Assignments
for
CoSc 450, Programming Paradigms

J. Stanley Warford

December 8, 2021

Chapter, Section, and Exercise numbers in these assignments refer to one of four texts for this course:

- Hailperin: *Concrete Abstractions*, Max Hailperin, Barbara Kaiser, Karl Knight, Brooks/Cole, 1999.
- Bratko: *PROLOG Programming for Artificial Intelligence*, Ivan Bratko, Addison-Wesley, fourth edition, 2012.
- Ben-Ari: *Principles of Concurrent and and Distributed Programming*, Second Edition, Mordechai Ben-Ari, Addison-Wesley, 2006.
- Sestoft: *Java Precisely*, Third Edition, Peter Sestoft, The MIT Press, 2016.

1. Study Hailperin, Chapter 1.
2. Do Hailperin Exercises, 1.4b, 1.5, 1.9, 1.10.
   Insert all six definitions named `candy-temperature`, `tax`, `half-turn`, `quarter-turn-left`, `side-by-sid` and `pinwheel`, before the definition of the quilt patterns. The first three lines of the Definitions should be

   ```
   ;; Name
   ;; Date
   ;; Assignment
   ```

   and each exercise should be identified as

   ```
   ;; Exercise n.m
   ```

   Save the Definitions in a file named `a01.rkt`, where `.rkt` is the standard file extension for Dr. Racket source programs. Note the lowercase `a`. Hand in `a01.rkt` electronically per the instructions for your course.

Exercises 2 – 6 are programming problems. Submit them in a single file named `a02.rkt` electronically per the instructions for your course. Exercises 7 – 9 are written exercises. Type your solutions in a word processor or text editor and submit them in a single pdf file named `a02written.pdf` electronically per the instructions for your course.

1. Study Hailperin, Chapters 2, 3.

2. Do Hailperin, programming Exercise 2.1.

3. Do Hailperin, programming Exercise 2.8(c).
   Use your function power from Exercise 2.1.

4. Do Hailperin, programming Exercise 2.10.
   Name your function `num-odd-digits`.

5. Do Hailperin, programming Exercise 2.15.
   Be sure to include pinwheel with your other functions from Assignment 1. Error in book: `pin-wheel` should be spelled `pinwheel`.

6. Do Hailperin, programming Exercise 3.14(b).

7. Do Hailperin, written Exercise 2.18.
   Use the proof style as given in class.

8. Do Hailperin, written Exercise 3.14(a).
   The question is asking why the procedure generates a recursive process as opposed to an iterative (tail-recursive) process.

9. Do Hailperin, written Exercise 3.15.

Exercises 2 – 6 are programming problems. Submit them in a single file named `a03.rkt` electronically per the instructions for your course.

1. Study Hailperin, Sections 4.1, 4.3.

2. Do Hailperin, Exercise 3.10.

3. Do Hailperin, Exercise 3.11.

4. Do Hailperin, Exercise 3.12.
   Name the function `first-survivor-after` so that

   ```
   (first-survivor-after 0 8)
   ```

   returns 4, and

   ```
   (first-survivor-after (first-survivor-after 0 8) 8)
   ```

   returns 7.

5. Do Hailperin, Exercise 4.9.
   The test program will assume the vertices are entered in order bottom left, bottom right, and top.
   The call

   ```
   (triangle -1 -.75 1 -.75 0 1)
   ```

   should return an equilateral triangle.

6. Do Hailperin, Exercise 4.10.

Exercises 2 – 5 are programming problems. Submit them in a single file named `a04.rkt` electronically per the instructions for your course.

1. Study Hailperin, Chapter 5.

2. Do Hailperin, Exercise 2.12.
   Name your function `exponent-of-two` with one parameter.

3. Do Hailperin, Exercise 5.1.
   Your version of `together-copies-of` must have three parameters as does the original version. It should call a function with four parameters using the usual technique for making a tail-recursive function.

4. Do Hailperin, Exercise 5.4.
   You must supply a boolean expression to `num-digits-in-satisfying`, and the expression must be the shortest possible boolean expression.

