Report

Team 9

Aleksi Sapon-Cousineau - 260581670
Rohit Verma - 260710711
Ayesha Krishnamurthy - 260559114

Introduction

Go is a simple language inspired by C. While opinions on it can vary quite a lot (source: every reddit.com/r/programming thread on Go), it is still very popular and has been adopted by many companies for their products. Most importantly, Google, who have been sponsoring the language since its inception. Golite is mostly a subset of Go, which is focused on its C-like elements (types, expressions and statements). With the exception of the garbage collector, it is rather simple to implement. Yet it has a few twists, making it a good choice for a compiler design project. The Go specification and playground are also rather well made and accessible.

In this document, we will discuss our project design, from our implementation tools to the syntax, semantic and code generation phases. We will discuss the architecture and the various challenges we faced.

Language and tools

We chose SableCC over Flex and Bison as our "compiler-compiler" as we had worked with SableCC for the individual assignments. We were aware that SableCC would limit us to using a JVM language, however all team members were comfortable with using Java and we were willing to trade off the efficiency that would have been gained using C or other lower level languages for the convenience of being able to get up and running quicker.

We chose to use Java 8 as it is now widely supported and offers a lot of useful features - for instance, we used the `forEach()` method extensively to iterate over lists of nodes in our Abstract Syntax Tree.

For a build system we opted to use Gradle. It's a task based tool designed original for Java. It's fast

compared to other ones like Maven, and easily extensible. It also manages dependencies for us, such as the Apache CLI library and the LLVM 3.9 bindings. We made a custom plugin to include the SableCC grammar processing as a build task. Through the configuration code, we also added more tasks to compile the runtime, and generate bash files for running our compiler and compiling the code generation output. It runs our tests on every build too. With it we've automated nearly every build step in the project.

Gradle doesn't need to be installed on the machine either, thanks to the Gradle Wrapper.

Syntax

SableCC works by taking in a specification file (ending in .sablecc). The specification file is in the Backus-Naur Form, which is widely used for defining exact descriptions of languages. Our specification file contains 5 main sections: Helpers, Tokens, Ignored Tokens, Productions, and Abstract Syntax Tree.

The "Helpers" section contains strings that allow for a more compact definition of tokens - e.g. if multiple tokens contain an underscore, it helps to define an `underscore` helper instead of adding `'_'` in multiple places.

The "Tokens" section then defines all the tokens that the scanner will match the input program with. "Ignored Tokens" specifies the tokens that will be ignored by the compiler. "Productions" specify the syntactic structure that the input program should have. For each production and AST node, every alternate should be preceded by a name in curly braces so that the node classes generated by SableCC all have distinct names.

Finally, the AST section consists of the AST nodes to be generated. The transformations from CST nodes to AST nodes is specified for each CST node defined in the 'Productions' section.

Lexing

We wrote a specification file for GoLite, using the syntax described above. After compilation of the file, SableCC outputs the lexer and parser for our grammar. The lexer simply ensures that the input program strings match tokens that were in our 'Tokens' section, and ignores things that match our 'Ignored Tokens' which consist of comments and white space tokens.

Parsing

We use CST to AST transformations in the "Productions" section to generate AST nodes from the input tokens during the parsing stage. This conversion was essential as it greatly simplified the work to be done in the later stages of our project. For instance, we needed to create 10 expression CST nodes types in order to avoid shift/reduce conflicts, which as we'd seen in class can be solved by the introduction of intermediary nodes (e.g. using terms, factors, and expressions instead of a single expression type). CST expression types are all converted to a single AST expression node type, which only requires information about the expression operator (addition, subtraction, etc.) and the values or identifiers in the expressions.

We followed the Go syntax specification quite closely. ASTs begin with the `prog` node which consists of the package name followed by declarations of variables, types, and functions. Statement nodes are children of functions, and expressions are children of statement nodes. The only case in which we deviated from the specification was for short variable declarations. In order to prevent a shift/reduce conflict, we chose not to limit the left-hand side to a list of identifiers, and instead used a list of expressions which we could weed out at a later stage.

Weeding

Our `Weeder` class deals with some of the problems that would have been either inconvenient or impossible to fix within our grammar file. It checks variable declarations for unique variable names, and that the number of elements on the left-hand side is equal to that of the right-hand side. Similarly for function declarations, it checks that the parameter names are unique. Uniqueness for variable and parameter names is checked using a `HashSet`. We also ensure that the blank identifier is used in the right places by checking the parent node of the blank identifier. We make sure that the parent is allowed to contain a blank identifier, as specified in the GoLite specification.

