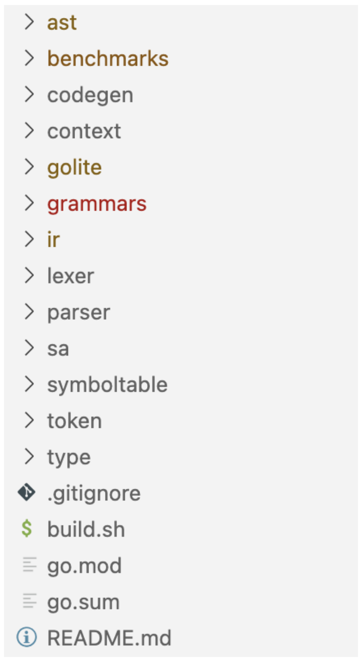
Golite Compiler Design Specification

Xihao Wang

March 10, 2023

1 System Description

This project is a completed compiler including phases of lexing, parsing, static semantics, LLVM intermediate representation and assembly code generation.



```
> ast
> benchmarks
> codegen
> context
> golite
> grammars
> ir
> lexer
> parser
> sa
> symboltable
> token
> type
.gitignore
build.sh
go.mod
go.sum
README.md
```

Figure 1: File Structure

The file structure is shown in Figure 1. A detailed description of some of the folders are:

- ast: Abstract Syntax Tree (AST) implementation.
- benchmarks: Benchmark tests.
- codegen: Contains auxiliary functions of ARM assembly code generating.
- context: Context for compile errors
- golite: Contains Main function of the whole compiler (i.e., the file that includes the main function or entry point code into the compiler).
- grammars: ANTLR grammar files(i.e., **GoliteLexer.g4** and **GoliteParser.g4**) and the full ANTLR runtime jar file (i.e., **antlr-4.11.1-complete.jar**).
- ir: LLVM IR instruction definitions and function of translating LLVM to ARM.
- lexer: Autogenerated lexer files. Additionally, we write lexer.go to connect the autogenerated code into our compiler implementation.
- parser: Autogenerated parser files. Additionally, we write parser.go to connect the autogenerated code into our compiler implementation.
- sa: Semantic Analysis definitions, including entry of building symbol table and error report.
- symboltable: Symbol Table definition.
- token: Token definition, token line and column information.
- type: The type of token definition.
- README.md: Specific instructions to run our compiler.

In the following paragraphs, we will discuss the implementation of different phases in our compiler.

2 Implementation

2.1 Lexing and Parsing

The lexer and parser locate at **proj/lexer** and **proj/parser**. We use ANTLR (ANother Tool for Language Recognition) to build the lexer (i.e., scanner) and parser. We wrote the grammar files (i.e., **GoliteLexer.g4** and **GoliteParser.g4**) according to the CFG grammar which describes the Golite syntax. Then we use generate.sh to run the ANTLR with our grammar files and generated the lexer and parser in Go language.

For lexer, in our own **lexer.go**, we use *antlr.NewCommonTokenStream()* to generate the TokenStream and return all tokens by *lex.stream.GetAllTokens()* , including token's line, columnd, text and type information.

For parser, in our own **parser.go**, we implemented generating AST. The main process is that, we call *antlr.ParseTreeWalkerDefault.Walk* to walk through the whole tree by node. We also need to implement the exit listeners, when we are walking through the tree, each time we exit from a node, we generate the AST node and save in a map by location(i.e., location is our key, we use the line number + column number as our key).

Our exit order is from leaf(i.e., Factor) to root(i.e., Program). More specifically ,as shown in Figure 2, we will generate the *Factor AST node* first, and *Program AST node* last. When we are generating *Program AST node*, we already have these three nodes, *Types AST node*, *Declarations AST node*, *Functions AST node*. We only need to look up the map by location, get them from map and store them in the *Program AST node*.