5. Do Hailperin, Exercise 5.7.

1. Study Hailperin, Chapter 6.

2. Do Hailperin Exercise 6.7.
   Modify the exercise to write a three-pile game of Nim. The representation described in the book for a two-pile game of Nim is a single integer of the form $2^n \times 3^m$ where n is the number of coins in the first pile and m is the number of coins in the second pile. For the three-pile game of Nim, the game state will be a single integer of the form $2^n \times 3^m \times 5^k$ where n is the number of coins in the first pile, m is the number of coins in the second pile, and k is the number of coins in the third pile. The game is still played with two opponents, human and computer.

   Notice that you must implement `exponent-of-in` as a function with two parameters, as it will be tested separately from the game.

   For your programming convenience, here is a link to the two-pile Nim game with the implementation as a two-digit number with the 10's place the number in the first pile and the 1's place the number in the second pile. Although this implementation restricts the number of items in each pile to be no greater than nine, your implementation with `exponent-of-in` will have no such restriction.

   http://www.cslab.pepperdine.edu/warford/cosc450/a05.rkt

   Incorporate your function in a complete Nim game and hand it in as file `a05.rkt`.

Exercises 2 – 5 are programming problems. Submit them in a single file named `a06.rkt` electronically per the instructions for your course.

1. Study Hailperin, Sections 7.1, 7.2, 7.3, 7.4, 7.5.

2. Do Hailperin, Exercise 7.8(f).
   Name your function `largest-in` with one parameter. If the list is empty, print an error message using the display function, not the error function. Your program should not crash if the list is empty.

3. Do Hailperin, Exercise 7.9(a).
   Return `#f` if the lists are not the same size and `#t` if they are both empty. Be sure to name your function `list-<` as specified by the problem.

4. Do Hailperin, Exercise 7.14.
   Name your function `my-map`.

5. Do Hailperin, Exercise 7.42.

Exercises 2 – 6 are programming problems. Submit them in a single file named `a07.rkt` electronically per the instructions for your course.

1. Study Hailperin, Sections 8.1, 8.2, 8.3.

2. Do Hailperin, Exercise 7.15.

3. Do Hailperin, Exercise 8.1.
   Output an error message using the display function (not the error function) if the tree is empty. Note that the tree is a binary *search* tree.

4. Do Hailperin, Exercise 8.4. Do not use `append`.

5. Do Hailperin, Exercise 8.6.
   The first parameter should be the number and the second parameter should be the binary search tree.

6. Do Problem Flatten.
   Procedure `flatten` should take a list and return a list with all the elements in the same order but with no nested parentheses. If the parameter is not a list, output an error message using the display function (not the error function). Here are some test cases for flatten.

```
> (flatten 'a)
Bug: parameter is not a list
> (flatten '())
()
> (flatten '(()))
()
> (flatten '(a))
(a)
> (flatten '(a b c))
(a b c)
> (flatten '(((a))))
(a)
> (flatten '(a b (c d (e (((f))) () g) h) i))
(a b c d e f g h i)
```

   You can write `flatten` with a single `cond` using functions `display`, `not`, `list?`, `null?`, and `append`, as well as the usual Scheme operators.

1. Study Bratko, Sections 1.1 to 2.4.

2. Do Bratko, programming Exercises 1.4, 1.5.
   For `grandchild( X, Y)`, `X` is the grandchild of `Y`, not the other way around. Similarly for aunt. Incorporate your relations into the program of Figure 1.8. Name your file `a08.pl`.

3. Do Bratko, written Exercises 1.7c, 2.1, 2.3.
   Type your solutions in a text editor or word processor. Hand in your solutions in a pdf file named `a08written.pdf`.

1. Study Bratko, Sections 2.5 to 3.2.

2. Do Bratko, written Exercises 2.6, 2.8, 2.11.
   Exercise 2.6(d) may have an error. Assume it terminates with a period, not a comma.
   Type your solutions in a text editor or word processor. Hand in your solutions in a pdf file named `a09written.pdf`.

