ECS 140A Programming Languages

FALL 2022

Homework 4

About This Assignment

- This assignment asks you to complete programming tasks using the Go programming language and the SWI-Prolog programming language.
- **You are only allowed to use the subset of Go and Prolog that we have discussed in class. No credit will be given in this assignment if any of the problem solutions use material not discussed in class.** Please use Piazza for any clarifications regarding this issue.
- **Your program will receive no credit if it does not compile. Note that for Prolog programs, your program may have syntax errors but still pass all the tests. Make sure to read through all the output of `swipl -s <test-name>.plt`. Usually the error starts with "ERROR".** You should also try to remove "Warnings" in your program.
- To complete the assignment (i) download `hw4-handout.zip` from Canvas, (ii) modify the `members.txt`, `.go` and `.pl` files in the `hw4-handout` directory as per the instructions in this document, and (iii) zip the `hw4-handout` directory into `hw4-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 `hw4-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 `hw4-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.

1 nfa (10 points)

For this part of the assignment, you only need to modify `hw4-handout/nfa/nfa.pl` and `hw4-handout/nfa/nfa.plt`.

- An non-deterministic finite automaton (NFA) is defined by a set of states, symbols in an alphabet, and a transition relation.

  A state is represented by an integer. A symbol is represented by a character.

  Given a state `St` and a symbol `Sym`, a transition relation `transition(N, St, Sym, States)` defines the list of states `States` that the NFA `N` can transition to after reading the given symbol. This list of next states could be empty.

  A graphical representation of an NFA along with the corresponding transition facts are shown below.

```
transition(expTransitions, 0, a, [1,2]).
transition(expTransitions, 0, b, [2]).
transition(expTransitions, 1, a, []).
transition(expTransitions, 1, b, [0]).
transition(expTransitions, 2, a, []).
transition(expTransitions, 2, b, []).
```

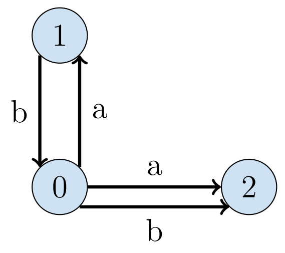


Figure 1: Example NFA diagram with corresponding `transition` facts for the NFA `expTransitions`.

- 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*.

- Complete the definition of the predicate `reachable(N, StartState, FinalState, Input)` in `hw4-handout/nfa/nfa.pl`, which which is true if the NFA `N` can reach state `FinalState` from the state `StartState` after reading the input list of symbols in `Input`.

  The `transition` facts are listed in `hw4-handout/nfa/transition.pl`.

- Use the following commands to run the unit tests provided in `hw4-handout/nfa/nfa.plt`:

  ```
  $ cd hw4-handout/nfa/
  $ swipl -s nfa.plt
  ```

- Ensure that the coverage for the `hw4-handout/nfa/nfa.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/nfa/nfa.plt`.

- A different set of the `transition` facts will be used when grading.

2 matrix (15 points)

For this part of the assignment, you only need to modify `hw4-handout/matrix/matrix.pl` and `hw4-handout/matrix/matrix.plt`.

- A $1 \times m$ matrix of integers can be represented as a list in Prolog.

  For example, the 1×3 matrix $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ is represented as the list `[1, 2, 3]`.

- Given a $1 \times m$ matrix *mat*, the numbers a and b are *adjacent* in m if and only if a occurs immediately to the left or right of b in *mat*.

- In `hw4-handout/matrix/matrix.pl`, implement the `are_adjacent` predicate.

  `are_adjacent(List, A, B)` returns `true` if the two numbers `A` and `B` are adjacent in the $1 \times m$ matrix represented by `List`, else it returns `false`.

- An $n \times m$ matrix of integers can be represented as a list of list of integers stored in row-major order. For example, the 2×3 matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is represented as the list `[[1, 2, 3], [4, 5, 6]]`.

- In `hw4-handout/matrix/matrix.pl`, implement the `matrix_transpose` predicate.

  `matrix_transpose(Matrix, Result)` returns `true` if Result is the transpose of the $n \times m$ matrix represented by `Matrix`.

