McGill University

COMP 520 - Compiler Design

Gompiler - glc, a GoLite Compiler

Final Report

Lore, J.
Lougheed D.
Wang A.

April 14, 2019

Contents

1 Introduction

1.1 Overview of Go

Go[15] is an open source imperative programming language created by Rob Pike, Ken Thompson and Robert Griesemer (Google employees). It was designed for easier code writing, having built-in concurrency support and speedy compilations. It features goroutines (lightweight thread), channels, interfaces, methods, closures, defer, maps, slices, multiple return values, modules, garbage collection and optional semi-colons. See the Golang main page[5] for more information.

1.2 Overview of GoLite

GoLite[15] is a (slightly modified) subset of Go. It does not support goroutines, channels, interfaces, methods (functions are still supported), closures, defer, maps, multiple return values, modules, garbage collection and optional semi-colons (insertion rule[7] 1 is supported, not 2). Additionally, it is restricted to ASCII encoding (instead of UTF-8), has a fixed precision, no imaginary numbers and has a few extra keywords (`print`, `println`, `append`, `cap`). The base types only include `int`, `float64`, `bool`, `rune` and `string`. Declarations must also come before their use (for variables, functions and types) and constant declarations are not supported. Function bodies should also not be empty. For range loops, labelled breaks/continues and fallthrough are not supported as well. GoLite is the source language of our compiler.

1.3 Structure of Report

In this report, we discuss the language and tool choices we made for this project, as well as an overview of each compiler step (scanner, parser, weeder, symbol table, typecheck and codegen respectively), with major design decisions. Testing is also discussed.

2 Language and Tool Choices

2.1 Implementation Language: Haskell

For our `GoLite` compiler, we decided to use `Haskell` as the implementation language, as many of the required tasks can be done more naturally (i.e. tree traversal, recursing over self defined structures, enforcing types, etc.).

2.2 Tools

For scanning, we use the program `Alex`[1], a lexical analyser generator which is similar to `flex` but produces `Haskell` code instead. We use `Happy`[8] for parsing, a parser generator which resembles `bison/yacc`. `Alex` and `Happy` can be interlinked nicely, passing on useful information such as the line number, column number, character offset in file, and error messages.

2.3 Miscellaneous

We use Travis CI[14] to build our project on every push to GitHub, as well as run our tests to make sure that new changes did not break existing functionality.

There are several other good Haskell scanning/parsing options, such as `Megaparsec`. We decided on `Alex/Happy` because of their syntactic similarity to `flex/bison`, which we learned in class.

We use `stack`[9] to manage our Haskell project and corresponding tests. We implemented additional tests using the `hspec`[10] package.

2.4 Target Language: JVM Bytecode

We decided on targeting JVM bytecode for our compiler, through the Krakatau[13] bytecode assembler. Krakatau bytecode syntax is derived from Jasmin, but with a more modern codebase (written in Python) and some additional features.

We chose Krakatau as it is more widely supported, much more popular (985 stars on Github[13]) and has additional features when compared to other bytecode assemblers, such as Jasmin.

2.4.1 Advantages

The primary advantages of targeting JVM bytecode are: portability, execution speed, and its focus on stack manipulation, as opposed to a higher level language, which aids in overcoming some of the common pain points of GoLite code generation:

Portability The JVM has been ported to many common platforms, meaning code written in GoLite, when compiled with our compiler, will be able to run on any platform the JVM can run on.

Execution Speed Although Java is often considered slow as opposed to ahead-of-time compiled languages such as C and C++ due to its garbage collection and non-native compiled code, most implementations of the JVM provide JIT compilation. By targeting JVM bytecode, we take advantage of this, and our generated code is often faster than if we generated code in an interpreted language such as Python. Sometimes, the JVM can be faster than even ahead-of-time compiled programs, since run-time information is available for optimization purposes.

Stack Based/Low Level The fact that JVM bytecode is fairly low level gives us lots of granular control for changing the behavior of constructs, especially when dealing with odd `GoLite` / `GoLang` behavior that doesn't map perfectly to more common programming languages.

Operating on a stack makes some of the operations mentioned in class as 'difficult to implement' surprisingly easy. In particular, swapping variables (e.g. `a, b = b, a`) is fairly straightforward. The right-hand side must be evaluated before assignment, which can be done by pushing and evaluating all RHS terms, left to right, onto the stack; followed by popping each value in turn and loading them into corresponding locals. Compared to temporary variable or register allocation, this is a very natural way to implement this construct.

2.4.2 Disadvantages

The main disadvantages of generating JVM bytecode are its low-level semantics and its slightly slow speeds versus an ahead-of-time compiled language (in general).

The low level of JVM bytecode is particularly frustrating to deal with when it comes to types that are not representable by an integer or floating point number and are thus classes which must be instantiated; in the JVM, this includes strings and `struct`. This means that operations such as comparisons and string concatenations go from being a few bytecode instructions to significantly longer patterns.

3 Scanner

3.1 Overview

The first step of the compiler pipeline is to scan user input into tokens, abstract representations of each separate expression in the source code. To do this, we use the tool `Alex`[1] and define regular expression rules (`golite.x`) to match certain corresponding tokens. Handing of most GoLite tokens was straightforward and similar to class assignments. In other cases, we had to deal with some of Go/GoLite's quirkiness when compared to something simple like `MiniLang`.

3.2 Design Decisions

3.2.1 Semicolon Insertion

Semicolon insertion was the first scanning hurdle. To solve this, we use start codes, a built in feature of `Alex`, and have special rules for certain states.

If the last token we scan is something that can take an optional semicolon, we enter the *nl* state, where newlines are transformed into semicolons after specific tokens. Scanning anything else (that isn't whitespace or otherwise ignored) returns the scanner to the default state (0), where newlines are just ignored.

This solution seems elegant, as we don't have to traverse the whole stream post-scanning or store context, other than the start code. It also makes sense to let the scanner handle these situations automatically, rather than trying to deal with them in the parsing phase.

3.2.2 Block comment support

Block comments cannot be easily scanned using only regular expressions. One potential solution was to use start codes again (changing state upon opening a comment). This was ruled out since it would potentially add a lot more start codes (i.e. states), as we'd have combinatorial possibilities with the newline insertion state (since block comments spanning newlines should also insert semicolons when they're optional). Using a state approach would also make it harder to catch unclosed comment block errors.

Our solution iterates through the scanner's input once we receive an open comment block and ignore everything until the comment is closed. If the comment never closes, we can easily emit an error.

To account for semicolon insertion with block comments, we set a semicolon flag to true if we encounter any new line inside the block comment, and we insert a semicolon if we're in the aforementioned newline insertion state.

Adding newlines at the end of the file if they aren't present already `Go` requires a semicolon or newline at the end of function declarations. Sometimes, there may not be a newline at the end of a file, so no semicolon insertion occurs, resulting in a parsing error.

In order to correct for this type of file, we enforce a final newline by appending one if necessary. To do this, the code string is preprocessed prior to scanning. This is the easiest solution, as otherwise we would have to try inserting a semicolon if we are in the *nl* state when we encounter an `EOF`, while also making sure to return an `EOF`. This would be difficult without additional context information.

