ECS 140A Programming Languages

Spring 2018

Homework #2

Due 11:59pm Tuesday April 24th, 2018

This assignment asks you to complete programming tasks using the Go programming language. This assignment should be worked on individually. Please turn in your solutions electronically via Kodethon by the due date.

Getting Started

First, download the project files from Kodethon. Please see this support page[1] for details on downloading the required project files, as well as how to submit your solutions via Kodethon. Next, go to "Switch Environments" from your Kodethon dashboard and choose the "go" execution environment. Finally, you will need to open the Kodethon Terminal to execute commands. This can be done by selecting the grid icon in the top bar, selecting "CDE Shell", and then clicking the "Terminal" button in the upper-right. (**NOTE:** The CDE Shell behaves very differently from the Terminal. Make sure you're using the Terminal!) Further questions regarding Kodethon can be directed to the course Piazza forum using the `kodethon` tag.

The project code includes a simple "Hello, World!" program written in Go. Before changing anything, make sure you understand how to build and run this program. You can find the source code for this program in the "hello/" folder. The first thing you should do is set the GOPATH[2] environment variable so that the Go compiler knows how to traverse your project. You can do this by using `cd` in your terminal to navigate down to the homework directory, then running `export GOPATH=$(pwd)`.

You should work through the first four modules in A Tour of Go[3] to familiarize yourself with the language.

For the first three parts of this project, you will be working with a simple arithmetic language. This language comes with a parser, turning strings like `1 + x / 3` into abstract syntax trees (ASTs), and an evaluator, turning ASTs into numbers. You will be analyzing and transforming these ASTs to make the arithmetic language more useful; part 3 in particular adds basic support for units of measurement.

For the fourth through sixth parts of this project, you will be working with the AST of the Go language itself. You may need to traverse these ASTs differently; make sure to familiarize yourself with the documentation for go/ast[4] in particular, as well as go/token[5],

[1] https://support.kodethon.com/d/38-how-to-use-a-course-as-a-student
[2] https://golang.org/doc/code.html#GOPATH
[3] https://tour.golang.org/list
[4] https://golang.org/pkg/go/ast/
[5] https://golang.org/pkg/go/token/

go/format[6], and go/parser[7]. This article[8] gives a gentle introduction to using these packages.

For several parts of this project, you will need to write tests and ensure 100% test coverage of your code. You can generate a coverage profile using the `go test` command. For example, to generate a coverage profile for the `Simplify` method in the `eval/` package, run `go test eval -run Simplify -coverprofile=Simplify.cov`. You can then run `go tool cover -func=Simplify.cov | grep simplify.go` to see what the coverage results are. You can graphically see which lines of your code are covered by testing using the `go tool cover -html=Simplify.cov` command, which opens a new browser window with the results. (On Kodethon, you may need to download the HTML file for local viewing. Add the flag `-o Simplify.html` to generate an HTML file, which you can then download from Kodethon.) See this post[9] for more on coverage testing.

1 Depth (15 points)

For this part, you must implement a `Depth` method on `expr` ASTs, as declared in `eval/ast.go`. The `Depth` method should return the maximum number of AST nodes between the root of the tree and any leaf (literal or variable) in the tree.

Fill in the empty method bodies in `eval/depth.go` with your implementation. Make sure it passes the sample tests provided in `eval/depth_tests.go`.

2 Simplify (15 points)

Sometimes we have an expression where we have values for some of its variables, but not all of them. Or perhaps there are some obvious evaluations we can do without involving variables, as in "x + 5 + 8", which can be simplified to "x + 13". Your task is to write a `Simplify` method that fills in whichever variables we know and simplifies the expression.

As another example, "x + 2 + y" with `Env{'x': 3}` will be simplified to "3 + 2 + y". Because of how the provided parser works, the input is interpreted as `+(x, +(2, y))`, and this can't be simplified without reassociating the addition operation. Don't worry about reassociating or rotating the AST – just make sure the subtrees that you can simplify directly are simplified.

Your code 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 `Literals` should be replaced with a `Literal` representing the result.

[6] https://golang.org/pkg/go/format/
[7] https://golang.org/pkg/go/parser/
[8] https://zupzup.org/go-ast-traversal/
[9] https://blog.golang.org/cover

- 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.

