ECS 140A Programming Languages
Homework 2

About This Assignment

- This assignment relates to parsing and asks you to complete programming tasks using the Go programming language.
- **You are only allowed to use the subset of Go that we have discussed in the course. No credit will be given in this assignment if any of the problem solutions use material not discussed in the course.** Please use Piazza for any clarifications regarding this issue.
- To complete the assignment (i) download `hw2-handout.zip` from Canvas, (ii) modify the `members.txt` and `.go` files in the `hw2-handout` directory as per the instructions in this document, and (iii) zip the `hw2-handout` 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 of `hw2-handout`.
- This assignment can be worked on in a group. Even if working in a group, each student has to submit a copy of the work to Canvas to get credit. See the Syllabus for more details.
- List all the names and email addresses of all members of the group in the `members.txt` file in the `hw2-handout` directory, one per line in the format `name <email>`.

  If you are working individually, then only add your name and email to `members.txt`.
- Refer to **Homework 0** for instructions on installing the correct versions of the programming language as well as using CSIF computers.
- 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.
- Keep your homework solution after you submit it. You may need to use it for later assignments.

1 branch (20 points)

For this part of the assignment, you only need to modify `hw2-handout/branch/branch.go` and `hw2-handout/branch/branch_test.go`. You will be writing Go code that works with the abstract syntax tree (AST) of the Go language itself.

- 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. In this part we consider the following branching statements in Go:

  - `If statements.`
  - `Switch statements.`
  - `Type switch statements.`
  - `For statements.`
  - `Range 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`.

- See the unit tests in `hw2-handout/branch/branch_test.go` to better understand the behavior of the code you have to write.

  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
  ```

Working with the Go AST

- 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.

2 Term Parser (30 points)

For this part of the assignment, you only need to modify `hw2-handout/term/parser.go` and `hw2-handout/term/parser_test.go`.

- You need to define a `struct` type that implements the `Parser` interface defined in `hw2-handout/term/parser.go`. Do not modify this interface.

  Specifically, this type needs to implement the method `Parse(string) (*Term, error)` that takes a `string` and parses it to a `*Term` if string is in the language of grammar G defined below, else it returns an error.

  The grammar G, whose start symbol is `<start>`, is:

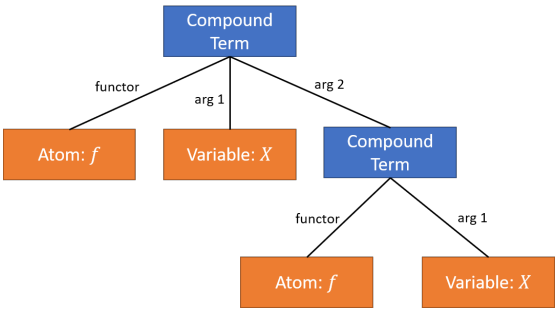
  ```
  <start>    ::= <term> | ϵ
  <term>     ::= ATOM | NUM | VAR | <compound>
  <compound> ::= <functor> ( <args> )
  <functor>  ::= ATOM
  <args>     ::= <term> | <term>, <args>
  ```

- The type `Term` is defined in `hw2-handout/term/term.go`. A term can be a variable, atom, number, or a compound term, as indicated by `TermType`.

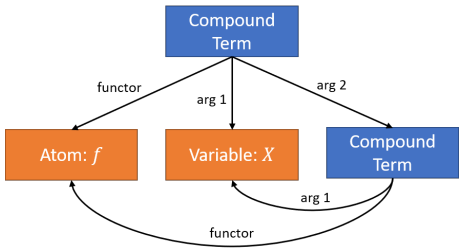
  Instead of returning an Abstract Syntax Tree (AST), the `Parse` method returns a DAG[1] (Directed Acyclic Graph) representing a term. For example, the output of `Parse` for the input string "`f(X,f(X))`" should be the term DAG in Figure 1b instead of a term AST in Figure 1a. A DAG is a more compact representation of a term, because the representations for common sub-terms are shared. Such a representation

[1] https://en.wikipedia.org/wiki/Directed_acyclic_graph

not only reduces space requirements, but also improve time efficiency of algorithms working over terms. As we will see later on in the course, these terms are used in Prolog.



(a) AST of term $f(X, f(X))$



(b) A valid DAG of term $f(X, f(X))$

Figure 1

See also the tests in `hw2-handout/term/parser_test.go` to understand the behavior of `Parse`.

- You need to modify the `NewParser` function in `hw2-handout/term/parser.go` to create an instance of the type that satisfies the `Parser` interface.

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

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

  From the `hw2-handout/term` 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
  ```

- We have provided an implementation of a *lexer* in `hw2-handout/term/lexer.go` for the grammar G.

  The lexer performs lexical analysis, converting an input string into a sequence of lexical *tokens*, which correspond to terminals in the grammar G (e.g., `ATOM`, `NUM`, `VAR`, `)`) or the end-of-file (EoF) symbol.

  For example, the lexer turns the input string "`f(X,f(X))`" into tokens `"f", "(", "X", ",", "f", "(", "X", ")", ")"`.

- We have provided unit tests the lexer in `hw2-handout/term/lexer_test.go`. These tests should help you understand how to use the lexer in your implementation of the parser.