COMP 520 Compiler Design
Group Milestone #4

Code Generation for GoLite

Due: Friday, April 24 11:59 PM

In this final milestone, you will design the code generator for your compiler (a fully functional GoLite compiler!). Begin by fixing all issues from the two previous phases, paying special attention to the valid programs – they must pass typechecking to output code.

Question 1: *Benchmarks* (5 points)

For each team member, provide 1 benchmark that executes for around 5s (in the Go playground) and computes the solution to a real world problem. It should be valid according to the reference compiler. Use a variety of language features to test performance throughout the generated code.

Question 2: *Code Generator* (45 points)

Target Language

First, decide upon your target language for the project. 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. Each language has its own set of advantages and disadvantages, so think carefully!

Note: The reference compiler outputs C++, therefore you must select another language.

Code Generator

Implement the `codegen` mode, designing a code generation pattern for each language construct. As part of our evaluation, we will cover the following language features as well as the efficiency of the generated code (for more details, see the GoLite tutorial):

1. Identifiers: blank identifiers and keywords;
2. Scoping rules: function, block, and control-flow;
3. Variable declarations: initialization and multiple declarations;
4. Types: Base types
    - Structs (equiality, tagged vs. untagged);
    - Arrays (bounds checking, equality);
    - Slices (bounds checking);
5. Assign statements: multiple and single, copying;

6. Short declarations: multiple and single, declaration vs. assign, copying;
7. If statements: initialization, condition types;
8. Switch statements: initialization, expression types, multiple case expressions, break;
9. Loops: infinite, while, 3-part;
10. Printing: output formats, newlines;
11. Functions: pass-by-value and return-by-value semantics;
12. Built-ins: cap, append, and len;

Execution

To help test your codegen implementation, the template repository (`https://github.com/comp520/Assignment-Template`) contains an `execute.sh` script that you must modify for your project.

- `test.sh`: For each program in `programs/3-semantics+codegen/valid/`
    1. Script executes `codegen` on your compiler;
    2. Compiler writes the output code 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` and checks that your output matches the expected result.
- `execute.sh`: Takes the *original* `.go` file path as a parameter, and executes the generated code. **You must modify this script to find and execute your generated code.**

Codegen programs should begin with tilde comments (`//~`) describing the intended output. When the code generation is successfully implemented, the output should match that of the comments.

```
//~2 3

package main

func main() {
        println(2, 3)
}
```

If the program is intended to crash (e.g. out-of-bounds error), include an exclamation comment (`//!`). Be sure to test the output in Go playground!

There is also a folder `programs/3-benchmarks+codegen/valid` (and corresponding `verify.sh` script) that allows you to time the execution of your generated code.

What to hand in

Create a tag in your Github repository named *milestone4* (`http://git-scm.com/book/en/v2/Git-Basics-Tagging`). Your project must be kept in the following format:

```
/
    README   (Names, student IDs, any special directions for the TAs, resources)
    programs/
        3-benchmarks+codegen/
            valid/
    src/       (Source code and build files)
    execute.sh (Updated execute script)
    build.sh   (Updated build script)
    run.sh     (Updated run script)
```