For example, the transpose of the 2×3 matrix above is the 3×2 matrix $\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$, which is represented as the list `[[1, 4], [2, 5], [3, 6]]`.

You are not allowed to use the built-in `transpose/2` predicate.

- Given a matrix *mat*, we say that the numbers a and b are *neighbors* in *mat* if b occurs to the immediate left, right, top, or bottom of a in *mat*.

  For example, in the 2×3 matrix above, 1 and 2 are neighbors, 2 and 5 are neighbors, 5 and 6 are neighbors, and 2 and 6 are NOT neighbors.

- In `hw4-handout/matrix/matrix.pl`, implement the `are_neighbors` predicate.

  `are_neighbors(Matrix, A, B)` returns `true` if the two numbers `A` and `B` are neighbors in the $n \times m$ matrix represented by `Matrix`, else it returns `false`.

- Use the following commands to run the unit tests provided in `hw4-handout/matrix/matrix.plt`:

  ```
  $ cd hw4-handout/matrix/
  $ swipl -s matrix.plt
  ```

- Ensure that the coverage for the `hw4-handout/matrix/matrix.pl` file is 100%. See the `%Cov` column in the output of the unit tests above.

  Write additional tests, if needed, in `hw4-handout/matrix/matrix.plt`.

3 Disjointset (6 points)

- In this part of assignment, you will implement the disjoint set data structure, **which is the same as the disjointset assignment in homework 1. You can (re)use your solution or the solution we discussed in class.**

- For this part of assignment, you only need to modify `hw4-handout/disjointset/disjointset.go` and `hw4-handout/disjointset/disjointset_test.go`.

- A *disjoint-set data structure* maintains a collection $\mathcal{S}$ of disjoint dynamic sets $S_1, S_2, \ldots, S_k$; viz., $S_i \cap S_j = \emptyset, i \neq j$.

- For the purpose of this assignment, assume that initially $\mathcal{S}$ contains a singleton set for each integer; that is, $\mathcal{S} = \{\ldots, \{-2\}, \{-1\}, \{0\}, \{1\}, \{2\}, \{3\}, \ldots\}$.

- Each dynamic set S_i is identified by a *representative*.

  The only requirement for this representative is that it is an element of S_i, and that the representative remains the same if S_i has not been modified.

  The `FindSet`(u) returns the representative of the (unique) set containing u.

  For example, given the initial value of $\mathcal{S}$ stated above, we have `FindSet`$(1) = 1$, `FindSet`$(2) = 2$, and so on.

- The collection $\mathcal{S}$ can be modified using the `UnionSet` operation.

  Let S_u and S_v be the dynamic sets containing u and v, respectively. The operation `UnionSet`(u, v) merges the dynamic sets S_u and S_v. The resulting set $S_u \cup S_v$ is added to $\mathcal{S}$, while S_u and S_v are removed from $\mathcal{S}$.

  For example, performing `UnionSet`$(1, 2)$ on the initial value of $\mathcal{S}$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\ldots, \{-2\}, \{-1\}, \{0\}, \{1, 2\}, \{3\}, \ldots\}$. Now `FindSet`$(1) =$ `FindSet`(2).

  Performing `UnionSet`$(0, -1)$ on $\mathcal{S}$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\ldots, \{-2\}, \{-1, 0\}, \{1, 2\}, \{3\}, \ldots\}$. Now `FindSet`$(-1) =$ `FindSet`(0).

  Performing `UnionSet`$(2, -1)$ modifies $\mathcal{S}$ to be $\mathcal{S} = \{\ldots, \{-2\}, \{-1, 0, 1, 2\}, \{3\}, \ldots\}$. Now `FindSet`$(-1) =$ `FindSet`$(0) =$`FindSet`$(1) =$`FindSet`(2).

- The `DisjointSet` interface type specifying the `FindSet` and `UnionSet` functions is defined in `hw4-handout/disjointset/disjointset.go`. **Do not change this interface type**.

- Modify `hw4-handout/disjointset/disjoint.go` to define a `struct` type that implements the `DisjointSet` interface. Modify the `NewDisjointSet` function to create an instance of this type.