It is also worth noting that `Weeder` extends `DepthFirstAdapter` and we override the "in" and "out" functions only for the nodes that we need to inspect.

The weeder keeps track of what scope statements are in. This is done using a Scope enum, which can have the values of `TOP`, `FUNC`, `IF`, `FOR` or `SWITCH`. When a program node is encountered, the `TOP` scope is pushed onto our scope stack and popped once it has been exited, and the same is done for all functions and "if", "for" and "switch" statements. This is used when checking if `break` and `continue` occur within the correct scopes.

The weeder also checks for multiple default cases, and that statement expressions are function calls only.

Semantics

Types

The `Type` class has abstract methods for classification (numeric, integer, comparable and ordered), and an abstract method for the "resolve" operation. For each type, `equals()` and `toString()` are implemented. The latter prints types in the Go format, which is useful for debugging and error messages.

We have seven implementations of this abstract class:

1. `BasicType` - the Go basic types are singletons of this class. The private constructor of this class takes in the name of the Go type. The resolve method simply returns the `BasicType` object itself. In addition to methods inherited from `Type`, methods for casting have been added. These methods ensure that the cast is to `int`, `rune`, `float` or `bool` and that casting is done from one of these types, or is an identity cast.

2. `IndexableType` is the parent of the `ArrayType` and `SliceType`. It simplifies type-checking for operations performed on arrays and slices. It has a constructor that takes in the `Type` of the component, and a `getComponentType()` method.

3. `ArrayType` implements `IndexableType`. In addition to the inherited methods, it stores the length of the array.

4. `SliceType` implements `IndexableType` as well. It implements the inherited methods.

5. `StructType` is an implementation of type that stores its fields as a list of `StructType.Field`. The inner class `Field` stores the `Type` and name of each field.

6. The `FunctionType` is an implementation of `Type` that stores the parameters and return type of the function. It is used as the type of identifier expressions that reference functions, which is necessary since Go directly calls values. We make use of the Java 8 `Optional` class here to wrap the potentially null return type. This is a good way of ensuring safety.

7. `AliasType` is a name and reference to any other `Type`. It is the only type that doesn't return itself in the implementation of `resolve()`. Instead it unwraps the aliased type of any further aliases, by calling `resolve()` on its inner type. This is done recursively until the type returned is no longer an `AliasType`. This type also needs the ID of the context in which it was declared, along with the name of the alias. This is because aliases are only unique within different scopes.

If two alias objects are created for the same symbol, then they should be equal because it is the same name and scope. Relying on `==` would always return `false`, but comparing the names would return true for aliases in different scopes. So the context ID is used to properly check for equality.

This hierarchy of type classes allows for `resolve()` to be used effectively by our type checker and provides only the relevant information for each kind of type.

Symbols

All symbols implement the abstract `Symbol` class. They all have a name, a type and source position information, which helps with providing better error messages. To obtain the source position information, a utility interface `SourcePositioned` was created. `Symbol` implements this interface, which provides helper methods for printing out position information.

We have three kinds of symbols:

1. `DeclaredType` is used for type declarations. While type declarations are normally always aliases, this isn't the case for the pre-declared basic types in the universal context. For this reason, the class will accept any kind of type. When an alias type is used, it ignores the alias and uses the type being aliased. This was done to support the special case of pre-declared types.

2. `Variable` stores the name and type of a variable.

3. `Function` stores the name, the `FunctionType` of the function, and contains a list of the `Variable` parameters.

These classes provide all the information needed for our symbol table, which is implemented through our `Context` system.

Contexts

The `Context` abstract class is basically the symbol table. It stores a map of names to `Symbols`. Every context is assigned a serial ID, which is used both for uniqueness and for debugging. Every context except the `UniverseContext` has a parent. We can recursively search through parent contexts for a given symbol, which is used to implement the block shadowing rule, in which a symbol declared in an inner context shadows one that is declared in a context that's higher up in the hierarchy. We can also declare a symbol in a context using `declareSymbol()`. This simply adds it to

the map, after checking for duplicates. If the identifier is blank, then nothing is added (as per the Go specification). When a return statement is encountered, we need the type of the enclosing function. We added a method for this purpose, which traverses the context hierarchy until a `FunctionContext` is found. The remaining methods are used to print and convert contexts to strings, for debugging and for the `-dumpsymtab` option.

