Last name, NetID

____

CSC 372 Final Exam
Wednesday, April 14, 2014

READ THIS FIRST

Read this page now but do not turn this page until you are told to do so. Go ahead and fill in your last name and NetID in the box above.

This is a 100-minute exam with a total of 100 points of regular questions and an extra credit section.

The last five minutes of the exam is a "seatbelts required" period, to avoid distractions for those who are still working. If you finish before the "seatbelts required" period starts, you may turn in your exam and leave. If not, you must stay quietly seated—no packing up— until time is up for all.

You are allowed no reference materials whatsoever, aside from the sheet mentioned on Piazza.

If you have a question, raise your hand. We will come to you. DO NOT leave your seat.

If you have a question that can be safely resolved with a minor assumption, like the name of a function or the order of function arguments, state the assumption and proceed.

Feel free to use abbreviations.

Don't make a problem hard by assuming it needs to do more than is specifically stated in the write-up.

If you're stuck on a problem, please ask for a hint. Try to avoid leaving a problem completely blank—that's a sure zero.

It is better to put forth a solution that violates stated restrictions than to leave it blank—a solution with violations may still be worth partial credit.

When told to begin, double-check that your name is at the top of this page, and then **put your initials in the lower right hand corner of the top side of each sheet, checking to be sure you have all seven sheets.**

BE SURE to enter your NetID on the sign-out log when turning in your completed exam.

Problem 1: (6 points)

Cite three things about programming languages you learned by watching your classmates' video projects. Each of the three should be about a different language.

Note: If you were at last night's review session you can't cite the Python feature that was mentioned. (Please ask if you were at the review session and have a question about this restriction!)

Problem 2: (6 points)

Write the following simple Prolog predicates. There will be a half-point deduction for each occurrence of a singleton variable or failing to take full advantage of unification.

Restriction: No library predicates may be used other than `is/2`.

`last(?List,?Elem)` specifies the relationship that `Elem` is the last element of `List`, which is assumed to be non-empty.

`member(?Elem, ?List)` specifies the relationship that `Elem` is a member of `List`.

`length(?List, ?Len)` specifies the relationship that `List` is `Len` elements in length.

Problem 3: (8 points)

Write a Prolog predicate `insrel(+List, -WithRels)` that instantiates `WithRels` to a copy of `List` with one of the atoms <, >, and = inserted between each pair of values to reflect the relationship between those values. Assume all values are numbers. Examples:

```
?- insrel([5,3,7,7,0],L).
L = [5, >, 3, <, 7, =, 7, >, 0] .

?- insrel([10,10],L).
L = [10, =, 10] .

?- insrel([7],L).
L = [7] .

?- insrel([],L).
L = [].
```

Only the first result is of interest; don't worry about behavior if the user responds with a semicolon.

Note that <, >, and = are valid symbolic atoms and are therefore shown without quotes.

Hint: Write a helper predicate `which(+X,+Y,-Rel)` that works like this:

```
?- which(3,4,Op).
Op = < .
```

Problem 4: (7 points)

Write a Prolog predicate `alltails(+List, -T)` that first instantiates `T` to the tail of `List`. If an alternative is requested, it generates the tail of the tail of `List`, and so forth, thus generating "all the tails".

Restriction: Only "cons", unification, and recursive calls to `alltails` may be used. In particular, you can't use `append` or write your own version of `append` and use it.

```
?- alltails([10,20,30,40],T).
T = [20, 30, 40] ;
T = [30, 40] ;
T = [40] ;
T = [] ;
false.

?- alltails([x],T).
T = [] ;
false.

?- alltails([],T).
false.
```

Don't forget the restriction!

Problem 5: (10 points)

Write a Prolog predicate `pinch(+List, -Pair)` that instantiates `Pair` to a series of `pair/2` structures. The first `pair` structure has the first and last values of `List`. The second `pair` has the second and next-to-last values of `List`. And so forth. Example:

```
?- pinch([a,b,c,d,e,f],P).
P = pair(a, f) ;
P = pair(b, e) ;
P = pair(c, d) ;
false.
```

If a list has an odd number of values, the last `pair` has two copies of the middle element.

```
?- pinch([1,2,3],P).
P = pair(1, 3) ;
P = pair(2, 2) ;
false.

?- pinch([1],P).
P = pair(1, 1) ;
false.
```

`pinch` fails on an empty list.

```
?- pinch([],P).
false.
```

Restriction: You may not use any arithmetic in your solution.

You may assume the presence of `last(?List,?Elem)`.