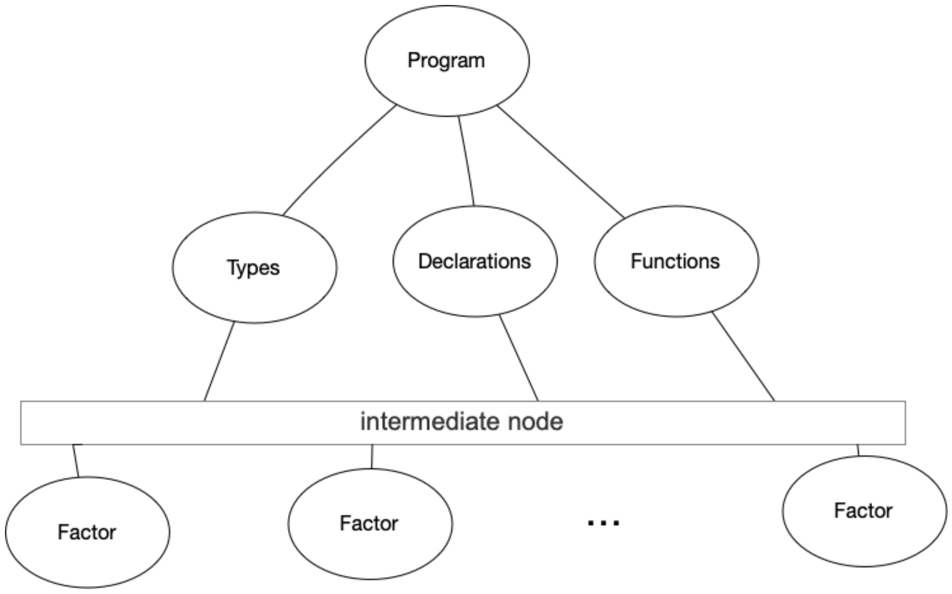


Figure 2: Basic Blocks Structure

2.2 Static Semantics

The compiler builds an AST and symbol tables, which are used in program type checking. If any errors occur, it prints syntax errors to the terminal with error messages. So the static semantics are split into two main parts: 1. Constructing the ”nested” symbol table; 2. Traversing the AST and checking type matching and semantic errors.

For constructing the “nested” symbol table, it is implemented by traversing from the root to the bottom. It includes three sub-symbol tables: Funcs, Globals, Structs and a Location table mapping function name with its variable and parameter location in the stack or register, i.e. arm table. To construct the ”nested” symbol table, we first record the type declarations and global variables at the top-level symbol table. Then, for each function, we construct a new symbol table and set the parent attribute to the top-level symbol table. In each function, we focus on the type of arguments, local variables and the return type.

For traversing the AST and type checking, the most important tasks are to check type matching and variable declaration. For variable redeclaration/duplication, it is checked when building the symbol table. When we insert a variable/struct/function into the symbol table, we will first check if it already exists and if yes, we append errors to the error buffer. When checking if types match, we infer types based on the symbol tables we have built above. For example, for a nested expression, different operators will lead to different types. There are several varied cases to determine the accurate type of this expression.

Control-flow paths are implemented by checking the return value of functions. Since main function is an entry point, we need to ensure the main function exists. For each function definition, the CheckReturn method checks the control-flow. We plan to use it to check if the return type of the function matches the actual return statement in the function and append errors if any. However, this part is only partially implemented for now.

2.3 LLVM IR

The definition of LLVM IR instruction locates at **proj/ir** folder. Because we get the LLVM IR by traversing the AST tree, we define the *TranslateToLLVM* that translate the validated AST from semantic analysis into LLVM IR located at **proj/ast** folder.

AST nodes are converted to LLVM IR instructions and then stored in **BasicBlock** struct. The traversal order is same as generating the AST in Figure 2. We call *Program.TranslateToLLVM()* at first, in the *Program.TranslateToLLVM* then we call *TranslateToLLVM()* of *Types*, *Declarations*, *Functions*, the rest can be done in the same manner.

We define the **BasicBlock** struct to implement the Control-flow. **BasicBlock** struct has **Preds** and **Succs**, two lists of **BasicBlocks**, use to store the successors and predecessors of the current block. We store each head block of each function in a slice (**i.e., var blocks** []**BasicBlock**), as shown in Figure 3. We only represent the successors of each block in the Figure 3 below.

When we translating the *Function*, we generate the **FuncEntry** at first and append it to the **blocks**, then we tranlating all the code inside the *Function* to LLVM IR, when we meet the *Conditional* or *Loop* we need to generate new blocks and connect them together as the CFG.

We can display the LLVM IR by iterate throught the **blocks**, and using BFS(Breadth-first search) to traverse the CFG started by each **FuncEntry**.