- Unit tests have been given to you in `hw4-handout/disjointset/disjointset_test.go`. From the `hw4-handout/disjointset` directory, run the `go test` command to run the unit tests.

- If needed, write new tests, in `hw4-handout/disjointset/disjointset_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw4-handout/disjointset` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw4-handout/disjointset` 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
  ```

- The assignment does NOT *require* you to use a specific algorithm when implementing the `DisjointSet` interface; you are free to implement it in any way you want.

  However, you are *encouraged* to implement the data structure as a disjoint-set forest: each disjoint set is represented as a rooted tree with the root of the tree containing the representative. The *path compression* and *union by rank* heuristics ensure that the height of these rooted trees do not grow too large.

  Some of you might have been taught this algorithm in ECS 60 or ECS 122A. You will also find this approach described in a standard algorithms textbook, such as

  Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

  or in this Wikipedia article.

- We will be using a **timeout of 100 seconds** when running the tests. Our reference solution completes the tests in around 1 second.

4 Term Parser (6 points)

- In this part of assignment, you will implement a parser for term, **which is almost the same as the term parser assignment in homework 2 with one additional requirement. You can (re)use your solution or the solution we discussed in class.**

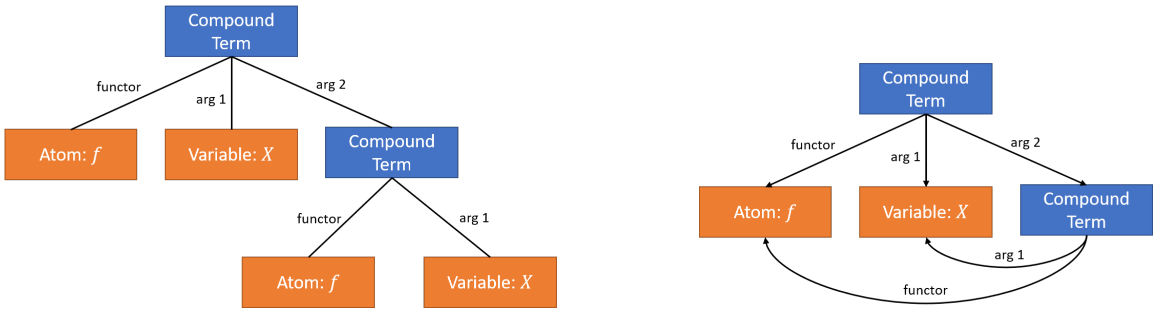
- For this part of assignment, you only need to modify `hw4-handout/term/parser.go` and `hw4-handout/term/parser_test.go`.

- You need to define a `struct` type that implements the `Parser` interface defined in `hw4-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>
```



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

Figure 2: AST and DAG of term $f(X, f(X))$.

- The type `Term` is defined in `hw4-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

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

`Parse` for the input string "`f(X,f(X))`" should be the term DAG in Figure 2b instead of a term AST in Figure 2a. A DAG is a more compact representation of a term, because the representations for common sub-terms are shared. Such a representation 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.

See also the tests in `hw4-handout/term/parser_test.go` to understand the behavior of `Parse`.

- **Additionally, all terms parsed by the same instance of your type—which satisfies the `Parser` interface—should be in the same DAG. In other words, they should share terms.**

  Take the code snippet below as an example:

  ```
  parser := NewParser()
  term1, err := parser.Parse("f(X, f(X))")
  term2, err := parser.Parse("f(f(X), Y)")
  ```

  the `term1` and `term2` parsed by the same instance of a `Parser` implementation from two input strings `"f(X, f(X))"` and `"f(f(X), Y)"` should share common terms in the same DAG as illustrated in Fig. 3. This requirement is useful when implementing term unification (part 5).

  The solution for homework 2 that we released satisfies this requirement.