3.2.3 Nicer error messages

We decided to use `ErrorBundle` from `Megaparsec`[11] in order to output nicer error messages, with program context for easier debugging from an end-user programming perspective:

```
Error: parsing error, unexpected ) at 5:22:
  |
5 | func abstract(a, b, c) {
  |                      ^
```

With `Alex`' default behaviour, we did not have access to the entire source file string, as it is not kept between steps. In order to generate the contextual message, we modified the `monad` wrapper provided with `Alex` (see `TokensBase.hs`) and changed the `Alex` monad to wrap over a `Either (String, Int) a` instead of `Either String a`, i.e. in addition to storing an error message on the left side of the monad we also carry an `Int` which represents the offset of the error. When we want to print the error message, we can then append the part in the source file where the error occurred.

3.3 Testing

To test the scanner, we implemented hspec tests that test the token output of scanning certain strings, lots of which was a reverse mapping of the scanner rules (to make sure things mapped to what we expected). Additionally, we added random generation of hex, octal, decimal, float numbers, random characters, strings and identifiers, to make sure everything maps as expected.

4 Parser

4.1 Overview

After having scanned in the source code into tokens, we now need a more structured representation of the program, namely an AST or abstract syntax tree. In order to accomplish this, we use `Happy`[8] and define the rules of our grammar (`golite.y`) that will transform certain sequences of tokens into a corresponding part of the AST.

4.2 Design Decisions

4.2.1 Grammar

Many of our difficulties in the grammar were associated with identifier and expression lists, used in declarations/signatures and assignment/function calls respectively. The grammar was refactored to fix this by allowing identifier lists to become expression lists if needed, in a way which avoided introducing other conflicts.

The first issue we encountered was with list ordering. LR parsers work more intuitively with rules that put the newly-created terminal after the recursively-expanding non-terminal. However, since Haskell uses recursive lists defined in the opposite way, it is significantly more efficient to prepend items. This prepending results in a reversed ordering, which must be handled after the list is 'complete'.

Adding an extra non terminal to manage reversals for each list would needlessly increase our grammar and generated code size, so we decided against it as a solution. The solution we use is to differentiate lists containing at least one non-identifier expression (i.e. using either all non-identifiers, identifiers plus one non-identifier, or otherwise mixed lists, as grammar base cases) from lists of entirely identifiers. Then, the expression list non-terminal is allowed to yield either a mixed list or a pure identifier list depending on what is needed.

Another caveat of how lists are handled in the grammar, again a compromise to prevent ambiguity, is that the actual grammar constructs that represent lists correspond to a list of size two or more, which doesn't exactly match the Go spec (where a list may be 0/1 or more, depending on the case). The actual single-item non-terminals are allowed to represent a list of size one when needed, meaning this disparity is resolved in the actual AST construct, which is closer to a direct representation of the Go / GoLite specifications.

4.2.2 AST

We modeled our AST as close as possible to the actual Go and GoLite specs, to try and ensure that impossible states are inherently prevented by the Haskell type checker, reducing run-time errors. Although we don't have type-checking implemented at this milestone, we can use this technique to enforce definitions such as 'exactly one', 'one or more', and 'zero or one'. This modeling is not always perfect. For example, a list of identifiers[6] is 'one or more' (in Haskell, `NonEmpty`). Many locations make it optional. While a direct translation would be `Maybe (NonEmpty a)`, we choose to make it a possibly empty list `[a]` as it makes more sense.

Simplified Data Type Categories Some splits, such as `add_op` and `mul_op` are distinguished purely to demonstrate precedence; they are in fact only used once in the specs, so we decide to merge them directly in our `ArithmOp` model. Several other instances exist.

Given we are creating an AST, rather than a CST, we can further compact parts of the grammar. For instance, an `if` clause in the spec leads to an `IfStmt` construction, whose `else` body is either a block (with surrounding braces) or another `if` statement (no surrounding braces). In our case, we don't need to model the braces, so we can treat the `else` body exclusively as `Stmt` rather than the more verbose `Either Block IfStmt`. The grammar enforces that this `Stmt` is not any other type. Some splits, such as `add_op` and `mul_op` are distinguished purely to demonstrate precedence; they are in fact only used once in the specs, so we decide to merge them directly in our `ArithmOp` model. Several other instances exist.

Given we are creating an AST, rather than a CST, we can further compact parts of the grammar. For instance, an `if` clause in the spec leads to an `IfStmt` construction, whose `else` body is either a block (with surrounding braces) or another `if` statement (no surrounding braces). In our case, we don't need to model the braces, so we can treat the `else` body exclusively as `Stmt` rather than the more verbose `Either Block IfStmt`. The grammar enforces that this `Stmt` is not any other type.

Structure Simplification For `var` and `type` declaration, we make no distinction between single declaration (exactly one) and block declaration (0 or more). Unlike types, which produce different formats, we decide to enforce all var declarations to be single declarations. In other words, `var ( a = 2; b = 3 )` becomes `var a = 2; var b = 3` and `var ( a = 2 )` would become `var a = 2`. Note that we cannot further simplify multi declarations `var a, b = 2, 3` because they happen in one stage; `var a, b = b, a` does not always produce the same output as `var a = b; var b = a`.

4.2.3 Pretty Printer

When creating our pretty printer, we chose a top down approach. Every node has the ability to output a list of strings, which makes it easier to format indentation. Each node is also only concerned with its respective subtree, and does not require context from its parent. We focused on aesthetics, focusing on proper spacing and alignments. In the case of expressions, we tried to add brackets sparingly, though further optimizations can be done down the road (a nested binary op does not always need brackets, if the order of precedence matches). To produce the full program, we simply join the list of strings in the full program, intercalated with new lines.

4.3 Testing

To facilitate testing, we wanted to ensure that each feature can be tested separately. When building the parser, we expose entry points to each main stage, so we can focus on tests for floats, expressions, statements, etc on top of the full program. To facilitate future refactoring, we base our tests on quality checks, vs structural checks. In other words, we simply check if a parser succeeds or fails. This was very important, as testing the full AST structure meant that it was very difficult to make changes in the future.

On top of the necessary tests with the provided scripts, we wanted as much testing within Haskell as we can. Most inputs are embedded to avoid IO operations, and are done with the help of `hspec` so we can make use of Haskell's testing stack. This allows us to easily run all tests, generate coverage reports, and rerun selective tests if we need to.

To ensure incremental updates, we wanted to make sure that we tested every step of the way. As a result, every build, with the exception of updating documentation, ran through Travis CI.

5 Weeder

5.1 Overview

Certain language constraints are too complicated to implement in the parsing stage (or at least without greatly complicating the rules of the grammar). Take for example checking if the LHS and RHS of an assignment have the same number of expressions. If one wanted to have that in a grammar, they'd need specific rules for each length (or they'd have to define the LHS and RHS of assignment lists as one language construct instead of just being expression lists). Thus, in order to get rid of the syntactically invalid parts of our grammar, we weed through the results of the parser, checking that certain constrains that aren't enforced hold, else we'd error on said source. To do this, we recurse over our AST and check the constraints.

