Computer Science 324 | 10 November, 2008
St. George Campus | University of Toronto

Homework Assignment #3

Due: Friday, 5 December, 2008, by 5pm

No submissions will be accepted after 5pm Friday, 5 December

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.*

Handing in this Assignment

What to hand in on paper: Please use an **unsealed** envelope, attaching the cover page provided here to the front. Note that without a properly completed and **signed** cover page, your assignment will not be marked. Put inside:

1. A printout of your code. **Write your student number on the first page of this printout**.
2. Your answer to question 4.
3. For each procedure that you write, you must describe the testing strategy that you used. This should include a table listing the test cases that you designed for your procedure, what output your procedure returned, and an explanation of why the test case and output are significant in verifying the correctness of the procedure. Read the following for a discussion of good testing practices:

   `http://www.cs.toronto.edu/~gpenn/csc324/software.testing.pdf`

You must hand in this part of the assignment on the due date at the location designated by your TA.

What to hand in electronically: In addition to your paper submission, you must submit your code electronically. Use this command: `submit -c csc324h -a A3 a3.pl`, from the directory where `a3.pl` lives. Type **man submit** for more information. You can also use the CDF secure website: `https://www.cdf.utoronto.ca/students`.

Warning: marks will be deducted for incorrect submission. Note that if the code submitted electronically differs from the code submitted on paper, we will only mark the electronically submitted version (if, in such a case, you put comments, etc. only on paper, we will mark the question as if no comments, etc. were provided).

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact function names and argument(s) (including the order of arguments) specified,
- use the exact file name specified (`a3.pl`),
- not load any other files from your submitted file, and
- not display anything but the function output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Other Information You must read the requirements for code and marking information on the following web page: `http://www.cs.toronto.edu/~zkincaid/a3guidelines.html` This document constitutes part of the official requirements for this assignment.

Bulletin Board: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the bulletin board, linked from the course home page.

Computer Science 324 | 10 November, 2008
St. George Campus | University of Toronto

Homework Assignment #3
Cover Sheet

Last Name:

First Name:

CDF login:

Email:

I have read, understood, and agree to the policies described in the Course Information handout, including the policy on collaboration.

Signature: ______________________________

Question 1. (10 marks) Define a predicate, `freq/1`, which takes as its argument a (double-quoted) input string, and uses `assert` and `retract` to count the number of times each character occurs in the string. You will need the built-in `char_code/2` to convert between characters and their ASCII codes. The dynamic predicate you assert should be of the form, `count(Char,Count)`, where `Char` is the character (*not* its ASCII code), and `Count` is a natural number. You cannot make any assumptions about the state of the `count/2` clauses before your predicate is called, so be sure to initialize the Prolog database by calling `retractall(count(_,_))` at the beginning of your code.

Question 2. (30 marks) A reference chain in Prolog is implemented as a recursively nested term data structure that pairs a structure with a variable, e.g., *Reference-Structure*. It is dereferenced by chasing the recursive embedding until an uninstantiated variable is witnessed as a reference. This indicates that the structure paired with the uninstantiated variable is the most current version of that structure. For example, the chain:
`(X-f(a,b))-g(a,Y)`
represents a structure that has been updated from `g(a,Y)` to the new, and current structure of `f(a,b)`. The update was achieved by instantiating the reference variable associated with `g(a,Y)` to the pair, `X-f(a,b)`.

Pure Prolog has no way to destructively modify instantiated terms, so reference chains are often used when destructive updates are necessary.
(a) (5 marks) Define a predicate, `deref/3`, which takes as input to its first argument a reference chain, and returns as its second and third arguments the uninstantiated variable that is the chain's current reference, and the structure that is its current structure, respectively.
(b) (15 marks) Implement the *union-find* data structure using reference chains. Union-find keeps track of a mapping of discrete elements (in our case, ground Prolog terms) to disjoint sets, and supports three operations on this mapping:

1. `make/2`: given an element (first argument), this returns a mapping (second argument) that maps the element to a singleton set consisting only of that element (the mapping's domain should be the input element only);

2. `combine/3`: given two mappings (first two arguments), M_1 and M_2, with disjoint domains, D_1 and D_2, this returns a single mapping, M, in its third argument with the domain $D_1 \cup D_2$ such that $M(x) = M_1(x)$ if $x \in D_1$ and $M(x) = M_2(x)$ if $x \in D_2$. You may assume that the domains of M_1 and M_2 are disjoint.

3. `find/3`: given an element (first argument) and a mapping (second argument), this returns the set (third argument) to which the mapping maps the element;

4. `union/2`: given two sets (first and second arguments) from some mapping, U_1 and U_2, the mapping is updated so that any element that belonged to U_1 or U_2 before now belongs to $U_1 \cup U_2$.

