The Usual Stuff


About the if-then-else structure (->) and disjunction (;)

The rules about using if-then-else and disjunction are the same as for assignment 8.

once/1

Various tests run by the tester use the higher-order predicate once(Goal), which limits Goal to producing one result. Example:

?- once(nth0(Pos, [a,b,a,c,a], a)).
    Pos = 0.

Potential alternatives that don't materialize

The tester is not sensitive to potential alternatives that don't materialize. Consider this behavior for last, from the first problem below.

?- last([1,2,3],X).
    X = 3 ;
    false.

It prompts because somewhere there's at least one more clause that can be tried. The tester won't distinguish between the above behavior and the following behavior:

?- last([1,2,3],X).
    X = 3.

Problem 1. (5 points)  append.pl

Note: This problem has restrictions!

The purpose of this problem is to help you see that append/3 can be used for lots more than concatenating two lists. You are to write several simple predicates that heed this restriction: the only predicates you can use are length/2 and append/3. For some predicates, like head, a single append goal will be all you need:

    head(List,Elem) :- append(...).

Additionally, you may use the [E1, E2, ..., EN] list syntax, like in append([X,Y],[],Z), but you may not use the [E1, E2, ..., EN | Tail] notation, introduced on slide 192.

Here are the predicates you are to implement:
head(?List, ?Elem) expresses the relationship that the first element of List is Elem. It fails if the list is empty.

?- head([a,b,c],H).
H = a.

?- head([a,b,c],x).
false.

last(?List, ?Elem) expresses the relationship that the last element of List is Elem. It fails if the list is empty.

?- last([1,2,3],X).
X = 3 ;
false.

?- last(L,4).
L = [4] ;
L = [G2199, 4] ;
L = [G2199, G2205, 4] ;
...keeps going...

init(?List, ?Init) expresses the relationship that Init is all but the last element of List. It fails if List is empty.

?- init([a,b,c,d],I).
I = [a, b, c] ;
false.

tail(?List, ?Tail) expresses the relationship that Tail is all but the first element of List. It fails if List is empty.

?- tail([a,b,c],T).
T = [b, c].

min2(+List) fails if List is not at least two elements long.

?- min2([1]).
false.

?- min2([1,2]).
true.

mem(?Elem, ?List) behaves just like the built-in member/2:

?- mem(2,[1,2,3]).
true ;
false.

?- mem(E,[1,2,3]).
E = 1 ;
E = 2 ;
E = 3 ;
false.

contains(+List,?SubList) expresses the relationship that List contains SubList.
?- contains([1,2,3,4,5],[3,4,5]).
    true ;
false.

?- contains([1,2,3,4,5],[10,20]).
    false.

?- contains([1,2,3],S), S \== [].
    S = [1] ;
    S = [1, 2] ;
    S = [1, 2, 3] ;
    S = [2] ;
    S = [2, 3] ;
    S = [3] ;
    false.

firstlast(?List,?FL) expresses the relationship that FL is a list containing the first and last elements of List. It fails if List is empty.

?- firstlast(L,[1,2]).
    L = [1, 2] ;
    L = [1, _G3410, 2] ;
    L = [1, _G3410, _G3416, 2] ;
    ...keeps going...

halves(?List, ?First, ?Last) expresses the relationship that the first half of List is First and the second half is Last. It fails if the length of List is odd.

?- halves([1,2,3,4],F,S).
    F = [1, 2],
    S = [3, 4] ;
    false.

?- halves([1,2,3,4,5],F,S).
    false.

?- halves([],F,S).
    F = S, S = [] ;
    false.

?- halves(L,[a,b],S).
    L = [a, b, _G4635, _G4638],
    S = [_G4635, _G4638].

Remember the restriction: you may only use append and length in these predicates!

Testing note

Use a9/t append.pl -t PREDICATE to test an individual predicate. As you've seen before, the -t follows the problem name.
Problem 2. (2 points) middle.pl

Write a predicate middle(+List,+N,?Middle) that expresses the relationship that the middle N elements of List are Middle.