5.2 Design Decisions

5.2.1 Parser Weeding

Most of the weeding operations needed are simple and don't rely on external context. As a result, we were able to define recursive traversal methods to verify relevant statements, and create verifiers that validate at a single level. Haskell helped immensely here, as we were able to use pattern matching to produce performant and independent functions.

For verification where statement context was important (`break` and `continue`) we made a simple modification to the recursive traversal to avoid exploring scopes under `for` or `switch` statements where applicable.

Each verifier returns an optional error, and we are able to map the results and return the first error, if any.

5.2.2 Typecheck Weeding

We implemented additional weeding passes for certain constraints that could be verified either at the weeding level or the typecheck level, since in most cases it was easier to check via weeding. The constraints we use weeding to check for this milestone are:

- Checking correct use of the blank identifier. It was much easier to recurse through the AST and gather all identifiers that cannot be blank, and then check this whole list, versus checking usage in the type-checking pass. Additionally, because we have offsets in our AST, we could easily point to the offending blank identifier without having to make error messages for each specific incorrect usage, as it is obvious what the incorrect usage is when we print out the location of the blank identifier.
- Ensuring non-void function bodies end in return statements. It was easier to do a single weeding pass; otherwise, we'd have to recurse differently on functions that have a return type versus functions that don't have a return type during typechecking. Essentially, we'd need context-sensitive statement typechecking, with two different versions depending on return type. This was not deemed worth it compared to a single weeding pass.
- Ensuring void function declarations do not have any non-void return statements. Similar to the above, it is easier to do a weeding pass then require context-sensitive (i.e multiple different) traversal functions given the context of the current function declaration.
- Ensuring any 'main' or 'init' function declarations do not have any parameters or a return type. This is again a straightforward weeding check. It could also have been done via type-check, but it is slightly easier to implement with weeding passes (no symbol table required).
- Checking that any top-level declarations with the identifier `init` or `main` are functions, since they are special identifiers in Go (and GoLite) in the global scope.

5.3 Testing

Testing weeding rules is relatively simple. We simply provide a program that should parse, verify that it does parse, and also verify that it fails the weeding rule.

At this stage, we decided to improve our testing process for invalid programs. Given that error messages are unrelated to AST structure, we created a new error type class, where we represent error strings with data structures. For instance, each weeding rule will have its own constructor, with optional arguments depending on how much detail we wish to add. We then create a function that converts each data type to a string.

When testing for invalid programs, we can now also check that the message contains the string of our desired error. Given that we have a unique prefix, we can ensure that there are no accidental matches with other errors. This ended up being useful for two reasons: It allowed us to create errors in a more compact way, and avoid the possibility of making typos when pasting error message strings in numerous locations. It also ensured that invalid programs were invalid for the reasons we expect. After we refactored to the new model, we noticed that some of our old tests were indeed invalid for the wrong reasons.

6 Symbol Table

6.1 Overview

Before one can typecheck their AST to make sure it is semantically valid, they must create a symbol table that keeps track of the type of identifiers (variables, type maps, function calls), such that when we typecheck we can find out the type of a variable. This is usually done using a cactus stack (i.e. a stack of hash maps representing different scopes), which is the approach we used as well.

6.2 Design Decisions

6.2.1 State Monad

In Haskell, data structures are typically immutable, and much of the language is designed around this. One of the main design decisions made around the symbol table was deciding whether to go with an immutable or mutable symbol table. In an immutable symbol table, a new symbol table would have to be made every time a scope is added or modified. Right away, despite this being a conceptually better fit for the language, the potential performance degradation of constantly re-building the symbol table becomes evident.

As a result of this performance impact, we decided on using a mutable symbol table, with mutability supported via Haskell's ST[3] monad. The constraint provides runST[4] (which removes the `ST`, i.e. mutability, from an interior data structure). This is proven[2] to keep functions pure. This made the `ST` monad a better choice than other monads providing mutability (the main example being `IO`, which if used would have resulted in all our functions after symbol table generation being bound by `IO`, i.e. impure and also harder to work with, as they are wrapped by an unnecessary monad). The trade-off of this decision was a large increase in the difficulty of implementing the symbol table, which made up a huge portion of the work for this milestone. However, once we finish using the symbol table, our final result (a typechecked/simplified, proven-correct AST) is pure and very easy to manipulate for the `codegen` phase in the next milestone.

6.2.2 Core Model

The symbol table needs to account for a lot of constraints related to typechecking. However, the core state model itself is simpler, so we decide to separate the two. Our core data structure is simply a list of scopes, where each scope contains an index, hashtable (with key type `k` and value type `v`), and optional context (of type `c`). We then provide some general functions, such as retrieving value `v` given `k`, which recurses through all tables and finds the first match, if any. We also provide a `wrap` function, which allows us to create a new scope, execute some actions, then exit. Note that we do not expose enter and exit, as it should be impossible to do only one of the two operations.

While messages is separate, we add a list of type `m` to our core model, as well as a functions to add a message or disable messages. The need for disabling is because some typecheck errors are not actually symbol table errors. In the event that we receive a typecheck error that is valid for the symbol table, we disable message outputs.

This separation of core logic to the symbol table logic helps simplify our constraints, as well as testing (see 6.3.1).

6.2.3 Symbol Data

For our first iteration of symbol table storage, we created a new data type `Symbol` to represent each symbol, analogous to the different type of symbols (types, constants, functions, and variables). With respect to our core, this represents type `v` and type `m`. A `Symbol` can be one of:

- `Base`: Base types (`int`, `float64`, etc.)
- `Constant`: Constant values (only used for booleans in GoLite)
- `Func [Param] (Maybe SType)`: Function types, with parameters and an optional return type.

- `Variable SType`: Declared variables, of type SType.
- `SType SType`: Declared types (note: in Haskell, the first `SType` here is a constructor, and the second `SType` is a data type attached to the constructor.) In this way, all possible symbol types are encompassed by a single Haskell data type.

We also created a new data type `SType`, which is used to store vital information concerning the types we define in the original `AST`. An `SType` can be one of:

- `Array Int SType`: An array with a length and a type.
- `Slice SType`: A slice with a type.
- `Struct [Field]`: A struct with a list of fields.
- `TypeMap SIdent SType`: A user-defined type, with an identifier and an underlying type, which may be recursively mapped, eventually to a base type.
- `PInt, PFloat64, PBool, PRune, PString`: Base types.
- `Infer, Void`: Special `SType` for inferred types and void return values.

In this way, all possible GoLite types, including user-defined types, are accounted for.

All types that are left to be inferred during symbol table generation are updated when type-checking (we infer the type of variables and then update their value in the symbol table, to make sure things like assignments don't conflict with the original inferred type).

6.2.4 Cyclic Types

Initially, our model above was sufficient in defining all type variants. Even if a type were cyclic, we could create a data structure that references itself. Given that Haskell is lazy, it will only generate nested types as we needed. While this is an elegant solution, it unfortunately makes modifying types difficult, and checking type equality near impossible. Our solution is to introduce a new cyclic model (`Cyclic.hs`). For a generic data structure `T` to be cyclic:

- It must provide `isRoot :: T -> Bool`, to check if the data is a root type. This is usually an empty constructor, which labels a field as a root type.
- It must provide `hasRoot :: T -> Bool`, to check if the data contains a root type, and is therefore cyclic.

