ECS 140A Programming Languages

Winter 2019

Homework 2

About This Assignment

- This assignment asks you to complete programming tasks using the Go programming language.
- To complete the assignment (i) download `hw2-handout.zip` from Canvas, (ii) modify the `.go` files in the `hw2-handout` directory as per the instructions in this document, and (iii) zip the `hw2` directory into `hw2-handout.zip` and upload this `zip` file to Canvas by the due date.

  Do not change the file names, create new files, or change the directory structure in `hw2-handout`.
- This assignment has to be worked on individually.
- We will be using Go version 1.11.4, which can be downloaded from https://golang.org/dl/

  Run the command `go version` to verify that you have the correct version installed:

  ```
  $ go version
  go version go1.11.4 <other output>
  ```
- Go 1.11.4 is installed on all *CSIF machines* in the directory `/usr/local/go/`; the `go` binary is, thus, at `/usr/local/go/bin/go`.

  Consider adding `/usr/local/go/bin` to your `PATH`.
- Information about using CSIF computers, such as how to remotely login to CSIF computers from home and how to copy files to/from the CSIF computers using your personal computer, can be found at http://csifdocs.cs.ucdavis.edu/about-us/csif-general-faq.
- Begin working on the homework early.
- Apart from the description in this document, look at the unit tests provided to understand the requirements for the code you have to write.
- Post questions on piazza if you require any further clarifications. Use private posts if your question contains part of the solution to the homework.

1 triangle (5 points)

- Modify the unit tests in the `TestGetTriangleType` function in `hw2-handout/triangle/triangle_test.go`.

  One unit test has already been written for you.

- The goal is to write enough unit tests to get 100% code coverage for the code in `hw2-handout/triangle/triangle.go`.

- From the `hw2-handout/triangle` directory, run the `go test -cover` command to see the current code coverage.

- From the `hw2-handout/triangle` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  ```

  ```
  $ go tool cover -html=temp.cov
  ```

- Do NOT modify the code in `hw2-handout/triangle/triangle.go`.

2 min (5 points)

- Modify the `Min` function in `hw2-handout/min/min.go`.

- The function `Min` should return the minimum/lowest value in the argument `arr`, and `0` if the argument is `nil`.

- Some unit tests are provided for you in `hw2-handout/min/min_test.go`.

  From the `hw2-handout/min` directory, run the `go test` command to run the unit tests.

- If needed, add more unit tests in `hw2-handout/min/min_test.go` to get 100% code coverage for the code in `hw2-handout/min/min.go`.

- From the `hw2-handout/min` directory, run the `go test -cover` command to see the current code coverage.

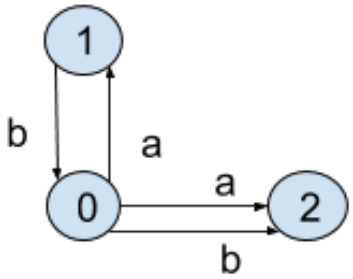
- From the `hw2-handout/min` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  ```

  ```
  $ go tool cover -html=temp.cov
  ```

3 nfa (10 points)

A non-deterministic finite automaton (NFA) is defined by a set of states, symbols in an alphabet, and a transition function. A state is represented by an integer. A symbol is represented by a rune, i.e., a character. Given a state and a symbol, a transition function returns the set of states that the NFA can transition to after reading the given symbol. This set of next states could be empty. A graphical representation of an NFA is shown below:



In this example, $\{0, 1, 2\}$ are the set of states, $\{a, b\}$ are the set of symbols, and the transition function is represented by labelled arrows between states.

- If the NFA is in state 0 and it reads the symbol a, then it can transition to either state 1 or to state 2.
- If the NFA is in state 0 and it reads the symbol b, then it can only transition to state 2.
- If the NFA is in state 1 and it reads the symbol b, then it can only transition to state 0.
- If the NFA is in state 1 and it reads the symbol a, it cannot make any transitions.
- If the NFA is in state 2 and it reads the symbol a or b, it cannot make any transitions.

A given final state is said to be *reachable* from a given start state via a given input sequence of symbols if there exists a sequence of transitions such that if the NFA starts at the start state it would reach the final state after reading the entire sequence of input symbols.

In the example NFA above:

- The state 1 is reachable from the state 0 via the input sequence *abababa*.
- The state 1 is *not* reachable from the state 0 via the input sequence *ababab*.
- The state 2 is reachable from state 0 via the input sequence *abababa*.