Restriction: Just like problem 1, you may only use append and length in this predicate.

?- middle([a,b,c,d,e],3,M).
M = [b, c, d] ;
false.

?- middle([a,b,c,d,e],5,M).
M = [a, b, c, d, e] .

?- middle([a,b,c,d,e],2,M).
false.

?- middle([a,b,c,d,e,f],2,M).
M = [c, d] ;
false.

?- middle([a,b,c],1,M).
M = [b] ;
false.

As the above examples imply, an even-length list has an even-length middle, and an odd-length list has an odd-length middle.

Assume that N (the length of the middle) is greater than zero.

Don't forget the restriction!

Problem 3. (2 points) splits.pl

This problem reprises splits.hs from assignment 3. In Prolog it is to be a predicate splits(+List,-Split) that unifies Split with each "split" of List in turn. Example:

?- splits([1,2,3],S).
S = [1]/[2, 3] ;
S = [1, 2]/[3] ;
false.

Note that Split is not an atom. It is a structure with the functor /. Observe:

?- splits([1,2,3], A/B).
A = [1],
B = [2, 3] ;
A = [1, 2],
false.

Here are additional examples. Note that splitting a list with less than two elements fails.
?- splits([],S).
false.

?- splits([1],S).
false.

?- splits([1,2],S).
S = [1]/[2] ;
false.

?- atom_chars('splits',Chars), splits(Chars,S).
Chars = [s, p, l, i, t, s],
S = [s]/[p, l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p]/[l, i, t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i]/[t, s] ;
Chars = [s, p, l, i, t, s],
S = [s, p, l, i, t]/[s] ;
false.

My solution uses only two predicates: append and \=.

Problem 4. (3 points) posints.pl

Write a predicate posints(+List) that succeeds iff all elements in List are positive (>0) integers.

Restriction: Your solution must not be recursive!

?- posints([1,2,3,4,5]).
true.

?- posints([1,2,3,-4,5]).
false.

?- posints([1,xyz,3,4,5]).
false.

?- posints([1]).
true.

I hope you'll ponder this one for a little while but here's a hint:
http://www.cs.arizona.edu/classes/cs372/spring16/a9/posints-hint.html

Problem 5. (3 points) repl.pl

Write a predicate repl(?E, +N, ?R) that unifies R with a list that is N replications of E. If N is less than 0, repl fails.

?- repl(x,5,L).
L = [x, x, x, x, x].
?- repl(1,3,[1,1,1]).
true.

?- repl(X,2,L), X=7.
X = 7,
L = [7, 7].

?- repl(a,0,X).
X = [].

?- repl(a,-1,X).
false.

**Problem 6. (3 points) pick.pl**

Write a predicate `pick(+From, +Positions, -Picked)` that unifies `Picked` with an atom consisting of the characters in `From` at the zero-based, non-negative positions in `Positions`.

?- pick('testing', [0,6], S).
S = tg.

?- pick('testing', [1,1,1], S).
S = eee.

?- pick('testing', [10,2,4], S).
S = si.

?- between(0,6,P), P2 is P+1, pick('testing', [P,P2], S), writeln(S), fail.
te
es
ti
in
ng
g
false.

?- pick('testing', [], S).
S = ''. 

If a position is out of bounds, it is silently ignored. My solution uses `atom_chars`, `findall`, `member`, and `nth0`.

**Problem 7. (5 points) polyperim.pl**

Write a predicate `polyperim(+Vertices,-Perim)` that unifies `Perim` with the perimeter of the polygon described by the sequence of Cartesian points in `Vertices`, a list of `pt` structures.

?- polyperim([pt(0,0),pt(3,4),pt(0,4),pt(0,0)],Perim).
Perim = 12.0.