We can then create a new model for any data structure, which takes in two values: the root data and the current data. When we retrieve the current data, we can check if it should be the root, and return either the root data or the current data respectively. This allows us to create a recursive type by access, but where the data structure itself is non recursive. Creating the data is simple, as we simply set the initial data as both the root and the current values. Subsequent changes involve retaining the root, while only modifying the value.

The use of `hasRoot` is namely to provide valid equality logic. If the current type has a root value, we must also check root equality. If not, only current data equality is required.

6.3 Testing

6.3.1 Symbol Table Core

When testing symbol table core, we provide a data type containing potential actions. For instance, we can look up, insert, or check messages. To make things easier, we can replace the types for keys, values, and messages to integers, since we aren't testing actual symbol table logic. Afterwards, we can simply run a chain of actions on a single symbol table, where each action results in an `hspec` expectation. If all expectations are met, then the test passes.

6.3.2 Symbol Table

Since the symbol table is mainly just a string output, it was harder to test. We added a few cases to check for success/no error, but as for the actual printing, most of that was checked manually by running it against programs as we weren't really able to add symbol table string checks due to time constraints. Most of our testing of the symbol table module were done as typecheck tests (as those were a lot more important and core to the actual compilation process).

7 Typecheck

7.1 Overview

Before we can try to convert our AST into our target language (i.e. code generation), we must verify that it is semantically correct such that our generated code will also be semantically correct (we don't want to assign a string to a variable that is of type float, for example). To do this, we recurse over our AST while generating the symbol table and check that everything is well-typed using the typecheck rules/spec[16].

7.2 Design Decisions

For type-checking, we decided on a single-pass approach, combining symbol table generation and statement type-checking. This improves performance, and is feasible as a product of GoLite's declaration rules, which specify that identifiers must be declared before they can be used.

The other approach we considered involved generation of a type-annotated AST, with types of expressions contained in the AST, so that we could get rid of `ST` mutability from the symbol table as soon as possible (some of us did a similar thing for the assignment, but this was mainly relevant for print statements in C codegen needing to know the type of the expression they're printing).

We decided on doing all typechecking at the same time as symbol table generation because type inference has to be done to generate this new AST, and type inference requires typechecking (e.g. `"a" + 5` has no inferred type, since the expression is undefined; we only know this because of typechecking).

At first, we were going to generate an annotated AST only to typecheck things that aren't expressions. However, at that point, since we were already doing one in-depth pass of the original AST for symbol table generation, we decided that we might as well do the other half of typechecking in the same phase, since it seemed odd to split typechecking between the symbol table and a separate pass. The alternate approach may have been more feasible if type inference did not require typechecking, but in GoLite it did not seem to make sense. Therefore, after the one pass of our original AST, the final result is a typechecked AST, with extremely limited type annotation.

Additionally, we decided to resolve all type mappings (except for structs) to their base types when generating this new AST: all the casts/equality checks/new type usages are already validated in typechecking, so we don't need them anymore, nor do we need the mappings. Thus, our new AST was also able to get rid of type declarations (except for structs).

7.2.1 Functions

Because the name of a function should overshadow all names in earlier scopes (i.e. base types), we add the function to the symbol table without any arguments before typechecking. We then typecheck the parameters (if we didn't add the function yet the parameters could incorrectly refer to a mapping of the function name in a previous scope) and update the function's entry in the symbol table.

7.2.2 New AST

As mentioned above, dependency on the SymbolTable results in a dependency on the `ST` monad, which adds complexity to each operation. As a result, after typechecking we create a new AST, which reflects the new constraints we enforce. Namely:

- Typecheck errors are caught beforehand, so we no longer need offsets, or error breakpoints.
- All variables are properly typechecked, and therefore reference an explicit type. Each type is composed of potential user-defined parent types, which eventually map to a GoLite data type. This includes cases like function signatures, where we can associate each parameter with a type rather than allowing lists of identifiers to map to a single type, syntax sugar present in GoLite and in the first AST. In preparation for code generation, we use our new AST exclusively, without any other mutable data structures. Any additional information we need for code generation is added to a third AST downstream, called the resource AST, which has minimal changes to models used at this and previous previous stages.

An added benefit of adding items to the AST is to support caching. Haskell computes data lazily, so having any argument within a data type will ensure at most one computation. As shown above, providing a way of getting expression types is very important. While we do have all the information we need within the previous AST, we felt it was inefficient to require recursion through binary and unary expressions. Additionally, recursion for binary expressions may lead to errors in the event that the two types do not match. To avoid this possibility, we simply offload the verification to the symbol table, and assume that the provided type is valid. We add the explicit type to the ast, and now future stages simply require a field access to find it.

7.3 Testing

To test typecheck rules, we provide a program that should either pass or fail the typecheck phase. Most of our tests were implemented as hspec tests, but we added a few invalid typecheck programs as files for submission (see the appendix).

8 Code Generator

8.1 Overview

As discussed in the Language and Tool Choices section, we decided on targeting JVM bytecode for our compiler's code generation, using the Krakatau bytecode dialect and assembler. JVM bytecode posed an interesting challenge due to its low-level syntax and semantics, while providing a well-tested and established set of libraries (the Java standard library).

In order to generate bytecode, we decided to first create a bytecode-adjacent intermediate representation (IR) in Haskell. This allowed us to reduce errors in code generation by taking advantage of Haskell's strong type system to represent common bytecode constructs, and let us write a common mapping from the IR to bytecode in order to reduce the chance of typos.

Four major concepts that had to be handled carefully when converting GoLite to JVM bytecode were identifier scoping, assignments, struct generation, and copying behaviours. These are ideas that do not map exactly to bytecode, since bytecode does not have scoping at all, with the exception of method locals and classes, assignments (storing values) work completely differently, and the JVM is pass-by-reference for assignments and method calls.

While we were given the general outline for most bytecode generations in class, we did not discuss switch/case statements. Mapping the GoLite form of these constructs to bytecode proved interesting, especially considering limitations on execution of the switch expressions and laying out labels to provide proper default branch behaviour.

8.1.1 Scoping in GoLite

In `GoLite`, new scopes are opened for block statements, `for` loops, `if` / `else` statements and function declarations (for the parameters and the function body). A new scope separates identifiers (which are associated with type maps, variables, functions and the constants `true` and `false`) from the other scopes' identifiers. Whenever we refer to an identifier, it references the identifier declared in the closest scope.

There is nothing very special about scoping in `GoLite`. The main notable thing is that something like `var a = a` will refer to `a` in a previous scope, not the current `a` that was just declared, unlike languages like `C`. On the other hand, recursive types such as `type b b` fail as expected, and do not reference a type from higher scopes.

8.1.2 Switch Statements in GoLite

In `GoLite`, `switch` statements consist of an optional simple statement, an optional expression and a list of case statements. Case statements are either a case with a non-empty list of expressions, or a default case with no additional expression. Each case statement also contains a block statement, containing code to execute upon match. This makes them structurally different when compared to Java or `C` / `C++`, where:

- Simple statements don't exist.
- Expressions aren't optional.
- Case statements don't match on a list of expressions.

