'
            )
        return self.arr[index]

    def __setitem__(self, index, val):
        if (index >= self.len):
            raise IndexError(
                'Slice index out of range.'
            )
        self.arr[index] = val

    def __del__(self):
        self.duped[0] -=1
```