?- polyperim([pt(0,0),pt(0,1),pt(1,1),pt(1,0),pt(0,0)],Perim).
Perim = 4.0.
There is no upper bound on the number of points but at least four points are required, so that the minimal path describes a triangle. (Think of it as ABCA, with the final A "closing" the path.) If less than four points are specified, `polyperim` fails with a message:

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4)],Perim).
At least a four-point path is required.
false.
```

The last point must be the same as the first. If not, `polyperim` fails with a message:

```
?- polyperim([pt(0,0),pt(3,4),pt(0,4),pt(0,1)],Perim).
Path is not closed.
false.
```

Note: check first for the minimum number of points and then a closed path.

This is not a course on geometric algorithms so keep things simple! Calculate the perimeter by simply summing the lengths of all the sides; don't worry about intersecting sides, coincident vertices, etc.

Be sure that `polyperim` produces only one result.

Problem 8. (14 points) `switched.pl`

This problem is a reprise of `switched.rb` from assignment 6. `a9/births.pl` has a subset of the baby name data, represented as facts. Here are the first five lines:

```
% head -5 a9/births.pl
births(1950,'Linda',f,80437).
births(1950,'Mary',f,65461).
births(1950,'Patricia',f,47942).
births(1950,'Barbara',f,41560).
births(1950,'Susan',f,38024).
```

`births.pl` holds data for only 1950-1959. Names with less than 70 births are not included.

Your task is to write a predicate `switched(+First,+Last)` that prints a table much like that produced by the Ruby version. To save a little typing, `switched` assumes that the years specified are in the 20th century.