We have four kinds of contexts that all implement `Context`:

1. `UniverseContext` encloses all GoLite code by default. It always has 0 as its ID and does not have a parent. It is used as a singleton. The pre-declared variables and identifiers are always present, and the `declareSymbol()` method is overridden to always throw an exception.

2. `TopLevelContext` is the next context. Its parent is always the `UniverseContext` and its ID is also always 1.

3. `FunctionContext` always has `TopLevelContext` as its parent. It doesn't allow the declaration of nested functions. It also stores the `Function` symbol for the declaration, to be returned when looking up an enclosing function.

4. `CodeBlockContext` always has either a `FunctionContext` or another `CodeBlockContext` as its parent. It's used for everything else: "if-else", "for" and "switch" statements and cases. The kind data is stored for better debug information.

Type checking

The `TypeChecker` class extends `AnalysisAdapter` and overrides methods for visiting each node of our AST. The type checker contains `HashMap`s that map nodes in our AST to our custom `Type` and `Symbol` classes, which are used and updated by the visitor methods. Below is an overview of the implementation of each section of `TypeChecker`.

Declarations

Type declarations create a new `DeclaredType` in the context. The type is an alias of the declared one, using the name of the declaration. This is declared in the current context.

Variable declarations are an implementation of the specification. When no type is given, we use the type of the given value, otherwise it is the same as the given type. The variables are declared in the context, which will throw an exception if the name has already been used.

Function declarations open a new `FunctionContext`, then declares the parameters as variables in the context. After the function body has been type-checked, the function context is closed. After type-checking the body, it is checked for a terminating statement at the end. This is done with a separate `AnalysisAdapter` implementation in `TerminatingStmt` that ensures that functions return on all paths by visiting all function statements.

Short variable declarations result in at least one new identifier being declared in the current context. The type-checker ensures that at least one identifier on the left-hand side has not been previously declared by looking the identifier up in the current context's symbols. If this condition is met and all of the expressions on the right-hand side are well-typed then all previously undeclared identifiers are declared in the current context.

Statements

Empty statements, `continue` and `break` are trivially well-tpyed and are handled by overriding the visitor methods for these nodes with empty methods.

Return statements with an expression result in the expression being type-checked and its type being compared with that of the enclosing funciton type. If the types are the same, then the return statement is well-typed. If no return expression is given, a check is done to ensure that the function doesn't have a return type.

Assignment statements are dealt with by ensuring that all the identifiers on the left-hand side have been declared (or are the blank identifier) and type-checking the right-hand side to ensure that all given expressions are valid. Then the identifier and expression lists are traversed to ensure that each identifier-expression pair has the same type.

The `print` and `println` statements result in a type-check of all given expressions to ensure that any `AliasTypes` resolve to a `BasicType`. If no expressions are given, this is trivially well-typed.

Declarations and short declarations are type-checked as per the rules in the 'Declarations' section above.

For-loop statements result in the creation of a `CodeBlockContext` in which the for-loop initialization condition and/or statements are placed. Thus if any new variables are created during initialization or if variable values are updated, the changes will be reflected within the `CodeBlockContext` for the loop body. The initialization statements and expressions are then type-checked and we check that the loop condition (if given) resolves to `boolean`. A `CodeBlockContext` is then created for the loop body and all the statements within the loop body are type-checked. Once this is done, the context

for the "for" body and the outer context are closed.

If-statements result in the creation of `CodeBlockContext` for the initialization, and, if given, the statement and expression are type-checked and a check is done to ensure the condition expression resolves to a `boolean`. A `CodeBlockContext` is then opened for the if-body.

Switch-statements result in the creation of `CodeBlockContext` for the initialization, in which we type-check the initialization statement and/or expression (if given) and the type of the switch expression is stored to ensure that the case expressions have the same type. If no expression is given, we store the type as `boolean`. We then open a `CodeBlockContext` for each case, and ensure that the case expression types are the same as the switch expression type. After type-checking the body of a case, we close the context for the current case and repeat the process for all given cases. Once this is done, we close the outer context.

