ECS 140A Programming Languages

Spring 2018

Hints for Homework #2

Expression AST

Questions 1–3 of Homework 2 ask you to write Go to inspect and manipulate abstract syntax trees (ASTs) for expressions. These ASTs are defined in `eval/ast.go`. There are 5 different types that represent the expression AST: `Var`, `Literal`, `unary`, `binary`, and `measure`. Each of these types satisfy the `Expr` interface type defined in `eval/ast.go`. Take a look at the `Eval` methods defined for each of these types in `eval/eval.go`; this code should give an idea of how to traverse the AST.

Depth of expression AST

Question 1 of Homework 2 asks you modify the file `eval/depth.go` to compute the maximum depth of the tree for expression AST. You have to write the `Depth` function for each one of the AST types:

1. `Var`: The depth of the AST for a `Var` is just 1.

```
> x -> AST: x
  Depth = 1

> y -> AST: y
 Depth = 1
```

2. `Literal`: Similar to `Var`, the depth of the AST for a `Literal` is 1.

```
>  5-> AST: 5
  Depth = 1

> 23 -> AST: 23
 Depth = 1
```

3. `unary`: unary is little different from `Var` and `Literal`. For `unary` operators, there is a sign before the expression, which is going to add one more level to the depth of the expression without the sign:

```
>  -5 -> AST: -
                \
                 5
  Depth = 2

> +23 -> AST: +
                \
                23
 Depth = 2

> +(x+y+z) -> AST: +
                     |
                     +
                    / \
                   +   z
                  / \
                 x   y
  Depth = 1 + Depth of the AST for expression (x+y+z) = 4
```

4. `binary`: The `binary` has one operation and two expressions children, `x` and `y`, which are left and right children, respectively. You have to calculate the depth of both children and choose the greater one to return. **Do not forget to count the operation as a level.**

5. The `measure`: The code for handling `measure` is similar to that for `unary`.

Simplifying expression ASTs

Question 2 of Homework 2 asks to modify `eval/simplify.go` and implement a `Simplify` method for an expression AST. Simplification is carried out by recursively simplifying the children of an AST type, and then applying the rules listed in the Homework 2 handout on the resulting simplified expressions.

You might find the notion type assertions and type switches useful for inspecting the concrete types that an `Expr` interface type represents.

FlattenUnits

Question 3 of Homework 2 asks you to extend `eval/flatten.go` with more units and to handle division of certain units. Using the string concatenation operator + will greatly simplify your code for this question.

Working with the Go AST

Questions 4–6 of Homework 2 ask you to write Go code to inspect and manipulate Go code. Here are some hints as well as some sample code to help you get started:

- 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. This sample code using the notion function values and closures. You might find this useful for question 4 of Homework 2.
- 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. You might find this sample code useful for question 5 on Homework 2.
- You should refer to the documentation for `go/ast` at https://golang.org/pkg/go/ast/ to see what are 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/.
- This article[1] gives a gentle introduction to using `go/ast` packages.

Cyclomatic complexity

Question 6 of Homework 2 asks you to compute the cyclomatic complexity of a function. The **cyclomatic complexity** of a function is the number of distinct paths that an executing program might follow through the function. Here are two of the most important rules of determining cyclomatic complexity:

- If one block of code `A` is followed by another block of code `B`, the number of paths through both of them together is the number of paths through `A` *times* the number of paths through `B`.
- If there is a *choice* between one block of code `A` and another block of code `B`, the number of paths through them both is the number of paths through `A` *plus* the number of paths through `B`.

Suppose we have the following simple function:

[1] https://zupzup.org/go-ast-traversal/

```
func foo(x uint) uint {
    if x == 4 {
        return 2
    } else {
        if x == 5 {
            return 42
        } else {
            return 101
        }
    }
}
```

First, let's check our intuition. Depending on the input, there are three paths through the function `foo`: one if `x` is 4, one if `x` is 5, and one if `x` is anything else. So the cyclomatic complexity of function `foo` should be 3.

Now let's break it down systematically. Function `foo` is made of two blocks of code: the outer if branch and the outer else branch. They're combined by a choice, so however many paths there are through both, we'll need to sum them up. Clearly there is only one path within the first block, since it's a single statement. We proceed the same way with the second block: it breaks down into two more blocks mediated by a choice (the arms of the inner if statement), both of which have two paths. Summing, we get $1 + (1 + 1)$, which is indeed 3.

Let's work a slightly more complicated example:

```
func bar(x uint) uint {
    var y uint
    if x == 4 {
        y = 2
    } else if x == 5 {
        y = 42
    }

    if x == 10 {
        y += 101
    }

    return y
}
```

Function `bar` is made up of three outer blocks combined in sequence, so we'll need to multiply their path counts. The last block is just a return statement, so it counts for only one path. The second block is an if statement with no else, but notice that there are *still* two paths through it: one if the condition succeeds, and one if it fails. The absence of an else block does not mean there are fewer paths! So the second block has two paths. Finally, the first block has three paths, by the same logic. So the cyclomatic complexity of the function `bar` is $3 * 2 * 1 = 6$. The tricky business with a missing else block comes up again

in `switch` statements: a switch statement always has an implied `default` case, just like an `if` statement always has an implied `else` branch.

Finally, let's look at an example with loops:

```
func baz() {
    for x := range []uint{1, 2, 3, 4} {
        if x == 1 {
            fmt.Println("A good number")
        } else {
            fmt.Println("Not so great")
        }
    }
}
```

Loop statements have two cases: the condition holds (so the loop continues) or the condition does not hold (so the loop ends). If the condition does not hold, we leave the loop and move on, giving us one path. Otherwise, we enter the loop body, which (in this case) has two paths. So we add one extra path to those of the loop body, and the cyclomatic complexity of the function `baz` is $1 + 2 = 3$.

Pseudocode for cyclomatic complexity

```
PseudoCyclo(node):
    If node is a block of statements:
        Return the product of PseudoCyclo(stmt) over
        all statements in node.
    If node is a clause in a switch statement:
        The clause contains a block of statements, so
        return the product of PseudoCyclo(stmt) for all
        statements in the clause.
    If node is a switch statement or type-switch statement:
        Return the sum of PseudoCyclo(clause) for all clauses
        in node, but also add one if there is no default clause.
    If node is an if statement:
        Return the sum of PseudoCyclo(then)
        and PseudoCyclo(else).
    If node is a loop:
        Return 1 + PseudoCyclo(body)
    Otherwise, it's some other kind of statement.
    We assume there's only one path for other kinds
    of statements, so return 1.
```

For question 6 of Homework 2, we only need to account for the following kinds of statements in the Go AST: `BlockStmt`, `CaseClause`, `SwitchStmt`, `TypeSwitchStmt`, `IfStmt`, `ForStmt`, and `RangeStmt`. All other kinds of statements can be assumed to have a single control flow path.

You probably should not use the `ast.Inspect` method in `go/ast` for computing the cyclomatic complexity. `ast.Inspect` does not give you much control over the order in which AST nodes are visited.