Overview of the compiler

Hengkuan Lu
ID: 12334243

December 11, 2021

1 Structure

Our compiler is consisted of a Front-end part and a Back-end part, where the Front-end part includes lexical analysis(a scanner), syntactical analysis(a parser), semantic analysis and intermediate representation(ILOC), and the Back-end part is the phase of code generation(though not fully completed). Unfortunately, there is no Middle-end part (Optimizer)in the compiler. The following sections of this paper are some more detailed description of each phase of the compiler and the mechanism behind.

2 Lexical Analysis

2.1 Lexemes

We firstly break the input source code(golite) into several kinds of tokens(lexemes), including the "ILLEGAL"(which represents a lexical error) and "EOF"(which represents the end of the input), functions("Scan", "Print" and "Println"), Data types("number", "true&false" and "nil"), special characters(semicolon, arithmetic characters, relation characters and etc.) and other key words("package", "import" and etc.). Besides, we have considered the comment in Golite(the lines that start with a "//").

2.2 Scanner

Then we build a deterministic finite state automata(DFA) based upon the grammar of Golite which you can see in our README.md file. With a DFA, we used direct coding strategy to build our scanner of the compiler. We do not use any Regular Expression package of Go, instead, we just follow our DFA and consider each situation while ignoring the whitespace(" ", "\ t" and "\ n").

3 Syntactical Analysis

3.1 Abstract Syntax Tree

All of the following Front-end part are based upon the data structure AST Node(interface). There are several structs(implementations) of AST Node: Program, Package(though not necessary for Golite), Types(definition of structs), Field(inner field of structs), Declaration, Function(including a slice of Paramter, a slice of Declaration and a slice of Statement), Statement(also a interface, including types like Block, Assignment, Read&Print, Conditional and (for)Loop Invocation and Return) and Expression(also a interface, including types like CompareExpr, BinOpExpr and etc.)
All these AST Nodes have several methods corresponded to the following parts of the compiler. They are, the "String" and "PrintAST" methods to print the AST structure of the source code, the "TokenLiteral" method for syntactical analysis, the "GetType", "TypeCheck", "checkReturn" and "performSABuild" methods for semantic analysis and "TranslateToILOC" method to transform the AST to ILOC intermediate representation.

3.2 Parser

Based upon the context-free grammars(CFG) of Golite, we use a leftmost derivation(LL parser). The task of the passer is to detect any syntactical error of the source code while building the Abstract Syntax Tree for further analysis.(rooted at the Program node) If there is a syntactical error, the passer should denote the information of the error and the returned AST should be "nil".

4 Semantic Analysis

After getting a valid AST representation of the source code, we would perform semantic analysis in two steps. Firstly, building a whole symbol table while detecting all redefinition of variables. Secondly, do type checking for all statements in all functions based upon the symbol table.

4.1 Symbol Table

The structure SymboTable has a point of its parent table and a map of string to Attribute. The Attribute interface has three implementations: VarEntry, StructEntry(has a map for its inner field) and FunctionEntry(has a slice of VarEntry and its own symbol table in local scope). Both VarEntry and StructEntry has an Register attribute for the ILOC intermediate representation and is initially attached one by one when the global symbol table is built in sequential order. When we are processing a variable that is already defined in the current scope, report this semantic error(we do not stop the semantic analysis until all errors are found). What's more, we have an "Usable" attribute for VarEntry, if a variable is in the inner field of a structure variable, it is set as "false" initially

and only became "true“ when the structure variable is “newed”(would return to “false” once been deleted), else they are “true” forever.

4.2 Type Checking

After a symbol table in the global scoped has been built, we do type checking for all required sections of the source code: all the statements within all defined functions. (the return type statement all checked apart by the “checkReturn” method) The rule of type checking is based on the given "Language Semantics“ of the "Language Overview“ on the course website. Actually, we can not guarantee that the semantic analysis are definitely correct for the whole all cases, however it works at least for our own test cases and all the given benchmarks.

5 Intermediate Representation

5.1 AST to ILOC

Only the AST nodes of Statement in Functions of the root Program all required to be translated to ILOC intermediate representation. We have defined for each function a structure FuncFrag with a label(function name) and a slice of Instrcution(which is another interface storing the information of ILOC for all the statements in that function). An Instruction is corresponded to an ILOC expression(i.e. add, sub, add, or, mov, str and etc.), each statement within a defined function is constructed by several different(at least one "mov“) ILOC instructions. It is worth mentioning that, we did not strictly follow the strategy in the course slides and videos: we defined a translateToILOC method for every Expression Node, even for a single ID(name of variable): just move from the original register to the same register(actually this can be omitted after the whole translation, but we have not preformed this optimization), and so in conditional loop, store the vale of the conditional expression in a regeister and compare it to "#1“ which has been stored in another new register right before this comparison. If they are equal, jump to the "if“ label, else jump to the "done“ if no "else“ block exists and to the "else“ label otherwise. This the same to any other statement that includes an Expression AST node.
We represent the bool variables and “nil” with integer like numbers: 0 for “nil”, ‘ for “true” and -1 for “false”(not 0 for “false” for the convenience of the code generation phase).

5.2 Function Frames

Besides, each FunFrag structure also has a Frame structure which contains the registers' information within the function frame: the local variables, the temporary variables , the reference access information of structures and all the registers and their relative location to the fram pointer. For every conditional loop and for loop, we use two more temporary registers for comparisons as discussed

previously, however these does not affect the correction of the code generation phase.

6 Code Generation

6.1 ILOC to Assembly Code

This phase of the compiler is only partially completed.
Each FuncFrag structure has a "GenerateCode" method which can transform its inner ILOC Instruction to assembly code(every ILOC Instruction has a "TranslateToArm" method). The naive algorithm for register allocating has been adopt. We give the main function (and other function with "void" return type) an Epilogue as there is no "Return" statement in the AST of such functions(and all the Epilogue of function are added within the "ret" instruction). Besides, we attached a skip label for "moveq" ,"movlt", etc because only "mov" is admitted in this phase.