COMP 520 Compiler Design
Group Milestone #3

Code Generation for GoLite

Due: Tuesday, March 27 11:59 PM

Overview:

The final two milestones of your project are for developing the code generator for your compiler. You should begin by fixing all issues from the previous two phases of the project, paying special attention to the valid programs – they must make it to the code generator to output code! This milestone consists of a status report and some test programs, but you should get started as early as possible on your implementation.

To help you test your implementations, we have updated the assignment template repository (`https://github.com/comp520/Assignment-Template`) with 3 new scripts:

- `test.sh`: Updated to include output verification:
  1. Script calls `codegen` on your compiler;
  2. Your compiler writes the output file *wherever it wants*; and
  3. Script calls `programs/3-semantics+codegen/valid/verify.sh`
- `programs/3-semantics+codegen/valid/verify.sh`: Verification script that calls `execute.sh` with the input file and checks the result.
- `execute.sh`: Take the original `.go` file path as a parameter, and execute the generated code depending on your output file. *You must modify this script to find and execute your generated code – it is required as part of the submission for milestone 4*

Question 1: *Semantics & Code Generation Status* (15 points)

Now that the typechecker is finished, your compiler should reject all invalid programs for which code generation is impossible. Code generation assumes that the input programs to this phase are valid, and outputs equivalent code for the target language.

- Decide upon what sort of code you will generate. You can choose between low-level code such as Java bytecode, LLVM-IR, etc. or high-level code such as C, JavaScript, Python, Java, etc. Discuss the known advantages/disadvantages of your choice.

  Note: For teaching purposes the reference compiler outputs C++, therefore you must select another language

- As part of milestone 4 we will be evaluating how closely your compiler implements the Go semantics, so it is important to have a good understanding of the input language. Describe the semantics of the Go language for 3 of the following key areas (we will discuss some of these in class):

    1. Identifiers: blank identifiers and keywords;
    2. Scoping rules: function, block, control-flow;
    3. Variable declarations: initialization;
    4. Types (bounds checking, equality, tagged vs. untagged):
        - Base types;
        - Structs;
        - Arrays;
        - Slices;
    5. Assign statements: multiple vs. single, copying;
    6. Short declarations: multiple vs. single, declaration;
    7. If statements: initialization, condition types;
    8. Switch statements: types, case expressions, break;
    9. Loops: infinite, while, 3-part for;
    10. Printing: output format;
    11. Functions:
        - Pass-by semantics;
        - Return-by semantics;
        - Allowed parameter/return types;

- For the 3 areas selected, briefly describe the relevant aspects of your target language and the mapping strategy you will follow. Example for part of the switch statement:

    In Go, case labels are associated with a non-empty list of potentially non-constant expressions (i.e. function calls, array indexing, etc). For cases with more than one expression, the expressions are evaluated from left-to-right until one evaluates to `true`*, in which case the body of the label is executed. In C++, case labels are associated with a single constant expression – it may not be a function call or similar. In our code generator, we thus opt to use the if/else-if/else construct to implement the equivalent execution, using boolean logic and short-circuiting for expression lists. Each case is associated with an if-block, and the else branch handles the default case.*

- In this milestone you will also start implementing your code generator:

- Design the code generation patterns for the key constructs of your language. Ensure that you are not choosing a strategy that will generate excessively slow code. For example, if you are generating C code, then attempt to generate code that the C compiler will be able to effectively optimize.
- Implement the code generation mode (`codegen`) for a subset of your language. Choose some key language features. You should choose some simple expressions, statements and control constructs to start with.
- A brief summary (less than 1 page) of what you have implemented so far.

Question 2: *Valid Programs* (10 points)

For this question, design 3 programs corresponding to your selected features in Q1 that implement and test the tricky parts of the semantics. Your programs should begin with tilde comments (`//~`) describing the intended output, followed by the `main` function. When the code generation is successfully implemented, the output should match that of the comments.

```
//~2
//~2

package main

func main() {
        var a int = 2
        println(a)

        switch a {
        case 1, 2:
                println(a)
        default:
                println("default")
        }
}
```

If the program is intended to crash (out-of-bounds error), then include an exclamation comment (`//!`). These comments are required to work with the verification script. Be sure to test the output against the Go playground!

What to hand in

Create a tag in your Github repository named *milestone3* (lowercase, no extra characters). Information about creating git tags can be found at: `http://git-scm.com/book/en/v2/Git-Basics-Tagging`. Your project should be kept in the following format

```
/
    README                (Names, student IDs, any special directions for the TAs)
    programs/
        3-semantics+codegen/
            valid/        (your 3 codegen construct tests)

    doc/
        milestone3.pdf    (your semantics and status report)
```