Code Generation Design

Tatum Alenko

April 14, 2019

Table of Contents

Contents

Section 1. Analysis

Checklist of implemented code generation components

Memory Allocation

- ☒ 1.1: Allocate memory for basic types
- ☐ 1.2: Allocate memory for arrays of basic types
- ☐ 1.3: Allocate memory for objects
- ☐ 1.4: Allocate memory for arrays of objects

Functions

- ☐ 2.1: Branch to a function's code block, execute the code block, branch back to the calling function
- ☐ 2.2: Pass parameters as local values to the function's code block
- ☐ 2.3: Upon execution of a return statement, pass the return value back to the calling function
- ☐ 2.4: Call to member functions that can use their object's data members

Statements

- ☒ 3.1: Assignment statement
- ☐ 3.2: Conditional statement
- ☐ 3.3: Loop statement
- ☐ 3.4: Input/output statement

Aggregate data members access

- ☐ 4.1: For arrays of basic types (integer and float), access to an array's elements
- ☐ 4.2: For arrays of objects, access to an array's element's data members
- ☐ 4.3: For objects, access to members of basic types
- ☐ 4.4: For objects, access to members of array or object types

Expressions

- ☒ 5.1: Computing the value of an entire complex expression
- ☐ 5.2: Expression involving an array factor whose indexes are themselves expressions
- ☐ 5.3: Expression involving an object factor referring to object members

Section 2. Design

The overall design of the code generator involved the implementation of a new visitor denoted as `CodeGenerationVisitor` using a static tag-based approach. This visitor would simply traverse over the entire AST and generate Moon specific code using a `CodeFactory` object. This object would receive specific formatted strings when visiting the various AST nodes to add to either an `exec` or `data` string member. The `exec` member tracks the lines of Moon code responsible for executing the assembly op-code instructions while the `data` member tracks the lines of code responsible for reserving specific memory elements of the various tags used in the `exec` section.

The initial steps taken in implementing the code generation started with being able to execute the `write()` statement to standard output. This was chosen simply because it would allow easy testing thereafter of the other phases. Afterwards, simple literal integer expressions were implemented such as `write(1)`. The next phase chosen was then assignment statements. Following this, integer arithmetics were implemented involving the various integer arithmetic operators such as `+`, `*`, `/`, and soon after logical-integer operations such as `not`, `or`, `and`. Finally, the relational operators for integers were implemented, such as `==`, `<>`, `<`, `<=`, `>`, and `>=`. Although not complete at the time of writing, the next phases that will be implemented will be conditional and looping statements involving integer expressions and statements, followed by the incorporation of array and object memory elements and finally implementation of function memory elements and function calls.

Please note that the test project is where the project can be easily ran and output the necessary files. The output files are located in `Moon.Tests/resources/grammar/out/codegen/<test-file-name>`. Unfortunately, there was no time to run a proper command line driver. The symbol table files end in `.parse.symbols` and the AST have `.dot.pdf` generated if you have the `dot` tool installed locally on your computer. Moreover, the files ending in `.codegen.m` represent the generated Moon assembly code and the `.codegen.m.out` represent the resulting output code after running through the Moon executable.

Section 3. Use of Tools

The use of the Moon virtual processor by Peter Grogono was used in this assignment to drive the execution of the assembly code produced by the compiler. Moreover, the use of the `lib.m` Moon library by Peter Grogono for facilitating the writing of strings to standard output.