For op-assignments, we type-check the left and right sides and check that they have the same type, otherwise an exception is thrown. We then check that the operation is valid for the given type as per the rules in the GoLite type-checking specifications.

Expressions

The literal expressions simply return their associated type.

For identifiers, we look up the symbol. It must exist and be either a variable or function, otherwise an exception is thrown. Functions are treated as callable values, but since no other operation supports function types, they can't be used for anything else.

Select expressions type-check if the value is a struct type and has a field of the selected name. If it does then the field type is returned, otherwise an exception is thrown.

Index expressions check that the value is an array or slice (`IndexableType`) and that the index resolves to an `int`.

Calls have a special case for casts. If the value being called is an identifier expression and the identifier references a type symbol, then we type-check it as a cast instead. Otherwise we check that the value has a function type, and that it has a return type. If the arguments are assignable to the parameters, then the expression has the same type as the return type.

For casts, we check that both the symbol and argument (which there should only be one of) resolve to basic types, and that they are valid for casting (we check casting from and to). The type is then the same as the casting type.

For appends, the first argument must be a slice and the second must be the same as the component type.

For unary and binary operators, we check that the resolved type(s) are of the proper category (integer, numeric, etc.) for the operator. Additionally, for binary types, we have to check that the unresolved types are the same. The resulting type depends on the operator.

Types

Named types just return the type of the `DeclaredType` symbol they reference.

For slice types, we wrap the component type in `SliceType`. Same for arrays, but we also include the length, which we obtain by evaluating the integer literal.

For struct types, we create the fields first, then wrap them in a `StructType`.

Other validation

After type-checking the body of a function, it is checked for a terminating statement at the end. This is done with a separate `AnalysisAdapter` implementation in `TerminatingStmtChecker` that ensures that functions return on all paths by visiting all function statements.

When type-checking a program, we first ensure that if the main function exists, it has no parameters and no return type. Once this check is done, we then proceed with type-checking the program.

Code generation

Target language

We have chosen to to target low-level code: LLVM IR. More specifically, we first convert Golite to a simpler custom IR before converting it to LLVM IR using the LLVM C API.

We used LLVM as a backend, because it's used by several popular languages like Rust, Swift, C, C++, D, etc. It can compile to many different targets and offers optimization passes for free. LLVM IR is relatively easy to work with, and due to its popularity is rather well documented and standardized, meaning that if we get stuck somewhere we have documentation to help us.

It also has a proper API to generate it using objects, which is a lot nicer than trying to print a

language like C. With it we can manipulate the IR more freely. Since we're working with the AST directly, things like names don't matter. If there's a conflict, LLVM will take care of generating new names when printing.

It has a major disadvantage: it is comparatively more difficult to implement than higher level languages like C/C++, but we wanted to take on this challenging task and in the process learn more about LLVM.

Using LLVM allows us to get experience with a major framework and toolchain that powers a vast number of compilers used in research and enterprise.

First conversion to custom IR

After type-checking, we run the AST and `SemanticData` produced by our type-checker through our `IrConverter`. The `IrConverter` visits AST nodes and converts them to our custom node types (found in `golite.ir.node`) to facilitate conversion to LLVM IR. The `Program` class contains all the relevant information required for code generation: the package name, global variables, and functions (through which statements can be accessed). The conversion allows for complicated statements to be split up into multiple simpler ones.

The IR also directly integrates the semantic data, meaning that we no longer have to carry it around in a separate `SemanticData` class. It is thus better suited for the final compilation passes.

As an example of our custom IR, consider this simple Golite program:

```
////////////////////////////////////////////////////
/* Golite program to check if a number is a palidrome */
////////////////////////////////////////////////////
package main

func findPalidrome() {

    // Variables
    var rem, sum int
    var temp, num int
    rem, sum = 0, 0

    // Number to be checked for palindrome
    num = 56788765

    // Run the actual loop
    temp = num
    for num > 0 {
```

```
        rem = num % 10 //Remainder
        sum = (sum * 10) + rem
        num = num / 10
    }

    // Print the messages
    if temp == sum {
        println("num is a palidrome")
    } else {
        println("num is NOT a palidrome")
    }
}

func main() {
    // Call the findPalidrome function
    findPalidrome()
}
```