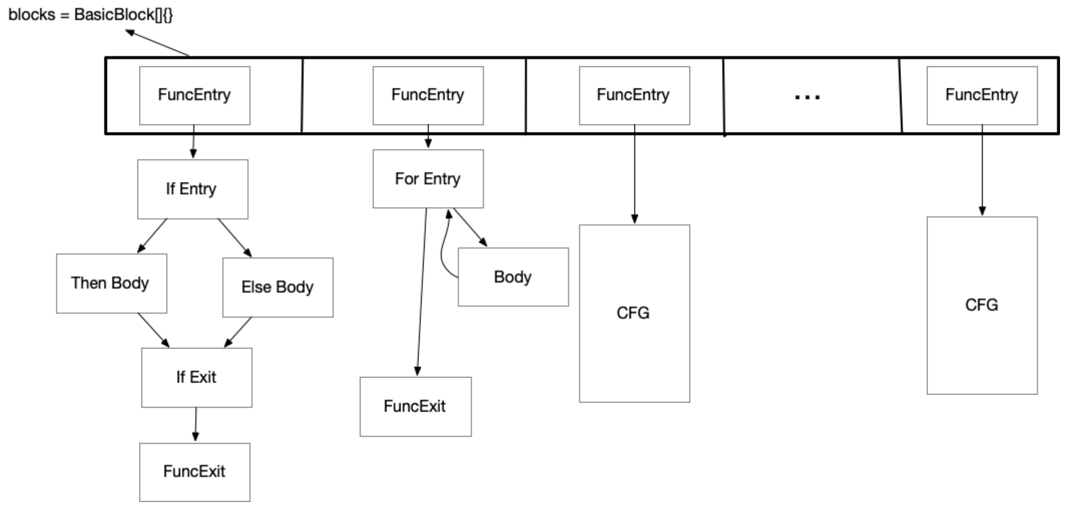


Figure 3: Basic Blocks Structure

2.4 Code Generation

The LLVM IR instruction to arm code translation is by calling *TranslateToAssembly* methods of each LLVM IR Struct located at **proj/ir** folder. The assembly code instruction definition is located at **proj/codegen** folder.

We use naive stack-based allocation in arm assembly code generation. The code generation takes the LLVM IR instructions and outputs them to an assembly file in ".s" format with the same name as the input file, including the header and footer (printf function and read function) of a valid assembly file.

For each function, we assign every local variable and temporary value(i.e., the register generated in LLVM IR) an offset to the SP(Stack Pointer) on the stack. For each function, The location of each variable and temporary value are stored in an **armTable**, which is passed into *TranslateToAssembly* function as parameter.

To better manage registers, we build a **registerTable** in **codegen** folder. For register 1 to 7, we use them for function parameters. If the function has parameters smaller than 7, we just leave the remaining register vacant. For register from 8 to 15, we use a registerTable to store if they are vacant or occupied. When allocating a new register, it loops all registers numbering from 8 to 15 and returns a vacant register to be allocated. Occupied registers are freed once it is no longer being used, for example, when facing an "str" LLVM instruction that stores the register value back to the stack.

To implement the control flow constructs (i.e., loops, conditionals, etc.) we use labels to represent their bodies and conditional expressions and branching operations to jump to those labels.

The debugging of code generation is very tricky, we confronted a lot of errors but cannot debug immediately and we only fix bugs based on printing information and variables for now.

2.5 Optimization

We don't have time to do the optimizations, but we do have some ideas and practical implementations.

For example, the most basic code like "a + 3": "3" is an integer literal, and during LLVM IR generation, our initial implementation will first load integer "a" into a new register and then store back to the stack. When adding them together, we will load "a" from the stack into a register and then add by "3" and then store in the stack, which is not efficient. In such case, we can just load "a" and "3" into two registers and add the value in the two registers together.

Besides, as mentioned by the professor in lectures and Ed, we could have improved our stack-based allocation by using graph coloring allocation.

3 Compile and Run

More detailed compile and run instructions are in README.md file. Here we just show an example of generating and executing assembly files.

1) Before running the test, please download the code into wing CS server:
```
git clone https://github.com/mpcs51300-win23/proj-siqi-23.git
git checkout arm
```
2) To generate .s assembly file, you need to cd to golite folder and then run:
```
go run golite.go -s ../benchmarks/simple/simple.golite
```
3) To run the assembly file, you will need to run:
```
 clang -o a.out simple.s
./a.out
```

References

[1] Class materials.