The simple statement is executed before the `case` checking. After that, the optional expression is compared with each `case` statement, evaluating and comparing expression lists from left to right. The first match enters that case's body, automatically breaking at the end of it. This makes cases significantly different semantically:

- Cases automatically break.
- Each `case` or `default` block defines its own scope for declarations.
- Case statement expressions do not need to be a constant expression.

8.1.3 Assignments in GoLite

In `GoLite`, assignments are either an assignment operator with a single LHS expression and a RHS expression, or two non-empty expression lists (LHS and RHS) of equal length. This makes them structurally different (for the two non-empty list case) from 'classic' assignments, which typically only allow one l-value. This structural difference is a lot more significant than it seems at first glance, because the assignments are done in a "simultaneous" way, that is `a, b = b, a` will swap the values of `a` and `b`. If the assignments were done sequentially, `a` and `b` would be the original value of `b` and wouldn't be swapped.

8.1.4 Structs in GoLite

`GoLite` provides structs as a way of defining custom data structures. These do not exist in the JVM, which uses classes for user-defined data types. Transforming structs into classes was, for the most part, a natural mapping, although copying and equality proved difficult to get correct.

8.1.5 Copying Behaviour in GoLite

In `GoLite`, assignments and function calls copy the values used. This is in contrast with the JVM, which uses references for object assignment and method calls. Overriding this behaviour was difficult, since it involved writing custom methods for structs, arrays, and slices.

8.2 Mapping Strategies

8.2.1 Scoping

JVM bytecode only has "scoping" for `methods`, as they have their own locals and stack. Block statements do not exist per se, and statements except called method bodies are scope-less. In higher level languages, we could just append the scope depth to all identifier names to keep them unique, also eliminating the need for separate scopes, as we already typecheck the correct use of identifiers. In our case, we have scoped identifiers in our newly generated typechecked AST which we convert to offsets (for locals) in the resource AST (see 8.3).

These offsets are in our intermediate representation, where the offsets are unique for each local variable declaration. Variables with the same scoped identifier will be given the same offset, and we can optimize our stack limit by reusing offsets when two variables can never occur at the same time due to branching.

Globals are a special case in JVM bytecode generation, since they are equivalent to fields on the main class. To generate these, the identifiers are kept and a prefix is prepended to the name: `__glc$fd__`. This ensures that the field names will not conflict with any reserved word in the JVM. Since only top-level variables retain identifiers, there is no concern about clashing names here.

8.2.2 Switch Statements

For the structural differences:

- Simple statements are modeled and executed as the first statement in the new "scope".
- Missing expressions are converted to the constant literal 'true' in the resource AST, prior to IR and code generation.
- For a list of expressions that is of length greater than one, we compare each element from the list one at a time, duplicating the value we're comparing to before each comparison (as otherwise we'll lose it during the stack operation). In other terms, the value we compare to during each `case` block is stored on the stack until the `switch` statement is done.

Semantically:

- After `case` expression comparisons, we either jump to the case body or keep going to the next `case` comparison or `default` block. This separates `case` 'headers' from `case` 'bodies', allowing all comparisons to be followed in the correct order with only the minimum required jumps.
- To automatically break, for each case statement, we add a `goto` to a label at the end of the switch statement.
- `default` statement jumps are placed after all jumps to case bodies as the fall-through case, when no other jumps are followed.

- Simulating new scopes is easy because of how our scoping works; the variable names are already resolved to their correct local's index in the resource AST and will not conflict during code generation.
- Expressions in `case` blocks not being constants does not matter too much for us (versus a target language like C), as we compare each expression normally (we are simulating switch statements using `goto` and comparisons, and aren't limited by any language-native `switch` statement definitions).

8.2.3 Assignments

There are multiple tricky things about assignments:

- Assignment operators. We cannot just convert `e += e2` to `e = e + e2`, where `e` is an expression, because `e` might contain a function call with side-effects, which we do not want to call twice (note that in some cases, the assignment operator has an equivalent bytecode instruction, i.e. incrementing and decrementing using `iinc`. However, we generalize in this discussion as most operators do not have an equivalent instruction to operate and assign at the same time). There are thus several cases for `e`:
  - `e` is just an identifier. Then, we can just convert `e += e2` to `e = e + e2`, as there will be no side effects.
  - `e` is a selector. If `e` is an addressable selector, then it is not operating on the direct/anonymous return value of a function call and so re-evaluating `e` will not produce any side effects. Thus we can do `e = e + e2` again.
  - `e` is an index, say `e3[e4]`. In this case, `e3` can be an anonymous `slice` from a function return, and `e4` could also be an anonymous `int` from a function return. In order to avoid duplicate side effects, we resolve `e3[e4]` first, including any function calls, to a value, storing the result on the stack. Then, we can operat on the stack, adding `e2` and assigning the result to whatever addressable is referenced by the assignment using the pre-calculated stack value.
  - The other cases for `e` are not `l-values`, and shouldn't happen in the type-checked AST.
- Assignment of multiple expressions. As mentioned earlier, we cannot do the assignments sequentially. Thus, we should evaluate the entire RHS, pushing each result onto the stack and then assigning each stack element one by one to their respective LHS expression l-value. This way, `a, b = b, a` will not overwrite or interfere with any values used on the RHS. This is one of the advantages of using a stack-based language, as values on the stack implicitly act like temporary variables, so we don't need to allocate other temporary resources for simultaneous assignment.
- Copying of slice headers, arrays, and structs. This proved exceptionally difficult to get correct, since we needed to ensure correct behaviour for slices and arrays, and generate copy methods on the fly for structs. See the section below for more on copying and struct generation.

8.2.4 Struct Generation

Struct definitions are collected and de-duplicated in the resource AST. To represent them in the JVM, we convert them to classes, renaming fields to avoid naming conflicts with reserved bytecode keywords. This mapping works well, but we also need equality and copying mechanisms, which proved more difficult to create.

To handle equality, we define an `s.equals(s2)` method on each struct `s` to compare it to another struct of the same type, returning a boolean. This compares each field of the two structs, using other `equals()` methods defined by us in order to compare arrays and slices (which do their own recursive comparisons internally).

8.2.5 Copying Behaviour

As mentioned, GoLite requires pass-by-value execution of methods and copying upon assignment, which differs from the default behaviour of the JVM. For primitives and strings, this is straightforward, since primitives are pass-by-value within the JVM, and strings are immutable in both languages.

Where it gets more complex is with arrays, slices, and structs. For structs, we defined a custom copy method on the classes generated for each struct definition. This method returns a new struct object, containing values copied from the current instance. For slices, arrays, and nested structs, a copy method is called.

For slices and arrays, it was noted that there was no existing Java container type that provides the behaviour we were looking for. As a result, we defined our own class (implemented in Java), called `GlcArray` (see 8.4.3), which can represent single- or multi-dimensional slices or arrays (and any nested combination of those data structures).

8.3 AST

In the codegen stage, we decided to create yet another AST layer between the previous checked AST and the bytecode IR. The primary reason was that bytecode conversion alone is quite complex, and we want to keep state related changes as simple as possible.