3. Do Bratko, programming Exercise 3.1(a).
   Name your goal `delete_three` with `L` as the first argument and `L1` as the second argument. Your goal should simply fail if list `L` has less than three elements.

   ```
   ?- delete_three( [a,b,c,d,e], L1).

   L1 = [a,b]
   ```

4. Do Bratko, programming Exercise 3.1(b).
   Name your goal `delete_six` with `L` as the first argument and `L2` as the second argument. Your goal should simply fail if list `L` has less than six elements.

   ```
   ?- delete_six( [a,b,c,d,e,f,g,h], L2).

   L2 = [d,e]
   ```

5. Do Bratko, programming Exercise 3.2(a).
   Note that `last/2` is a built-in prolog predicate, but it is of the form `last( List, Item)` instead of `last( Item, List)`. Name your goal `my_last` with the form `my_last( List, Item)`.

   ```
   ?- my_last( [a,b,c], Item).

   Item = c
   ```

6. Do Bratko, programming Exercise 3.2(b).
   Name your goal `your_last` with the form `your_last( List, Item)`.

   ```
   ?- your_last( [a,b,c], Item).

   Item = c
   ```

   Hand in all four of your goals for 3.1 and 3.2 in a file named `a09.pl`.

Homework requirement:
From now on, use anonymous variables as described in Section 1.13. Points will be taken off for consult warnings because of singleton variables.

Exercises 2 – 5 are programming problems. Submit them in a single file named `a10.pl` electronically per the instructions for your course.

1. Study Bratko, Sections 3.3 to 3.4.

2. Do Bratko, Exercise 3.4.
   Name your predicate `my_reverse`, and do not use the built-in predicate `reverse`. The recursive rule should have two goals, one of which is `my_reverse`, but the other of which is `conc`.

3. Do Bratko, Exercise 3.5.

4. Do Bratko, Exercise 3.6.
   You can do it with only one rule with `conc`. Note that the name of the predicate is `shift`, *not* `my_shift`.

5. Do Bratko, Exercise 3.11.
   `flatten` is built-in to `gprolog`, so you must name your predicate `my_flatten`. Here are two queries to illustrate the two base cases.

```
?- my_flatten( [], X).

X = []

?- my_flatten( a, X).

X = [a]
```

   Note that `my_flatten/2` must produce a list, even if the first argument is not a list. This is an unusual problem, because the base cases must be placed *after* the single recursive rule. When you test your predicate you will probably get erroneous instantiations after the first correct unification. With the tools we have so far in the course, you cannot avoid the later erroneous results. Think about why you cannot.

```
?- my_flatten( [a,b,[c,d],[],[[[e]]],f], List).

List = [a,b,c,d,e,f]
```

This assignment is from the Third Edition of Bratko, Chapter 4. You may access the sections you need for this assignment here.

http://www.cslab.pepperdine.edu/warford/cosc450/Bratko-ch4-3rd-ed.pdf

1. Study Bratko, Third Edition, Sections 4.1 to 4.3.

   For the programming part of the assignment download this text file

   http://www.cslab.pepperdine.edu/warford/cosc450/a11.txt

   which you can modify and hand in. You will need to change the file extension from `a11.txt` to `all.pl`.

2. Do Bratko, Third Edition, Exercise 4.1.
   Hand in your solution as `/* */` style comments in `a11.pl`. Put your query on a separate line in the comment section as follows:

```
/*
query4.1(a), families without children
Put your query here.

query4.1(b), all employed children
Put your query here.

query4.1(c), employed wives and unemployed husbands
Put your query here.

query4.1(d), children whose parents differ in age by at least 15 years
Put your query here.
*/
```

   so I can copy and paste your queries to test them. Write queries, not rules.

   The `a11.pl` file has the facts of the family database and has been augmented with families for which the other queries can be tested. For example, 4.1(b) is a query for employed children, so families have been added with at least one child who is employed. Be sure to include whatever utilities you need for your queries to work.