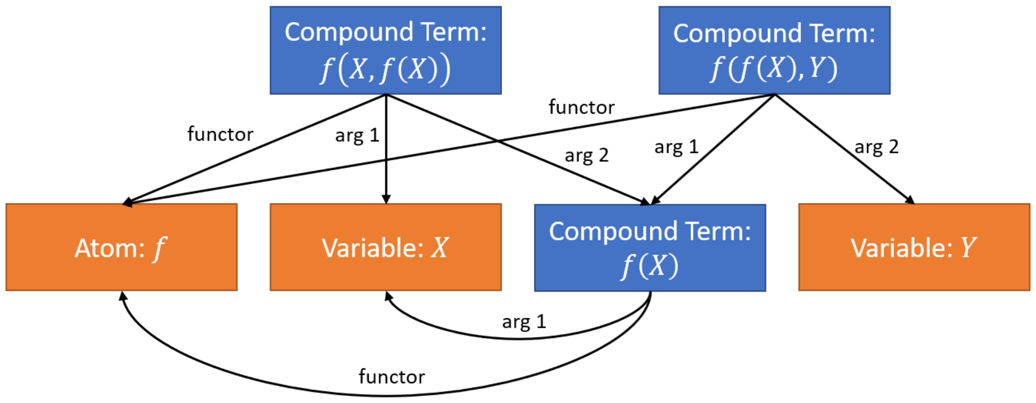


Figure 3: DAG of terms $f(X, f(X))$ and $f(f(X), Y)$.

- You need to modify the `NewParser` function in `hw4-handout/term/parser.go` to create an instance of the type that satisfies the `Parser` interface.

- If needed, write new tests in `hw4-handout/term/parser_test.go` to ensure that you get 100% code coverage for your code.

  From the `hw4-handout/term` directory, run the `go test -cover` command to see the current code coverage.

From the `hw4-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 `hw4-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 `hw4-handout/term/lexer_test.go`. These tests should help you understand how to use the lexer in your implementation of the parser.

5 Term Unification (20 points)

- In this assignment, you will implement the most general unifier for terms in the Go programming language using the disjoint set data structure and the term parser.

- For this part of the assignment, you only need to modify `hw4-handout/unify/unify.go` and `hw4-handout/unify/unify_test.go`, which implements the most general unifier for terms. The test file uses the term parser from part 4. Your solution can use the implementation of the disjoint-set data structure from part 3.

- For example, for the unification of two terms `f(X, g(1))` and `f(g(2), g(Y))`, the only solution is $\{X \mapsto g(2), Y \mapsto 1\}$.

- You need to define a `struct` type that implements the `Unifier` interface defined in `hw4-handout/unify/unify.go`. Do not modify this interface.

  Specifically, this type needs to implement the method `Unify(*term.Term, *term.Term) (UnifyResult, error)` that takes two terms as input and unifies them. `Unify(s, t)` returns the most general unifier if the input terms `s` and `t` are unifiable, otherwise returns an error.

- See the tests in `hw4-handout/unify/unify_test.go` to understand the behavior of `Unify`.

- You need to modify the `NewUnifier` function in `hw4-handout/unify/unify.go` to create an instance of the type that satisfies the `Unifier` interface.

- If needed, write new tests, in `hw4-handout/unify/unify_test.go` to ensure that you get 100% code coverage for your code in `hw4-handout/unify/unify.go`.

  From the `hw4-handout/unify` directory, run the `go test -cover` command to see the current code coverage.

  From the `hw4-handout/unify` 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
  ```

General Tips on Prolog

- When developing your program, you might find it easier to first test your predicate interactively before using the test program. You might find trace predicate useful in debugging your predicate. You can find information on tracing and debugging here: http://www.swi-prolog.org/pldoc/man?section=debugger.

- The command `swipl myFile.pl` runs the swipl interpreter with functions defined in `myFile.pl` already loaded (but not run).

- You can start swipl interactively using:

  ```
  $ swipl
  ```

- To load function definitions from `myFile.pl` in the current directory (notice the `.` at the end of the command, this should be at the end of every function call and at the end of the last line of every function definition):

  ```
  ?- [myfile].
  ```

- To exit error mode (i.e. an exception was thrown), type `a` (for abort):

  ```
  ?- bad_func(parameters).
  <error output>
     Exception: <error output> ? abort
  % Execution Aborted
  ?-
  ```

- You can exit the interactive swipl interpreter using:

  ```
  ?- halt.
  ```