The transition function for the NFA described above is represented by the `expTransitions` function in `hw2-handout/nfa/nfa_test.go`. Some unit tests have also been given to you in `hw2-handout/nfa/nfa_test.go`. From the `hw2-handout/nfa` directory, run the `go test` command to run the unit tests.

- Write an implementation of the `Reachable` function in `hw2-handout/nfa/nfa.go` that returns `true` if a final state is reachable from the start state after reading an input sequence of symbols, and `false`, otherwise.
- If needed, write new tests, in `hw2-handout/nfa/nfa_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw2-handout/nfa` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw2-handout/nfa` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

Working with expression AST

- For parts 4 and 5, you will be using the `expr` package in the `hw2-handout/expr` directory.
- You should NOT modify any code in the `hw2-handout/expr` directory.
- `hw2-handout/expr/ast.go` defines an abstract syntax tree (AST) for arithmetic expressions.
- The `expr.Parse` function parses a string into an AST; see `hw2-handout/expr/parse.go`.
- The `expr.Eval` method evaluates an expression given a mapping from variables to values; see `hw2-handout/expr/eval.go`.
- The `expr.Format` function formats an expression AST as a string; see `hw2-handout/expr/print.go`.
- Note the use of type switches and type assertions in this code for inspecting the concrete types that an `Expr` interface type represents.
- This code is also described in Chapter 7 of The Go Programming Language book: http://gopl.io.

4 depth (10 points)

- Implement the function `Depth` in `hw2-handout/depth/depth.go` that, given an AST represented by `expr.Expr`, returns the maximum number of AST nodes between the root of the tree and any leaf (literal or variable) in the tree.
- Unit tests have been provided in `hw2-handout/depth/depth_test.go`.

  From the `hw2-handout/depth` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw2-handout/depth/depth_test.go` to ensure that you get 100% code coverage for your code in `hw2-handout/depth/depth.go`.

  From the `hw2-handout/depth` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw2-handout/depth` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

5 simplify (20 points)

- Implement the `Simplify` function in `hw2-handout/simplify/simplify.go`.

  Given an arithmetic expression as an `expr.Expr` and an environment `expr.Env`, `Simplify` simplifies the given expression using the values of the variables in the environment and by performing some algebraic simplifications.

- Your implementation of `Simplify` should perform the following simplifications:

  - Any `Var` whose name is in the provided map should be replaced with its value.
  - Any `unary` operator whose operand is a `Literal` should be replaced with a `Literal` representing the result.
  - Any `binary` operator whose operands are both `Literal`s should be replaced with a `Literal` representing the result.
  - Any `binary` operator performing addition, where one operand is `0`, should be reduced. $(0 + X = X = X + 0)$
  - Any `binary` operator performing multiplication, where one operand is `1` or `0`, should be reduced. $(1 * X = X = X * 1$ and $0 * X = 0 = X * 0)$

  No other simplifications should be performed.

- Unit tests have been provided in `hw2-handout/simplify/simplify_test.go`.

  From the `hw2-handout/simplify` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw2-handout/simplify/simplify_test.go` to ensure that you get 100% code coverage for your code in `hw2-handout/simplify/simplify.go`.

  From the `hw2-handout/simplify` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw2-handout/simplify` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

Working with the Go AST

- For parts 6, 7, and 8, you will be writing Go code that works with the abstract syntax tree (AST) of the Go language itself.
- Information about the `go/ast` package can be found at https://golang.org/pkg/go/ast/, which lists the different types of AST nodes along with their fields.

  You might also find it useful to take a look at the source code for `go/ast`, which can be found at https://golang.org/src/go/ast/.
- Sample code illustrating how to build the AST given Go code and then print it using `ast.Print` can be found at https://play.golang.org/p/1g-lt3D1N6a.
- Sample code computing the number of function calls using `ast.Inspect` can be found at https://play.golang.org/p/cq2OI6CA6v_n.
- An alternative to `ast.Inspect` is to use `ast.Walk`, which uses the `ast.Visitor` interface type.
- Sample code that modifies Go code programmatically can be found at https://play.golang.org/p/mRKeN7SZD8g.
- An introduction to the `go/ast` package can be found at https://zupzup.org/go-ast-traversal/.

6 branch (20 points)

- Modify `hw2-handout/branch/branch.go` by implementing the `branchCount` function that returns the count of the number of branching statements in a Go function.
- *Branching statements* are those where the program has a choice of what to execute next; e.g., `if` and `for` statements.
- The `branchCount` function is called from the `ComputeBranchFactor` function that takes a Go program as a string, and for each function in that program, counts the number of branching statements in the Go function by calling `branchCount`.