   Here are some hints. For 4.1(a) and (b), the query is a single goal.

   For 4.1(d), first match `family` to instantiate variables `Husband`, `Wife`, and `Children`. Then, use `dateof birth` twice to instantiate `HusbandYOB` and `WifeYOB`. Here is the goal that will succeed if their ages differ by 15 years or more.

```
(HusbandYOB - WifeYOB >= 15
;
WifeYOB - HusbandYOB >= 15)
```

3. Do Bratko, Third Edition, Exercise 4.2.
   The query

```
?- twins( C1, C2)
```

should list all the twins. First, you should match `family` to get an instantiation of `Children`. Then, use the `del/3` predicate to get a list of children without `Child1`. `Child2` is a twin if he is a member of the smaller list and his date of birth is the same as that of `Child1`. You can use the builtin `member/2` predicate, but you should include our definition of `del/3` in your solution.

4. Do Bratko, Third Edition, Exercise 4.3.
   Here is an example query.

```
?- nth_member( 2, [a,b,c,d,e], X).

X = b
```

5. Do Problem Nondeterministic FSM.
   Write a Prolog program that implements the nondeterministic finite state machine (FSM) for integers according to Warford, Figure 7.17.

   http://www.cslab.pepperdine.edu/warford/cosc450/warford-fig-7-17.pdf

   The input alphabet should be the atoms `plus`, `minus`, `digit`. The states should be `si`, `sf`, and `sm` corresponding to I, F, and M in the figure. For example, the queries

```
accepts( si, [plus, digit]).
```

   and

```
accepts( si, [digit, digit]).
```

   should succeed and the query

```
accepts( si, [digit, plus, digit]).
```

   should fail. Note that `a11.pl` contains the `accepts/2` predicates for an arbitrary finite state machine, so you only need to write the facts for this specific machine.

Exercises 2 – 5 are programming problems. Submit them in a single file named `a12.pl` electronically per the instructions for your course.

1. Study Bratko, Chapter 5.

2. Do Bratko, Exercise 3.11.
   Make your solution deterministic. That is, after your predicate unifies with the flatten of the list, it should not give additional solutions that are not flattened. Remember to call your predicate `my_flatten` because `flatten/2` is built-in.

3. Do Bratko, Exercise 5.2.

4. Do Bratko, Exercise 5.5.

5. Do Bratko, Exercise 5.6.

Exercises 2 – 5 are programming problems. Submit them in a single file named `a13.pl` electronically per the instructions for your course.

1. Study Bratko, Sections 6.1 – 6.6, except 6.1.2.

2. Study the article, *Unification: A Multidisciplinary Survey*, Kevin Knight, ACM Computing Surveys, Vol 21, No. 1, March 1989.

   http://www.cslab.pepperdine.edu/warford/cosc450/Unification-Knight.pdf

3. Do Bratko, Exercise 6.3.
   Because `ground/1` is a builtin predicate, you must name your predicate `my_ground`. The following two queries should succeed

   ```
   ?- my_ground( w( x( a) ,y( b), z( c))).
   ?- my_ground( w( x( a) ,b , z( c))).
   ```

   and the following two queries should fail.

   ```
   ?- my_ground( w( x( a) ,y( B), z( c))).
   ?- my_ground( w( x( a) ,B, z( c))).
   ```

   should fail. An atomic term is grounded. A compound term is grounded if each of its arguments is grounded. For compound terms, use `=..` to access the argument list, and write another predicate to test each argument in the list.

4. Do Bratko Exercise 6.7.
   You must use `asserta/1` and `retract/1` and name your predicate `my_copy_term`. Here is a test of `my_copy_term`.

   ```
   ?- my_copy_term( abc(X, def(X), Y), C).

   C = abc(A,def(A),_)
   ```

   You can do this with one rule for `my_copy_term( Term, Copy)`. If you dynamically install a new predicate in the database with `Term` as its argument, then retract that predict with `Copy` as its argument, then `Copy` will be a copy of `Term`.