```
?- switched(51,58).
Dana  1.19  1.20  1.26  1.29  1.00  0.79  0.67  0.64
Jackie 1.40  1.29  1.14  1.13  1.11  0.94  0.72  0.57
Kelly  4.23  2.74  3.73  2.10  2.32  1.77  0.98  0.51
Kim    2.58  1.82  1.47  1.08  0.61  0.30  0.17  0.12
Rene   1.43  1.32  1.15  1.24  1.13  0.88  0.87  0.89
Stacy  1.06  0.81  0.62  0.47  0.44  0.36  0.29  0.21
Tracy  1.51  1.14  1.02  0.73  0.56  0.55  0.59  0.59
true.
```

If no names are found, `switched` isn't very smart; it goes ahead and prints the header row:
?- switched(52,53).
    1952  1953
true.

If you want to make your switched smarter, that's fine—I won't test with any spans that produce no names. Also, I'll only test with spans where the first year is less than the last year.

Names are left-justified in a ten-wide field. Below is a format call that does that. Note that the dollar sign is included only to clearly mark the end of the output.

?- format("~w~t~10|\$", 'David').
   David $  
true.

Outputting the ratios is a little more complicated. I use sformat, like this:

?- sformat(Out, '~t~2f~6|', 2.32754), write(Out).
   2.33
Out = "  2.33".

The call above instantiates Out to a six-character string (a list of integers that are character codes) and write(Out) outputs it.

See help(format/2) if you're curious about the details of using ~t but the essence is that you can use ~N| to specify that a field extend to column N, and then put a ~t on the left, right, or both sides of a specifier like ~w or ~f to get right, left, or center justification, respectively.

To consult a9/births.pl when you consult switched.pl, put the following line in your switched.pl.

:-[a9/births].

As mentioned in the assignment 8 write-up for queries.pl, that construction, :- followed by a query, causes the query to be executed when the file is consulted.

Note that :-[a9/births.pl]. fails with an error. For an extra point on this assignment, add a note to observations.txt with a speculative but sound explanation of why [a9/births] works but [a9/births.pl] doesn't. No Googling, etc., please!

You might also use that :- ... mechanism to cause a couple of tests to be run when the file is loaded. Below I define test/0 to do a couple of switched queries, putting a line of dashes between them. I then invoke it with :-test.

    test :- switched(51,58), writeln('-----'), switched(51,52).
    :-test.

That invocation of test must follow the definition of switched and consulting a9/births.pl.

You'll see that with swipl switched.pl the output from test appears before "Welcome to SWI-Prolog..."

Be sure to comment out lines like :-test. before turning in your solution. (That output will cause
a9/tester failures, too, as you'd expect.

My Prolog solution is significantly smaller than my Ruby solution but it's easy to get sideways on this problem if you don't come up with a good set of helper predicates. I suggest that you give it a try on your own but if it starts to get ugly, http://www.cs.arizona.edu/classes/cs372/spring16/a9/switched-hints.pdf shows how I broke it down. The points assigned to this problem are based on the assumption that you will take a look at the hints. Without the hints, and based on your current level of Prolog knowledge, I might assign this problem 25-30 points.

Problem 9. (14 points) iz.pl

In this problem you are to write a predicate iz/2 that evaluates expressions involving atoms and a set of operators and functions. Let's start with some examples:

?- S iz abc+xyz. % + concatenates two atoms.
   S = abcxyz.

?- S iz (ab + cd)*2. % *N produces N replications of the atom.
   S = abcdabcd.

?- S iz -cat*3. % - is a unary operator that produces a reversed copy of the atom.
   S = tactactac.

?- S iz -cat+dog.
   S = tacdog.

?- S iz abcde / 2. % / N produces the first N characters of the atom.
   S = ab.

?- S iz abcde / -3. % If N is negative, / produces last N characters
   S = cde.

?- N is 3-5, S iz (-testing)/N.
   N = -2,
   S = et.

The functions len and wrap are also supported. len(E) evaluates to an atom (not a number!) that represents the length of E.

?- N iz len(abc*15).
   N = '45'.

?- N iz len(len(abc*15)).
   N = '2'.

wrap adds characters to both ends of its argument. If wrap is called with two arguments, the second argument is concatenated on both ends of the string:

?- S iz wrap(abc, ==).
   S = '==abc=='.

?- S iz wrap(wrap(abc, ==), '* * 3).
   S = '*****abc*****'.
If \texttt{wrap} is called with three arguments, the second and third arguments are concatenated to the left and right ends of the string, respectively:

\begin{verbatim}
?- S is wrap(abc, '(, ')'.
S = '(abc)'.

?- S is wrap(abc, '>' * 2, '<' * 3).
S = '>>abc<<'.
\end{verbatim}

It is important to understand that \texttt{len(xy), wrap(abc, ==), and wrap(abc, '(, ')')}