With the addition of pretty printing, it will also be a lot easier for us to debug changes in go code than in bytecode. Implementing pretty printing involved creating a converter from each AST to the previous iteration. Once we get to the original AST, we can call its prettify function. We use this method to avoid duplicate prettify rules. As we print more complex code, we can update the base implementation, and have it fixed for all AST levels. Mapping changes between adjacent ASTs is also much simpler, as there are few changes between each stage.

For our resource AST, we wanted to address the following concerns:

- Local variables are labelled using offsets rather than identifiers.
- Stack heights should be minimized; indices should be reused where possible.
- Path splits should have unique label ids.
- All used type declarations should be mapped to a single top level list of structs. Each struct also requires a unique id for the classname.
- Within type definitions for expressions, structs require the unique id instead of the actual content information.
- All init functions need to be renamed into unique function names, then called in the same order from a parent init function.
- Empty for loop conditions and switch conditions should instead provide default fallbacks (see 8.2.2).

To do so, we defined our new AST models, then created another builder that is very similar to the symbol table. As there are no extensive rules, and it is mainly used for hashtables, there is only one module, instead of a core and more specific implementation.

8.4 Java Utils

To make things simpler, we offload some utility functions and classes to our 'glcutils' module. It is written in Java, which means that we don't have to write the bytecode ourselves, and that we can test it using JUnit.

8.4.1 Utils

Our core utility provides the following functions:

`String boolStr(boolean)` Helper to convert booleans to strings "true" and "false"

`<T> T copy(Object), Object copyObj(Object)` Helper to clone objects. If it implements our interface `GlcCopy` (see 8.4.2), it will call `copy()`. Otherwise it will return the original object. Given that structs, arrays, and slices implement copy, other types are primitives or strings, which are all immutable already. We supply the generic variant to make it easier to use in other Java utility classes.

`<T> T supply(Class<? extends T), Object supply(Class)` Helper to create new objects. If the class maps to a primitive or string, we return the default object. Otherwise, we call the empty constructor. Note that this cannot be used for `GlcArray`, as we require size information. We supply the generic variant to make it easier to use in other Java utility classes.

We also supply our own runtime exception class, namely for testing and separating our exceptions from generic ones.

8.4.2 GlcCopy

There were a few options for us to support copying:

- Implement Java's cloneable. All objects already have an unimplemented `clone()` function, but this becomes unnecessarily complicated to generate for structs.
- Generating a `copy()` function for all data types, returning the same class. While this is what the function is meant to do, it will require us to use reflection to call the function.
- Implement our own interface supporting `Object copy()`. This allows us to avoid reflection in our utilities.

We ended up using the last option, and we implement `GlcCopy` for all structs and `GlcArray`.

It is worth noting that while we can override the function and return a superclass (class more specific than `Object`), the underlying bytecode creates a "synthetic bridge" to support it. We wanted to avoid that, which is why all implementations still returns `Object`. We instead rely on unchecked casting to get the appropriate type.

8.4.3 GlcArray

For our array and slice implementations, we decided to create them using a common structure. 'GlcArray' serves as a multi array container, where subsequent dimensions and values are loaded lazily. This means that we can support indefinitely large arrays and slices, so long as the number of indices accessed does not exceed our allotted memory.

A `GlcArray` takes in an `[]int` depicting sizes, as well as a `Class` to create values. For slices, we simply require that the size value be less than 0. For instance, to create `[3][][5]string`, we'd call `new GlcArray(String.class, new int[]{3, -1, 5})`.

We enable usage of `append(...)` depending on whether the current dimension is a slice or not. When building `append(...)`, we also consider the conditions for copying arrays; it is only copied if the capacity changes, and we do a deep copy to ensure that the previous slice remains untouched.

To support laziness, we expose getter and setter functions. Setter simply sets a value at the appropriate index. Getter will initialize the value if it is not yet defined, then return it. Only for string arrays and slices will the value be nullable.

8.4.4 Slice Templates

As a proof of concept, we have Slice implementations in Java under our test folder. We again wanted to support lazy initialization, and borrow concepts from `GlcArray`. Namely, each field has a getter and setter, where the getter will initialize a value if it isn't initialized already. Primitive fields remained primitive, so they come preinitialized.

8.5 Lazy Equality

To support lazy struct equality, we had to do the following:

- For primitive fields, simply do a comparison
- For string fields, call `Object.equals(s1, s2)`. In other words, the fields are equal if they are both null, or both the same string.
- For other fields, check if they are both null, or compare the values after calling the getter. If both are null, we know that they result in the default, and are equals. We do not initialize to remain lazy. If any are non null, we must now ensure that both are initialized before comparing. We can directly use `Object.equals(Object)`, since the getter will always return nonnull values.

For `GlcArray`, it is very similar, but we do a comparison of the size definition, the class type, and then the underlying array data. If the underlying array itself is null, we know that the `GlcArray`s are equal.

8.6 Testing

Unlike previous testing, which was done in Haskell using `hspec`, we wrote complete programs as files to test code generation.

The main reason for changing the testing approach for this phase was that implementing automatic execution of other scripts (the Krakatau assembler) and checking standard output using Haskell was undesirable (introducing IO and outside programs in the tests). Additionally, the script used to run program solutions was already available to us, which made it relatively easy to implement automated testing.

We used the script provided for evaluating previous milestones to run and check the output of all of these programs at once. All test programs are located in the `programs/3-semantics+codegen/valid` and `programs/3-benchmarks+codegen/valid` directories within the repository in order to be automatically discovered by the script.

We also ran the tests in Travis automatically upon every push to GitHub, so we would be alerted if there were any regressions in program compilation or output as a result of a change.

We tried to comprehensively cover all aspects of GoLite mentioned in the lecture given on GoLite code generation. We also added standard functionality tests (such as function execution, variable assignment, etc.) and added tests to cover any bugs discovered during the development of Milestone 4.

Unfortunately, our tests did not completely cover all aspects of our code generation, as multiple bugs were discovered when the evaluation solutions were released. For the final submission of the compiler, concurrent with the submission of this report, we used the bugs discovered by running the evaluation script on the compiler to guide the creation of new test programs, increasing our test coverage of code generation.

For a complete list of code generation test programs and their functions, see Appendix A.

9 Conclusion

9.1 Experience

Overall, this was a valuable project and one of the biggest programming projects we have ever worked on. Our tool choices had a large learning curve, which affected speed of development at some points, but the benefits of using a strongly-typed functional language for development and a low-level but well-documented language like JVM bytecode as a target, especially in terms of learning experience, definitely outweigh the negatives. We are pleased to have created a functional compiler, while learning new languages in the process.

9.2 What We'd Change

While Haskell has allowed us to easily define types, certain restrictions still allow for invalid AST structures. For instance, we can only get the length of an array or a slice, yet the AST support all expressions. To properly restrict all types, we would need to further group data into multiple subsets of constructors[12]. While this would catch more problems at compile time, it will also make the code significantly more complex. The other approach is to allow errors at all stages of the compiler, even though all valid typechecked code has a code generation counterpart. This adds multiple catches of the same error at runtime, but keeps the code short. Our current approach is to 'throw' errors when they occur, without providing a function signature that indicates errors can occur. This allows us to keep our code short, but is problematic with evolution, as users would expect a pure function to be pure (void of errors).