It is converted into the following custom IR:

```
package main

func findPalidrome() {
    var rem int
    assign(rem, 0)
    var sum int
    assign(sum, 0)
    var temp int
    assign(temp, 0)
    var num int
    assign(num, 0)
    var assignTmp int
    assign(assignTmp, 0)
    var assignTmp0 int
    assign(assignTmp0, 0)
    assign(rem, assignTmp)
    assign(sum, assignTmp0)
    assign(num, 56788765)
    assign(temp, num)
    jump(startFor)

startFor:
    jump(endFor, !(num > 0))
    assign(rem, (num % 10))
    assign(sum, ((sum * 10) + rem))
    assign(num, (num / 10))
    jump(continueFor)

continueFor:
```

```
    jump(startFor)

endFor:
    jump(ifCase, (temp == sum))
    jump(elseCase)

ifCase:
    printString("num is a palidrome")
    printString("\0A")
    jump(endIf)

elseCase:
    printString("num is NOT a palidrome")
    printString("\0A")
    jump(endIf)

endIf:
    return
}

func main() {
    call(findPalidrome)
    return
}

init {
    return
}
```

As you can see, a lot of the implicit semantics are made explicit, such as: zero values, temporary variable in multiple assignments, and void returns at the end of functions. All the control flow is re-written as branches and labels. Most statements are simplified into multiple ones, such as one print per expression, and one variable per declaration. Operators are also made type specific; for example comparing integers uses `==`, but strings use `==$`. More complex operations, like struct and array equality are expanded to many lines of code.

All of the non-declaration statements use a new notation in the form of `stmtName(symbols and values)`, which is more verbose but easier to work with.

Type declarations are ignored, and all types and symbols have their aliases removed. This can make things a bit verbose when large structures are used, but it's much simpler to work with. It's difficult to keep structure names because Go doesn't actually have named types, instead it has named aliases to types. We would have to remove all the aliasing layers except the last one on structs, then create a new named structure type, and update all the references to it. This is a lot of error prone work, which we opted not to do.

The last function is a special one, which is called before the `main`, and is used to initialize the global variables. Since the program has no global variables, it is empty. Otherwise it would contain assignments.

If the Golite program does not provide a `main` function, then an empty one is generated.

The IR conversion also prepares the control flow for conversion to LLVM IR by removing any constructs that would violate the validation rules. This mostly means removing code after `return`, `break` and `continue`. The basic blocks are also always terminated with a `return` or an unconditional jump for the same reason.

This is done with a modification pass after the IR conversion. As an example, here is a valid Golite program that does not translate directly to LLVM IR:

```
package main

func abs(i int) int {
    if i < 0 {
        return -i
    } else {
        return i
    }
}

func main() {
    for i := -5; i <= 5; i++ {
        println(abs(i))
    }
}
```

This is the IR without the pass:

```
package main

func abs(i int) {
    jump(ifCase, (i < 0))
    jump(elseCase)

ifCase:
    return -i
    jump(endIf)

elseCase:
    return i
```

```
    jump(endIf)

endIf:
}

func main() {
    var i0 int
    assign(i0, -5)

startFor:
    jump(endFor, !(i0 <= 5))
    printInt(call(abs, i0))
    printString("\0A")
    jump(continueFor)
    printString("skipped")
    printString("\0A")

continueFor:
    assign(i0, (i0 + 1))
    jump(startFor)

endFor:
    return
}

init {
    return
}
```

Notice how the `abs` function has a label without any statements underneath it (an empty basic block). In LLVM this would violate the rule that each basic block must end with a terminator (branch, return or exception handling). Both the `ifCase` and `elseCase` blocks also violate a rule: a terminator must be the last in the block; yet a return appears before a branch. Same issue in the `main`'s `startFor` block. Another less obvious issue is the implicit fall-through from `main`'s entry block to `startFor`, which is invalid due to the lack of terminator.

We fix these issues in five steps:

- First we convert the statement list of a function into basic blocks, which are separated by labels
- Then we add a jump to the next block for any block that doesn't end with a terminator
- Next we remove all the statements that come after the first terminator in a block (unreachable)
- The previous step will make some blocks unreachable by removing jumps, so we now remove them
- Lastly we recombine the blocks into a statement list

This work is done by the `IrFlowSanitizer` class.