are simply structures. If \texttt{iz} encounters a two-term structure whose functor is \texttt{wrap} (like \texttt{wrap(abc, ==)}) its value is the concatenation of the second term, the first term, and the second term. \texttt{iz} evaluates \texttt{len} and \texttt{wrap} like \texttt{is} evaluates \texttt{random} and \texttt{sqrt}.

The atoms \texttt{comma, dollar, dot, and space} do not evaluate to themselves with \texttt{iz} but instead evaluate to the atoms '\', '$', '.', and ' ', respectively. (They are similar to \texttt{e} and \texttt{pi} in arithmetic expressions evaluated with \texttt{is/2}.) In the following examples note that \texttt{swipl} (not me!) is adding some additional wrapping on the comma and the dollar sign that stand alone. That adornment disappears when those characters are used in combination with others.

\begin{verbatim}
?- S is comma.
S = (',').

?- S is dollar.
S = ($).

?- S is dot.
S = '.'.

?- S is space.
S = ' '.

?- S is comma+dollar*2+space+dot*3.
S = ',,$$ ...'.

?- S is wrap(wrap(space+comma+space,dot),dollar).
S = '$. ,.$'.

?- S is dollarcomma+dot*3.
S = dollarcomma+dot*3.
\end{verbatim}

The final example above demonstrates that these four special atoms don't have special meaning if they appear in a larger atom.

Here is a summary for \texttt{iz/2}:

\begin{verbatim}
-Atom iz +Expr unifies Atom with the result of evaluating Expr, a structure representing a calculation involving atoms. The operators (functors) are as follows:

\texttt{E1+E2} Concatenates the atoms produced by evaluating \texttt{E1} and \texttt{E2} with \texttt{iz}.

\texttt{E*N} Concatenates \texttt{E} (evaluated with \texttt{iz}) with itself \texttt{N} times. (Just like Ruby.) \texttt{N} is a term that can be evaluated with \texttt{is/2} (repeat, \texttt{ig/2}). Assume \texttt{N} \texttt{>= 0}.
\end{verbatim}
E/N Produces the first (last) N characters of E if N is greater than (less than) 0. If N is zero, an empty atom is produced. (An empty atom is shown as two single quotes with nothing between them.) N is a term that can be evaluated with is/2. The behavior is undefined if abs(N) is greater than the length of E.

-E Produces reversed E.

len(E) Produces an atom, not a number, that represents the length of E.

wrap(E1,E2) Produces E2+E1+E2.

wrap(E1,E2,E3) Produces E2+E1+E3.

The behavior of iz is undefined for all cases not covered by the above. Things like 1+2, abc*xyz, etc., simply won’t be tested.

Here are some cases that demonstrate that the right-hand operand of * and / can be an arithmetic expression:

?- X = 2, Y = 3, S iz 'ab' * (X+Y*3).
X = 2,
Y = 3,
S = ababababababababababababab.

?- S = '0123456789', R iz S + -S, End iz R / -(2+3).
S = '0123456789',
R = '01234567899876543210',
End = '43210'.

Implementation notes

One of the goals of this problem is to reinforce the idea that for an expression like -a+b*3 Prolog creates a tree structure that reflects the precedence and associativity of the operators. is/2 evaluates a tree as an arithmetic expression. iz/2 evaluates a tree as a "string expression". Note the contrast when the same tree is evaluated by is and iz:

?- X is pi + e*3. % using is
X = 11.296438138966929.

?- X iz pi + e*3. % using iz
X = pieee.

It's important to understand Prolog itself parses the expression and builds a corresponding structure that takes operator precedence into account. display/1 shows the tree:

?- display(pi + e*3).
+(pi,*(e,3))
true.

Processing of syntactically invalid expressions like abc + + xyz never proceeds as far as a call to iz.

Below is some code to get you started. It fully implements the + operation.
% cat a9/iz0.pl
:-op(700, xfx, iz). % Declares iz to be an infix operator. (Remember that leading :-
% causes the code to be evaluated as a goal, not consulted as a
% clause.)

A iz A :- atom(A), !.
R iz E1+E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).

Here are examples that use the version of iz just above.

?- [a9/iz0]. % Note: no ".pl"
true.
?- X iz abc+def.
X = abcdef.

?- X iz abc+def, Y iz X+'...'+X.
X = abcdef,
Y = 'abcdef...abcdef'.

?- X iz a+b+(c+(de+fg)+hij+k)+l.
X = abcdgfhijkl.

Let's look at the code provided above. Let's first talk about the second clause:

R iz E1 + E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).

The first thing you may notice is that the head (R iz E1 + E2) doesn't match the
functor(term,term,...) form we've always seen. That's because the op(700, xfx, iz) call above lets us use iz as an infix operator, and that applies in both the head and body of a rule. Here is a completely equivalent version that doesn't take advantage of the op specification:

iz(R,E1+E2) :- iz(R1,E1), iz(R2,E2), atom_concat(R1, R2, R).