Currently, `GlcArray` is built on top of an object array, so all primitives are wrapped and unwrapped. We can easily provide variants for each primitive type to avoid this. Note that to support multi dimensional content, each primitive `GlcArray` requires both an object array and a primitive array. Only one will be initialized, depending on its depth. This added complexity was the primary reason why we didn't support it in our first iteration.

10 Contributions

- **Julian Lore**
  - M1: Wrote the majority of the scanner and handled weird cases, wrote a large amount of valid/invalid programs, implemented many other tests (`hspec` or small tests in our program) and looked over the parser, contributing a few things to it as well.
  - M2: Implemented weeding of blank identifiers, symbol table generation, typecheck (aside from type inference and expression typechecking) and submitted invalid programs.
  - M3/M4: Implemented the conversion from our bytecode intermediate representation to Krakatau[13] compatible bytecode using `ByteString`s for increased efficiency of string manipulation. Wrote the dynamic creation of IR bytecode classes with equality checks, getters and setters for structs based off Allan's struct template and some other bytecode functions corresponding to each generated file (like `<clinit>`). Add basic implementation of slice in Java that was built upon and vastly improved by Allan. Created the M3 submitted tests, most of (44/56) of the codegen tests and all the submitted benchmark

tests. Helped debug why tests weren't codegenning properly. Added Docker to the continuous integration so that it could assemble our generated programs with Krakatau[13] (required Python) and then run assembled classes (required Java).

- **David Lougheed**
  - M1: Wrote the bulk of the parser grammar and contributed to the weeder. Also wrote 3 of the valid programs and 8 of the invalid ones and had minor contributions to miscellaneous other components. Contributed to the testing of the parser and pretty printer.
  - M2: Worked on expression type-checking and type inference, including tests. Also worked on the weeding pass for return statements.
  - M3/M4: Created most of the intermediate representation used to model JVM bytecode in Haskell. Wrote many of the patterns used for IR (and thus bytecode) generation, especially base patterns such as statements, expressions, and control structures. These were generated from the resource AST described.
- **Allan Wang**
  - M1: Created the AST and helper classes for pretty printing and error handling. Wrote the base package for testing as well as some of the embedded test cases within `hspec`. Added integrations (Travis + Slack), and gave code reviews to the other components.
  - M2: Added data structures for error messages, and supported explicit error checking in tests. Created the data model for symbol table core. Added hspec tests.
  - M3/M4: Created resource AST, which replaces local identifiers with stack indices, collects struct definitions, and reorganizes init functions. Created GlcArray and struct template in Java, supporting lazy loading and equality checking. Added maven dependencies, and created junit tests. Contributed to go tests.

References

[1] Alex. `https://github.com/simonmar/alex`, 2019.

[2] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, Lars Birkedal. A Logical Relation for Monadic Encapsulation of State. In *Proceedings of the ACM on Programming Languages*, volume 2, pages 64:1–64:28, January 2018.

[3] Control.Monad.ST. `https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html`.

[4] Control.Monad.ST.runST. `https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad-ST.html#v:runST`.

[5] The Go Programming Language.

[6] The Go Programming Language Specification - IdentifierList. `https://golang.org/ref/spec#IdentifierList`, Nov 2018.

[7] The Go Programming Language Specification - Semicolons. `https://golang.org/ref/spec#Semicolons`, Nov 2018.

[8] Happy. `https://github.com/simonmar/happy`, 2018.

[9] Haskell Stack. `https://www.haskellstack.org`, 2019.

[10] Hspec. `https://github.com/hspec/hspec`, 2019.

[11] Megaparsec. `https://github.com/mrkkrp/megaparsec`, 2019.

[12] Restrict Pattern Matching to Subset of Constructors. `https://stackoverflow.com/questions/55556060/restrict-pattern-matching-to-subset-of-constructors`, 2019.

[13] Storyyeller. Krakatau: Java decompiler, assembler, and disassembler. `https://github.com/Storyyeller/Krakatau`, 2018.

[14] Travis CI. `https://travis-ci.com`, 2019.

[15] Vincent Foley-Bourgon. COMP-520 - GoLite project. `https://www.cs.mcgill.ca/~cs520/2019/slides/10-golite.pdf`, 2019.

[16] Vincent Foley-Bourgon. COMP-520 - GoLite Type Checking Specification. `https://www.cs.mcgill.ca/~cs520/2019/project/Milestone2_Specifications.pdf`, 2019.

A Test Programs

The majority of our tests (with the exception of code generation tests) were done in Haskell using `hspec`. However, we also submitted many standalone test programs, which are summarized below.

A.1 Scanning and Parsing

Invalid programs:

Scanner

1. `unclosed-string.go`: Literal string that does not have a closing quotation.
2. `unclosed-block-cmt.go`: Block comment with no end.
3. `string-no-char-esc.go`: Trying to escape a single quote inside of an interpreted string.

Parser

1. `type-alias.go`: Trying to define a type alias, which is not supported by `GoLite`.
2. `no-semi.go`: Statement without a semicolon or newline.
3. `no-prefix.go`: Prefix increment, not a Go construct.
4. `no-package.go`: Program without a package declaration.
5. `no-nest-blk-cmt.go`: Trying to nest block comments. This fails at the parser because `*/` are valid tokens, but their usage will not make sense at the parser phase.
6. `no-main-parens.go`: Trying to declare a function (main) without arguments/parentheses for arguments (or lack thereof).
7. `no-comma.go`: Multiple variable declaration with assignment with no commas.
8. `nested-func.go`: Trying to declare a function inside another.
9. `multiple-pks.go`: Attempt to declare multiple packages.
10. `multiple-defaults.go`: Multiple default case statements.
11. `len-args.go`: Multiple arguments to `len` function (remember that len is a keyword and it can only take in one argument as per the grammar).
12. `invalid-var-decl.go`: Trying to declare a variable without the `var` keyword.
13. `invalid-unary.go`: Trying to use `/` as a unary operator.
14. `invalid-short-decl.go`: Short declaration outside of a function.
15. `invalid-fallthrough.go`: `fallthrough` not supported by GoLite.
16. `invalid-continue.go`: `continue` used outside of a loop.
17. `invalid-break.go`: `break` used outside of a for loop or switch statement.
18. `incr-expr.go`: Trying to use an increment as an expression.
19. `func-decl.go`: Function declaration without the types for the arguments.
20. `for-no-block.go`: For loop with no block/braces.
21. `double-semicolon.go`: Two semicolons at the end of a statement.
22. `complex-stmt-in-if.go`: Non simple statement as a pre-statement of an if.
23. `bad-rune-escape.go`: Invalid escape sequence.
24. `bad-raw-string.go`: Trying to escape a backtick in a raw string.
25. `bad-expr-stmt.go`: Expression statement that isn't a function call.
26. `bad-assign-list.go`: Assigning one value to two variables (different size expression lists).
27. `append-args.go`: Only one argument to `append`, grammar requires two (`append` is a keyword).