Now after the pass our output IR is:

```
package main

func abs(i int) {
    jump(ifCase, (i < 0))
    jump(elseCase)

ifCase:
    return -i

elseCase:
    return i
}

func main() {
    var i0 int
    assign(i0, -5)
    jump(startFor)

startFor:
    jump(endFor, !(i0 <= 5))
    printInt(call(abs, i0))
    printString("\0A")
    jump(continueFor)

continueFor:
    assign(i0, (i0 + 1))
    jump(startFor)

endFor:
    return
}

init {
    return
}
```

As you can see, the `endIf` block was made unreachable and was removed. The unreachable statements in `startFor` were also removed. The `main`'s first block now explicitly jumps to the next.

It might not be obvious how this transformation solves the problem of the `endIf` block. First we have to observe that void-returning functions never have this issue, since they always end with a `return` statement (we make it explicit). Thus there always is a statement at the very end of the

function. For value-returning functions, we have to remember that we validated the return paths in the type-checker. This means that there will always be terminator(s) in the path to `endIf`. Then all we need to do is to remove the unreachable statements until those terminator(s) are the last block(s).

Runtime

There are a few operation in Golite which are rather complicated to implement in LLVM IR directly. Some of them are simply too verbose. Instead we can implement them in C (or any other compiled language) to make our lives easier. During the compilation process, we link the program object file to the runtime one.

In the file `src/main/c/golite_runtime`, we implemented the following functionality: printing to `stdout`, bounds checking, slice appending, string concatenation, and string comparison.

The printing functionality is implemented in C mostly because:

- Boolean are printed as "true" or "false" instead of 0 or 1
- Strings are not null-terminated in Go, which means that we need to print each character at a time using a loop

The other print functions are just there for consistency.

The remaining operations are implemented in C for readability and conciseness.

Notice that the integer types used have exact sizes. This is for better compatibility with LLVM, which also uses strictly defined sizes for integers.

Another important declaration in the runtime is the structure type `goliteRtSlice`. It has a length field and a pointer to a memory buffer. As the name suggests, it is the backing data structure for slices, but is also used for strings. Using this we can implement the pass-by-reference semantics of slices. String are immutable, so the pass-by semantics do not matter. Arrays are implemented as a value type, just like in C.

Finally, at the end of the file are two prototypes: `void staticInit(void)` and `void goliteMain(void)`. These are implemented by the code generator. The first contains code for initializing global variables, and the second is the actual Golite `main` entry point. The last declaration of the runtime is the C `main`, which simply calls `staticInit` followed by `goliteMain`.

Final conversion to LLVM IR

By using a custom IR, we can significantly reduce the number of different nodes we need to code-generate, and also make them more compatible with LLVM IR.

The first step in converting a program is to declare the external functions that are implemented in the runtime. We also declare a named structure type for `goliteRtSlice`.

Next we declare the global variables. Then we can generate the functions one after the other. One important thing here is to prevent the functions in the Golite program for interfering with the pre-defined names for linking with the runtime. This means that we need to ensure that only the `main` function is called `goliteMain`, and that none have the name `staticInit`. We can simply append a '1' to the name, and let LLVM figure out any further naming conflicts.

As we traverse the custom IR, we generate the corresponding LLVM IR. Most of this is rather straight forward, and based off the LLVM documentation.

To code-generate a function, we first declare one basic block per label. This is done to solve the issue of forward references to labels. Then we position an instruction builder at the end of the current label, and append instructions.

The first basic block of the function is reserved for stack allocations. This is because we don't want to to allocate inside loops, since the memory is only freed at the end of the function. Otherwise we'd grow the stack on each loop iteration and leak memory like crazy. With this we can guarantee that memory is allocated only once. The actually function body starts after this block. This the same strategy used by Clang. LLMV will optimize later on the stack memory usage.

For every function parameter, we allocate memory on the stack and copy the value into it. We then save a pointer to the memory corresponding to the variable. This is done so we can assign new values to parameters (which doesn't change the caller's arguments).

Variable declarations simply allocate stack memory and save the pointer to it.

Boolean, integer and float literals are converted to LLVM constant values. String literals are added to a constant pool, then a pointer to the character array is taken and is returned with the string length in a `goliteRtSlice`.

Identifiers are simply converted to a pointer to the variable's memory.

Select expressions get a pointer to the field inside the struct.

Index expressions are a bit more complicated: first we compute a pointer to the value, and the index.