With that equivalence explained, let's focus on the version in a9/iz0.pl, which uses the infix form:

R iz E1 + E2 :- R1 iz E1, R2 iz E2, atom_concat(R1, R2, R).

Consider the goal "X iz ab+cd". That goal unifies with the head of the above rule like this:

?- (X iz ab+cd) = (R iz E1+E2).
X = R,
E1 = ab,
E2 = cd.

If E1 is instantiated to ab then the first goal in the body of the rule would be equivalent to R1 iz ab. That goal unifies with the head of this rule:

A iz A :- atom(A), !.

This rule represents the base case for the recursive evaluation performed by iz. It says, "If A is an atom then the result of evaluating that atom is A." Another way to read it is, "An atom evaluates to itself." The result is that "R1 iz ab" instantiates R1 to ab. Note that atoms always lie at the leaves of the expression's tree.
It's important to recognize that because the \( iz(R, E1+E2) \) rule is recursive, it'll handle every tree composed of + operations.

Here are the heads (and necks) for all the iz rules that I've got in my solution:

\[
\begin{align*}
R & \text{ iz } E1+E2 : - \\
R & \text{ iz } E1*\text{NumExpr} : - \\
R & \text{ iz } -E : - \\
R & \text{ iz } E1 / \text{NumExpr} : - \\
R & \text{ iz } \text{len}(E) : - \\
R & \text{ iz } \text{wrap}(E,\text{Wrap}) : - \\
R & \text{ iz } \text{wrap}(E,\text{First},\text{Last}) : -
\end{align*}
\]

Via recursion, those heads handle all possible combinations of operations, like this one:

\[
?- \text{ X \text{ iz} } \text{wrap}((-\text{ab}+\text{cde*4})/\text{6+xyz}), '\text{Start}'>','<'+\text{(end*3+zz*2)}). \\
\text{X} = '\text{Start}>\text{edcedcxyz}<\text{endendendzzzz}'.
\]

**If you find yourself wanting to add a bunch of rules like**

\[
\begin{align*}
R & \text{ iz } (E1+E2+E2) : - ... \\
R & \text{ iz } (E1*E2) \text{ / } \text{NumExpr} : - ...
\end{align*}
\]

then STOP! You're not recognizing that a set of rules based on the heads above will cover ALL the operations.

*atomic_list_concat*

iz0.pl above uses concat_atom but atomic_list_concat(+List, -Atom) is a more general predicate:

\[
?- \text{atomic_list_concat}([\text{ab,c,def,ghij}], S). \\
S = \text{abcdefghij}.
\]

We'll later see in the slides that it can be used to split atoms, too.

**A minor parsing problem**

On the slides I fail to mention that Prolog requires some sort of separation between operators. Consider this:

\[
?- \text{X \text{ iz} } \text{abc+-abc.} \quad \% \text{No space between + and -} \\
\text{ERROR: Syntax error: Operator expected} \\
\text{ERROR: X \text{ iz} abc} \\
\text{ERROR: ** here **}
\]
ERROR: +-abc .

To make it work, add a space or parenthesize:

?- X iz abc+ -abc.
X = abccba.

?- X iz abc+(-abc).
X = abccba.

This issue only arises with unary operators, of course.

Problem 10. Extra Credit iz-extra.txt

For up to three points of extra credit, devise and implement three more operations for iz to handle. They can be operators or functions (like len and wrap). Perhaps use op/3 to define a new operator!

Include the required clauses in iz.pl and create iz-extra.txt, a plain text file that talks about your creation(s) and also shows them in action with swipl.

Problem 11. Extra Credit observations.txt

Submit a plain text file named observations.txt with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

Hours: 6
Hours: 3-4.5
Hours: ~8

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "Hours:". There must be only one "Hours:" line in observations.txt. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours: ", in either upper- or lower-case.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use a9/turnin to submit your work.

Just like on assignment 8 I'll use my plsize script to show you the "sizes" of my solutions. (Recall that the script counts left parentheses and commas, which I claim is a reasonable proxy for program size for Prolog source code.)
Here's what I see as of press time, with comments stripped:

$ a9/plsize $(grep -v txt a9/delivs)
append.pl: 65
middle.pl: 19
splits.pl: 7
repl.pl: 15
posints.pl: 9
pick.pl: 19
polyperim.pl: 53
switched.pl: 106
iz.pl: 76

Miscellaneous

Aside from -> and ;, and any per-problem restrictions notwithstanding, you can use any elements of Prolog that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Prolog slides 1-185.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) In Prolog, % is comment to end of line. Comments with /* ... */ , just like in Java, are supported, too.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 5 to 7 hours to complete this assignment.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put seven hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached seven hours. Give us a chance to speed you up!

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.