Valid programs:

1. `matrix_mult.go`: A valid program that multiplies a 1000×1000 pseudo-randomly generated matrix by itself 10 times using the naive $O(n^3)$ algorithm.

The other valid programs were copied to the `3-semantics+typecheck` as they output information that could be verified after running what was code generated.

A.2 Type Checking

Summary of the check in each invalid program:

1. `append-diff-type.go`: Append an expression of a different type than the type of the expressions of the `slice`.
2. `append-no-slice.go`: Append to something that isn't a slice.
3. `assign-no-decl.go`: Assign to a variable that hasn't been declared.
4. `assign-non-addressable.go`: Assign to a LHS that is a non-addressable field.
5. `cast-not-base.go`: Cast to a type that isn't a base type.
6. `dec-non-lval.go`: Decrement something that isn't an `lvalue`.
7. `decl-type-mismatch.go`: Declare and assign variable of explicit type to an expression of a different type.
8. `float-to-string.go`: Try to cast a `float` to a `string`.
9. `for-no-bool.go`: While variant of for loop with a condition that isn't a bool.
10. `func-call.go`: Function call with arguments of different type than function declaration arguments.
11. `func-no-decl.go`: Calling a function that hasn't been declared.
12. `function-already-declared.go`: Trying to declare a function that has already been declared.
13. `function-duplicate-param.go`: Trying to declare function with two params with same name.
14. `if-bad-init.go`: If with an init statement that does not typecheck (assignment of different type).
15. `inc-non-numeric.go`: Increment an expression that doesn't resolve to a numeric base type.
16. `index-not-list.go`: Index into something that isn't a slice.
17. `index.go`: Index that does not resolve to an int.
18. `invalid-type-decl.go`: Declare a type mapping to a type that doesn't exist.
19. `no-field.go`: Using selector operator on struct that doesn't have the field requested.
20. `non-existent-assign.go`: Assigning a variable to a non existent variable.
21. `non-existent-decl.go`: Trying to declare a variable of a type that doesn't exist.
22. `op-assign.go`: Op-assignment where variable and expression are not compatible with operator (i.e. `int + string`)
23. `print-non-base.go`: Trying to print a non base type.
24. `return-expr.go`: Returning an expression of different type than the return type of the function.
25. `return.go`: Return nothing from non-void function.
26. `short-decl-all-decl.go`: Short declaration where all variables on LHS are already declared.
27. `short-decl-diff-type.go`: Short declaration where already defined variables on LHS are not the same type as assigned expression.
28. `switch-diff-type.go`: Type of expression of case is different from switch expression type.
29. `type-already-declared.go`: Trying to define a type mapping to a type that already exists.
30. `var-already-declared.go`: Trying to declare a variable that is already declared.

A.3 Benchmarks and Code Generation

1. `color.go`: Get the optimal minimum number of colors to color certain graphs using the brute force $O(n^n)$ optimal solution (in order to take a long time without using a heuristic/approximation).
2. `knapsack.go`: Optimally solve the knapsack problem using dynamic programming, $O(nW)$, with $W = 12983$ and $n = 4361$.
3. `sudoku.go`: Solve a 16×16 sudoku grid using brute force with backtracking. Certain rows are left empty, allowing multiple solutions (as a problem with only one optimal solution would take too long to solve for a 16×16 and too short to solve for a 9×9).

A.4 Semantics and Code Generation

1. `append.go`: Appending to slices.
2. `append_multi.go`: Slices' underlying array-copying behaviour.
3. `append_multi2.go`: Same as above.
4. `appendbounds.go`: Bounds test for slices after appending.
5. `array.go`: Miscellaneous array behaviour and operation tests.
6. `arraybounds.go`: Array bounds test (negative index).
7. `arraybounds2.go`: Array bounds test (too-large index).
8. `assign.go`: Assignment test, including multiple assignments and operator-based assignments.
9. `assign2.go`: Array, slice, and struct assignments.
10. `basic.go`: Basic program which prints a number.
11. `binary.go`: Test for most binary expressions, with different operand types.
12. `binary_search.go`: Binary search on a size-10 array.
13. `blank_identifiers.go`: Miscellaneous basic blank identifier tests.
14. `cast.go`: Casting and print-formatting tests.
15. `closest_pair.go`: Large program to pseudo-randomly generate points and find the closest pair.
16. `for.go`: Different for-loop formats and behaviours.
17. `func_calls.go`: Function calls, including tests for correct execution count in case statements and short-circuiting of boolean logic.
18. `funccalls.go`: Function call tests with data structures (arrays, slices, and structs).
19. `funcret.go`: Function call returns, casting, blank identifiers, and print formatting.
20. `global_fields.go`: Default values for global variables (fields). **(Added for resubmission, post bug-discovery)**
21. `global_var_assignment.go`: Assignment of a local variable to a global variables with the same name. **(Added for resubmission, post bug-discovery)**
22. `ifscoping.go`: Scoping of variables within if/else statements, including short statement declarations.
23. `increment.go`: Increment and decrement operators on various addressable values.
24. `init.go`: `init` function tests.
25. `init2.go`: Same as above.
26. `init3.go`: Same as above.
27. `init4.go`: Same as above.
28. `init5.go`: Same as above.
29. `init_order.go`: `init` function execution order test.
30. `len.go`: Length built-in function test on the 3 supported data types (array, slice, and string).

31. `multi_array_string.go`: Nested array capacity and string printing test.
32. `needleman_wunsch.go`: Implementation of the Needleman-Wunsch algorithm for DNA alignment.
33. `nested_structs.go`: Nested struct/slice copying and modification behaviour.
34. `ntuple.go`: Large program defining a mini-library for n-tuple operations, built on slices.
35. `print.go`: Print formatting test.
36. `printfloat.go`: Float-printing test.
37. `prints.go`: Print formatting test.
38. `reserved_java.go`: Test for function names that might conflict with Java (JVM) keywords, since we were targeting JVM bytecode for code generation.
39. `return.go`: Test for return-value behaviour, including copying.
40. `rpn_calc.go`: Reverse-Polish notation calculator which takes in an array representing an expression.
41. `rune_esc.go`: Test for escape characters, including some ones requiring transformation (`\a`, `\v`) that don't exist verbatim in the JVM. **(Added for resubmission, post bug-discovery)**
42. `scope.go`: Miscellaneous scope tests.
43. `scope_same_var.go`: Scope tests for variables with the same identifier.
44. `shortdecl.go`: Small short declaration test, including blank identifiers.
45. `slice_copy.go`: Assignment slice-copying test.
46. `slicebounds.go`: Bounds test for empty slices.
47. `slicebounds2.go`: Negative-index slice bounds test.
48. `slicebounds3.go`: Too-large index slice bounds test.
49. `structeq.go`: Struct equality tests.
50. `switch.go`: Switch behaviour tests.
51. `switch2.go`: Switch behaviour tests with function-call expressions.
52. `unary.go`: Comprehensive unary operator usage test.
53. `vardecl.go`: Test of new variable declaration from function returns.
54. `vardeclblank.go`: Blank identifier variable declaration test.
55. `vardeclscoping.go`: Variable declaration scoping test.
56. `vardeclshadow.go`: Overshadowing test (of GoLite pre-provided values.)