Write your `Simplify` method in `eval/simplify.go`, and write tests for your code in `eval/simplify_test.go`. Ensure that your tests provide 100% coverage of code in `eval/simplify.go`.

3 FlattenUnits (20 points)

The supplied `Eval` cannot compute expressions involving units (`measure` nodes in the AST), so your next job is to implement a `FlattenUnits` method that replaces `measure` operations with arithmetic operations. The resulting AST must contain no `measure` operations. The second result of `FlattenUnits` must be the unit of the overall expression.

For binary operations, if the left-hand quantity is not a scalar, its unit determines the unit of the result. For example, "s(4) + min(5)" should produce an AST without units that, when evaluated, produces the value "304" (the number of seconds in 5 minutes, 4 seconds), and the second return value of `FlattenUnits` should be `"s"` (for "seconds"). On the other hand, addition and subtraction of incompatible units (e.g. "s(4) + F(10)") is illegal, and should cause a panic[10].

Your code does not need to handle multiplication or division of quantities with units. If you encounter an input like "m(5) / s(1)", you should call the standard function `panic()` to throw an exception and fail early. However, your code should still allow multiplication and division by scalars.

You should support the following kinds of quantities, in addition to scalars (unitless numbers)

- Length: Meter ("m"), kilometer ("km"), mile ("mi")
- Time: Second ("s"), milisecond ("ms"), minute ("min")
- Mass: Kilogram ("kg"), gram ("g"), pound ("lbs")
- Temperature: Fahrenheit ("F"), Celsius ("C"), Kelvin ("K")

Write your `FlattenUnits` method in `eval/units.go`, and write tests for your code in `eval/units_test.go`. Ensure that your tests provide 100% coverage of code in `eval/units.go`.

(We call this "FlattenUnits" because all of the units in the expression are "flattened" together, with the only unit left as the second result of the `FlattenUnits` call.)

[10] https://golang.org/pkg/builtin/#panic

4 ComputeBranchFactor (15 points)

For the next three parts, we'll be working with the AST of the Go language itself. As a warm up, write a `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 function. `ComputeBranchFactor` should return a `map[string]uint` from the name of the function to the number of branching statements it contains.

"Branching statements" are those where the program has a choice of what to execute next. These include `if` and `for`, as well as some other statements you may not have encountered before. Statements are named ending in "Stmt" in the `go/ast` documentation.

Write your `ComputeBranchFactor` function in `analysis/branches.go`, and write tests for your code in `analysis/branches_test.go`. Ensure that your tests provide 100% coverage of code in `analysis/branches.go`.

5 SimplifyParseAndEval (15 points)

The `eval` package from the first three tasks includes a `ParseAndEval` method that can be used in regular Go code. For this task, you will write a source preprocessor that simplifies `ParseAndEval` calls before the program is run. For instance, a call like `eval.ParseAndEval(``2 + 3'', env)` should be rewritten to `eval.ParseAndEval(``5'', env)`.

Write your `SimplifyParseAndEval` function in `analysis/rewrite.go`. Make sure it passes the sample tests provided in `analysis/rewrite_tests.go`.

The ability to manipulate the AST of a program enables you to do code generation[11]. For instance, the genny[12] library allows you to write a data structure once for a "generic" element type, and generate instances of that data structure for any other element type.

6 CyclomaticComplexity (20 points)

Cyclomatic complexity[13] is a measure of how many distinct basic control flow paths there are within a function, and it is sometimes used as a heuristic for how maintainable a function is. For this task, you will compute the cyclomatic complexity for each function in a Go program and output a `map[string]uint` from the function's name to its cyclomatic complexity.

You probably shouldn't use the `ast.Walk` method in `go/ast` for this. `ast.Walk` does not give you much control over the order in which AST nodes are visited.

Write your `CyclomaticComplexity` function in `analysis/cyclo.go`. Make sure it passes the sample tests provided in `analysis/cyclo_tests.go`.

[11] https://en.wikipedia.org/wiki/Automatic_programming
[12] https://github.com/cheekybits/genny
[13] https://en.wikipedia.org/wiki/Cyclomatic_complexity