Problem 6: (5 points)

Write a Prolog predicate `sumvals/0` that consolidates all `v/1` facts into a single fact that contains the sum of the terms of those facts. Assume all terms are numbers. There may be any number of `v/1` facts.

```
?- v(X).
X = 1 ;
X = 7 ;
X = 2.

?- sumvals.
true.

?- v(X).
X = 10.

?- sumvals.
true.

?- v(X).
X = 10.
```

If no `v/1` facts exist, `sumvals` creates the fact `v(0)`.

```
?- v(X).
false.

?- sumvals.
true.

?- v(X).
X = 0.
```

Problem 7: (12 points)

This is the problem you were told to expect that is like `connect` from assignment 8.

`path(+Start, +End, +Moves, -Path)` instantiates `Path` to a series of values from `Moves` that describe a path from `Start` to `End`, two points on a Cartesian plane. All alternatives are produced if requested. Each move can be used only once in a given path. Example:

```
?- path(p(0,0), p(3,4), [m(4,4),m(3,3),m(0,1),m(-1,0)], Moves).
Moves = [m(4, 4), m(-1, 0)] ;
Moves = [m(3, 3), m(0, 1)] ;
Moves = [m(0, 1), m(3, 3)] ;
Moves = [m(-1, 0), m(4, 4)] ;
false.
```

The call asks for a path from `(0,0)` to `(3,4)` using some combination of movements, which are `m/2` structures that specify <u>relative</u> changes in X and Y, respectively. Here's the set of moves from above:

```
m(4,4)    Move right 4 and up 4.
m(3,3)    Move right 3 and up 3.
m(0,1)    Move up 1.
m(-1,0)   Move left 1.
```

In the following case there's both an empty solution and a solution that uses all three moves.

```
?- path(p(10,20), p(10,20), [m(-2,0),m(1,2),m(1,-2)], Moves).
Moves = [] ;
Moves = [m(-2, 0), m(1, 2), m(1, -2)] ;
Moves = [m(-2, 0), m(1, -2), m(1, 2)] ;
...four additional permutations aren't shown...
```

If no suitable sequence of moves exists, `path` fails.

```
?- path(p(0,0), p(1,1), [m(1,0),m(0,2),m(1,0)], Moves).
false.
```

As the preceding example demonstrates, a successful series of movements must terminate at the destination, not merely pass through it.

Problem 8: (10 points)

Using the parsing method supported by Prolog's grammar rule notation write a predicate `ptime(+Spec,-Mins)` that parses atoms that represent time durations, like `'10m'`, `'5h'` and `'10:20'`, and instantiates `Mins` to the number of minutes represented by `Spec`.

Durations will either be a number (an integer >= 0) followed by "`m`" or "`h`", or two numbers separated by a colon. `'10m'` is ten minutes; `'2h'` is two hours—120 minutes. `'2:30'` is two hours and thirty minutes—150 minutes. Something like `'1:2000'` is valid, too (2,060 minutes).

```
?- ptime('10m',Mins).
Mins = 10.

?- ptime('2h',Mins).
Mins = 120.

?- ptime('2:30',Mins).
Mins = 150.

?- ptime('2:3',Mins).  % not required to be '2:03', for example
Mins = 123.
```

Assume you have a grammar rule `int(N) --> ...` that recognizes a non-negative integer and instantiates `N` to the recognized value. Example:

```
?- int(I,['2','3'],[]).
I = 23 .

?- int(I,['2','3',x],Left).
I = 2,
Left = ['3', x] ;
I = 23,
Left = [x] ;
false.
```

Problem 9: (7 points)

Write a **Haskell** function `ckconn` that takes a list similar to that used by a8's `connect.pl` and returns `True` or `False`, depending on whether the exact sequence and orientation of cables represents a valid connection.

Example:

```
> ckconn 'm' [('f',10,'m'), ('f',7,'m')] 'f'
True
```

In contrast to the Prolog version, the list of cables appears between the left and right endpoints. `ckconn` does not check the length of the configuration. Note that cables are specified with three-tuples, not lists.

Note that `ckconn` checks the list as-is. It doesn't consider alternatives in any way. For example, simply flipping the last cable in the configuration above produces `False`:

```
> ckconn 'm' [('f',10,'m'), ('m',7,'f')] 'f'
False
```

Any number of cables may be specified. If no cables are specified, `ckconn` returns `False`.

```
> ckconn 'm' [] 'f'
False
```

Your solution may be recursive or not; your choice!