Note that your implementation of union cannot return a new mapping as a third argument. It must update the mapping that it is passed as input so that, upon successful termination of `union/2`, the input mapping itself reflects the update.

Your representation of sets should be a reference chain. Do not use `assert` or `retract`.

(c) (10 marks) Now implement the union-find data structure without reference chains (and without `assert` and `retract`). **Hint:** term unification at the Prolog source-code level is implemented using reference chains and union-find at the abstract machine level. Call your predicates `makec/2`, `combinec/3`, `findc/3` and `unionc/2` so that they do not clobber your solutions for (b).

Question 3. (40 marks) In this question, you'll use Prolog's built-in backtracking search, combined with a strategy called *iterative deepening* to search for a digital circuit that simulates a given Boolean function.

You should assume that the Boolean function you are supposed to build a circuit for has been consulted into Prolog's memory in the form of a clause:
`bool(Name,Arity,Outputs).`
where `Name` is the name of the function, `Arity` is the number of inputs that it has, and *Outputs* is a list of $2^{\texttt{Arity}}$ outputs, one for each possible setting of the inputs, in the order $00\cdots0$, $00\cdots1$, ..., $11\cdots1$. For example, a binary exclusive or would be registered with the clause:
`bool(xor,2,[0,1,1,0]).`

(a) (20 marks) Write a predicate, `generate/2`, which takes a natural number `N` as input in its first argument, and successively produces as output in its second argument the term representation of a digital circuit on `N` inputs, consisting of AND, OR and NOT gates. Do not use any other kind of gate. If a call to `generate/2` backtracks, `generate/2` should producing another digital circuit on `N` inputs. `generate/2` should therefore never fail, since there are infinitely many possible combinations of `N` inputs when no bound on the number of gates is provided.

The term representation of a digital circuit uses the terms `and/2`, `or/2` and `not/1` to represent the gates of a circuit. If the input to a gate is one of the inputs to the circuit, then there should be an uninstantiated variable in that argument position, which uniquely identifies the input, i.e., all inputs lines that share the same circuit input use the same variable. If the input to a gate is the output of another gate, then the term representing that other gate appears in the corresponding argument position. One implementation of XOR, for example, is:
`and(not(and(X,Y)),not(and(not(X),not(Y))))`
A simpler example would be the obvious implementation of NAND:
`not(and(X,Y))`

We want smaller circuits wherever possible, so you should use *iterative deepening* to control Prolog's depth-first search. In each cycle of iterative deepening, it performs normal depth-first search until a given depth bound is reached. Iterative deepening begins at the bound of 0 (an input line only), performs search to this bound (which at 0 involves only iterating through every input line), then increments its counter to 1, performs search to this bound, successively succeeding with every digital circuit of depth 1, and so on. Since the number of gates roughly corresponds to the depth of the term representation, this will yield smaller circuits before larger ones. Be sure not to return a depth-n solution when the depth bound is set to depth $n+1$ or higher.

You will need `=..` to build your terms. This converts a list of length n into an $n-1$-ary term:

```
?- Term =.. [and,X,Y].
Term = and(X,Y)?
```

(b) (18 marks) Write a predicate, `test/3`, which takes as input a Boolean function name (first argument), and an arity (second argument), and the term representation of a digital circuit (third argument). `test/3` succeeds if and only if the digital circuit produces the Boolean function. To determine this, you will have to test the circuit on every possible setting of its input lines. You will find `term_variables/2` very helpful — it returns a list of all of the variables in a term. You may assume that the ordering of inputs assumed by `bool/3` is the same as the ordering of variables returned by `term_variables/2`.

(c) (2 marks) Finally, write a predicate, `circuit/3`, which takes as input a Boolean function name (first argument), and an arity (second argument), and produces as output the term representation of an equivalent digital circuit (third argument). Use `generate/2` and `test/3`. Generate-and-test approaches are often the simplest to implement, but they are often very slow. You will probably notice this as you run your code on functions with larger arities.

Question 4. (20 marks) (No programming is required for this one - hand your solution in on paper). Given a non-empty set, S, of size $n > 0$, find a method for representing every non-empty subset of S using $n+1$-ary terms, such that, if $\bar{U}$ and $\bar{V}$ are the term representations of the subsets U and V, respectively, then:

- the unification, $\bar{U} \sqcup \bar{V}$, exists if and only if $U \cap V \neq \phi$; and
- the unification, $\bar{U} \sqcup \bar{V}$, when it exists, is the representation of $U \cap V$.

Hint: use shared variables.