Then we need to find the length of the data. For an array, we just use the type, since it is constant. For slices, we need to access the length field in the struct. Next we call the bounds checking function in the runtime. Finally we can get a pointer in the memory are the index.

Note how the identifier, select and index expressions return pointers instead of values. This is because these expressions are assignable. This means that we need a reference to the memory into which a value should be copied. But we also need to use them as values in many cases. Sometimes we also want to have values as pointers instead (for passing the data to the slice append function for example). To do this we access values and pointers through the `getExprValue()` and `getExprPtr()` methods. These take care of adding the instructions to load a pointer when trying to access it as a value, and to store a value on the stack when trying to access a value as a pointer.

The append expression will compute the size in bytes of the item being appended, then will store it in the stack. It then calls the runtime function with the slice, data pointer and size. The returned value is a slice pointing to a new memory buffer, which contains the original slice data followed by the appended data. This is really a concatenation of a single item.

String concatenation is basically the same, since strings also use slices, and the runtime function for appending can also do concatenation.

String comparison is simply done by calling a runtime function. We pass an integer ID for the kind of comparison, and both slices.

The logical AND and OR are a lot more complicated than the other operators because of short-circuiting. We need to generate two new basic blocks, and insert them in the proper locations. This also implies updating the instruction builder to append at the correct block. The basic idea it to compute the left value first; then branch to the end, or compute the right value then branch to the end; then return the value based on which branch we took. We could do this with a phi node, but it's quite difficult to do when the left or right is a global variable. This is because we need to specify which basic block the value originates from, yet globals don't belong to any. Instead it's much simpler to allocate stack memory and store the intermediate results there. Later we will run optimization passes and let LLVM convert this to a phi instruction.

The remaining expression all translate directly to a single LLVM instruction. With similar names and semantics.

Return statements also translate to a single instruction.

The print statements are simply converted to a call to the runtime, with the value as the argument.

The memset 0 statement is converted to a call to the LLVM memset intrinsic. The first argument is a pointer to the data being cleared. The second is the size to clear. In our case, the data is a variable pointer, and the size is given by the type. There's a trick to compute this size: create a null pointer to the variable type, then treat it as an array and get a pointer to the second element. Now we just convert this to an int. Since the pointer is null, the first index is `0`, and the second is `sizeof(type)`. When optimizing, LLVM will replace these instructions by a constant integer, taking care of calculating the data size with padding for us.

Assignments are simply converted to storing the right side value into the memory pointed by the left side.

Labels were converted to basic blocks earlier. Now when we encounter one, we just have to move the instruction builder to the block, and start appending instruction there.

An unconditional jump is just a branch in LLVM. A conditional one is also a branch, but it's a bit different. Our IR only specifies the `true` destination, the `false` one being the instruction right after the jump. LLVM requires both. This can be solved by adding a new basic block after the jump, using it as the `false` destination.

The very last step is to apply optimization passes to the generated LLVM IR. We picked the following ones: constant propagation, instruction combination, memory-to-register promotion, global value numbering (redundant instruction removal), and control-flow graph simplification. This is a safe and rather basic set of optimizations.

Conclusion

Our experience working on the project was pretty fascinating, and even though most of our work was implemented in Java which we'd all worked with before, we still had plenty to learn. Understanding Go syntax and semantics was part of the challenge, as well learning how to use the LLVM C API. Using LLVM was possibly the biggest "difficult" decision we made, as we were worried we wouldn't get it working in time for the due date, but we pulled through and seeing our code generator work was really cool.

"I guess it went well. It works." - Aleksi Sapon-Cousineau

Contributions

Aleksi Sapon took care of the project lead, architecture and design. He also implemented symbols,

types and contexts, and most of the conversion from the custom IR to LLVM IR. Otherwise he also helped with every other aspect of the compiler. Finally he took care of the overall code quality and consistency.

Ayesha Krishnamurthy contributed to parts of the grammar, weeder, type-checker, IR-generator, and code-generator, with a focus on statements. She also contributed to fixing bugs in all parts of the compiler (that she admittedly sometimes created in the first place) and contributed test cases for milestones.

Rohit Verma contributed at all stages of the project including the scanner, parser, weeder, type-checker, the IR and CodeGenerator. He also thoroughly tested the code at all milestones to ensure the code passed all of the checks and usually discovered bizarre bugs towards the end. Finally he contributed to other parts of the milstones equally with the required artifacts like sample programs for testing.