Important: Include a type declaration for ckconn. I'll get you started:

```
ckconn :: Char ->
```

Problem 10: (7 points)

Write a **Haskell** function `connlen` that takes a list of the same type as `ckconn` and **prints** either the length of the configuration, like "`25 feet`" or "`nope`", if the configuration is not valid.

```
> connlen 'm' [('f',10,'m'), ('f',7,'m')] 'f'
17 feet

> connlen 'm' [('f',10,'m'), ('f',7,'m')] 'm'
nope
```

Restriction: Like on assignment 2, your solution for `connlen` must be non-recursive.

You may use assume a working `ckconn` from the previous problem and use it in `connlen`.

Remember: `connlen` produces output; its final result has type `IO ()`. (Not `String`, for example.)

Problem 11: (14 points)

Write a **Ruby program** `shuffile.rb` that reads lines from standard input, shuffles them, and then writes them to standard output.

`shuffile` requires one command line argument, `-N`, which specifies that lines are to be handled in blocks of N lines. Assume that standard input consists of a multiple of N lines.

The UNIX utility `seq` can be used to output a sequence of numbers. We'll use it to demonstrate `shuffile`'s behavior. `seq 6` outputs the first six integers:

```
% seq 6
1
2
3
4
5
6
```

With the argument `-1` the input lines are shuffled just like you might shuffle a deck of cards:

```
% seq 6 | ruby shuffile.rb -1
5
6
3
1
4
2
```

With the argument `-2` the input lines are treated as pairs—the first and second lines are kept together, as are the third and fourth, and the fifth and sixth. With the output of `seq 6` as input, the line "3" will always be followed by the line "4", for example.

```
% seq 6 | ruby shuffile.rb -2
3
4
1
2
5
6
```

There are only two possible results of `seq 6 | ruby shuffile.rb -3`. Here's one of them:

```
% seq 6 | ruby shuffile.rb -3
4
5
6
1
2
3
```

There must be exactly one command line argument and it must be of the form `-N`. If not, "`Oops!`" is printed and `exit(1)` is called:

```
% seq 6 | ruby shuffile.rb
Oops!

% seq 6 | ruby shuffile.rb 3
Oops!

% seq 6 | ruby shuffile.rb -2 3
Oops!
```

All the examples above use a six line input but `shuffile.rb` should be able to handle any number of lines of any length with arbitrary content, not just numbers.

Important: Note that there is an `Array.shuffle` method:

```
>> ["please","shuffle","me","up"].shuffle
=> ["up", "please", "shuffle", "me"]
```

Problem 12: (8 points, one point each)

(a) Who founded UA's CS department, and in what year?

(b) Why are Prolog warnings about singleton variables worth paying attention to?

(c) What's the fundamental difference between predicates like `member/2` and `append/3` in Prolog and their superficial analogs in other languages?

(d) Draw the box for Prolog's four-port model and label the ports.

(e) "cons" lists are found in many languages that make use of recursion. What is it about cons lists that makes them well-suited for recursive functions?

(f) Consider this claim: Prolog's grammar rule notation, like `sentence --> article, noun, verb`, is an example of syntactic sugar. Present an argument that either supports that claim or refutes it.

(g) Write a simple **Haskell function** and show an example of using a partial application of that function.

(h) What's a very good reason that `ckconn` above uses tuples instead of lists to represent the cables?

Extra Credit Section (½ point each unless otherwise noted)

(a) What movie inspired the "Original Thought" bonus?

(b) whm came to graduate school here at UA specifically to join a research team developing a programming language. What was the language?

(c) To what current CS faculty member does whm attribute the following quotation?
"When you come to a problem you can lean forward and type, or you can sit back and think."

(d) The Prolog 1000 is a compilation of applications written in Prolog and related languages. What's a little odd about it?

(e) Jean Ichbiah, Ada's designer, was said to have once predicted that in ten years only two programming languages would remain in use. Ada was one of the languages. What was the other?

(f) Cite a significant contribution to computer science made by Grady Booch.

(g) What's a language that has/had an "arithmetic if", one form of which looks like this:

```
IF (I-J) 100, 110, 120
```

(h) In terms of creators, what do `vim` and Java have in common?

(i) Market research has determined that the course title *Comparative Programming Languages* is dry and unappealing. Think up a new name for 372. It needn't be less dry but should be funny!

(j) whm would like to recycle a8's `buy.pl` in a future semester but needs mores `dontmix` facts with an element of humor. What's another pair he could use?

(k) How many students earned the "close reading" bonus on assignment 8?