  `ComputeBranchFactor` returns a `map[string]uint` from the name of the function to the number of branching statements it contains.

  You should not need to modify the implementation of `ComputeBranchFactor`.
- Unit tests have been provided in `hw2-handout/branch/branch_test.go`.

  From the `hw2-handout/branch` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw2-handout/branch/branch_test.go` to ensure that you get 100% code coverage for your code in `hw2-handout/branch/branch.go`.

  From the `hw2-handout/branch` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw2-handout/branch` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  $ go tool cover -html=temp.cov
  ```

7 rewrite (30 points)

- Modify the `rewriteCalls` function in `hw2-handout/rewrite/rewrite.go` to rewrite occurrences of calls to `expr.ParseAndEval` in the input Go code so as to simplify the first argument to `expr.ParseAndEval`.

  The definition of the function `expr.ParseAndEval` can be found in `hw2-handout/expr/parse_and_eval.go`.

- As an example of the rewrite, the following statement in the input Go program:
  `x := expr.ParseAndEval("2 + 3", env)`
  would be rewritten to
  `x := expr.ParseAndEval("5", env)`

- The `rewriteCalls` function you have to implement has to do the following:
  - Identify calls to `expr.ParseAndEval` in the input Go code.
  - If the first argument to this call is a string literal, then parse this string into an `expr.Expr`.
  - Simplify this `expr.Expr` using the `simplify.Simplify` function you implemented in part 5 (using an empty `expr.Env`).
  - Format the resulting `expr.Expr` into a string.
  - Replace the first argument in `expr.ParseAndEval` with this string representing the simplified expression in the input Go code.

- Unit tests have been provided in `hw2-handout/rewrite/rewrite_test.go`, which read inputs and outputs from the `hw2-handout/rewrite_test` directory.

  From the `hw2-handout/rewrite` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw2-handout/rewrite/rewrite_test.go` to ensure that you get 100% code coverage for your code in `hw2-handout/rewrite/rewrite.go`.

  From the `hw2-handout/rewrite` directory, run the `go test -cover` command to see the current code coverage.

From the `hw2-handout/rewrite` directory, run the following two commands to see which statements are covered by the unit tests:

```
$ go test -coverprofile=temp.cov
$ go tool cover -html=temp.cov
```

8 bonus (10 points)

- This part of the homework is **extra credit**. You will get full credit for Homework 2 even if you do not attempt this part of the homework.
- This part is an extension of the rewrite you had to do in part 7.
- Modify the `rewriteCalls` function in `hw2-handout/bonus/bonus.go` to rewrite occurrences of calls to `expr.ParseAndEval` in the input Go code so as to simplify the first argument to `expr.ParseAndEval` *and* replace the call to `expr.ParseAndEval` if the expression in the first argument simplifies to a numeric constant (literal).

  The definition of the function `expr.ParseAndEval` can be found in `hw2-handout/expr/parse_and_eval.go`.
- As an example of this "bonus" rewrite, the following statement in the input Go program:
  `x := expr.ParseAndEval("2 + 3", env)`
  would be rewritten to
  `x := 5`
- The `rewriteCalls` function you have to implement has to do the following:
  - Identify calls to `expr.ParseAndEval` in the input Go code.
  - If the first argument to this call is a string literal, then parse this string into an `expr.Expr`.
  - Simplify this `expr.Expr` using the `simplify.Simplify` function you implemented in part 5 (using an empty `expr.Env`).
  - If the simplified expression is an `expr.Literal`, then replace the call to `expr.ParseAndEval` with this literal in the input Go code.
  - If the simplified expression is NOT an `expr.Literal`, then format the resulting `expr.Expr` into a string. Replace the first argument in `expr.ParseAndEval` with this string representing the simplified expression in the input Go code.
- Unit tests have been provided in `hw2-handout/bonus/bonus_test.go`, which read inputs and outputs from the `hw2-handout/bonus_test` directory.

  From the `hw2-handout/bonus` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw2-handout/bonus/bonus_test.go` to ensure that you get 100% code coverage for your code in `hw2-handout/bonus/bonus.go`.

  From the `hw2-handout/bonus` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw2-handout/bonus` directory, run the following two commands to see which statements are covered by the unit tests:

  ```
  $ go test -coverprofile=temp.cov
  ```

  ```
  $ go tool cover -html=temp.cov
  ```