5. Do Bratko Exercise 6.8.
   Here is the specification of `powerset`.

   ```
   % powerset( Set, P)
   % P is a set of all the subsets of Set
   ```

   Here is a test of `powerset`.

   ```
   ?- powerset( [a,b,c], P).

   P = [[a,b,c],[a,b],[a,c],[a],[b,c],[b],[c],[]]
   ```

Write a predicate `subsets_with_backtracking` that generates the subsets with backtracking, then use `bagof` to collect them all into a list of lists. Here is the specification of `subsets_with_backtracking`.

```
% subsets_with_backtracking( Set, Subset)
% Subset is a subset of Set
```

For example, here is a sample run of `subsets_with_backtracking`.

```
?- subsets_with_backtracking( [a,b,c], Subset).

Subset = [a,b,c] ? ;

Subset = [a,b] ? ;

Subset = [a,c] ? ;

Subset = [a] ? ;

Subset = [b,c] ? ;

Subset = [b] ? ;

Subset = [c] ? ;

Subset = []
```

You can write `subsets_with_backtracking` with one base case fact and two rules, each of which has only one goal. The first rule expresses the fact that the set of all subsets that begin with `a` are the sets with `a` and all the subsets of `[b,c]`. The second rule expresses the fact that `Subset` is a subset of `[a,b,c]` if `Subset` is a subset of `[b,c]`.

Exercises 2 – 4 are programming problems. Submit them in a single file named `a14.pl` electronically per the instructions for your course. Exercise 5 is a written exercise. Submit it in a single pdf file named `a14written.pdf` electronically per the instructions for your course.

1. Study Bratko, Section 6.7.

2. Do Bratko, Exercise 6.11.
   Test your program with

   ```
   ?- see( 'myfile.pl'), findallterms( Term), seen.
   ```

   using this file

   http://www.cslab.pepperdine.edu/warford/cosc450/myfile.txt

   which should produce

   ```
   term(term1)
   term(term2)
   term(term3)
   abc
   def(1,2)
   ```

   If you query `findallterms/1` with `Term` instantiated, only those terms should be found. So, the query

   ```
   ?- see( 'myfile.pl'), findallterms( abc), seen.
   ```

   should produce

   ```
   abc
   ```

   The name of the file is `myfile.txt` to bypass security on download. Change the name to `myfile.pl` after you download it.

3. Do Bratko, Exercise 6.13.

   ```
   ?- starts( abc, 97).
   ```

   should succeed because 97 is the ascii value for the letter `a`, and

   ```
   ?- starts( abc, Ch).
   ```

   should give

   ```
   Ch = 97.
   ```

4. Do Bratko, Exercise 6.14.

5. Write a summary paper of Sections 1–5 of the article, *Unification: A Multidisciplinary Survey*.

   http://www.cslab.pepperdine.edu/warford/cosc450/Unification-Knight.pdf

   For each section, write a one paragraph description that (a) summarizes in your own words what the section is about, and (b) describes what you consider to be the most important or interesting points. Then, answer the following questions.

   1. Using Definitions 1.1, 1.2, and 1.3, is

   $f(x, g(y, b))$

   unifiable with

   $f(f(a), g(g(a, f(a)), b))$?

   If so, show the substitution that unifies them.

   2. Write the tree representation of the term

   $f(g(x), h(f(x)), i(f(x)))$

   3. Write the graph representation of the term

   $f(g(x), h(f(x)), i(f(x)))$

Write your solutions for the last two exercises using a text editor or word processor. Hand them in electronically as a PDF file named `a15written.pdf` per the instructions for your course. Note that you cannot simply change the file extension of a word processing document to `.pdf`. You must save the document as a PDF document.

1. Study Ben-Ari, Chapters 1, 2.

2. Do Ben-Ari, Exercise 2.1.

3. Do Ben-Ari, Exercise 2.2.
   p1, p2, p3 execute 10 times with a final execution of p1, which is the final test for termination of the loop. That is a total of 31 executions. Similarly, there are 31 statement executions in process q. Therefore, your scenario must be a list of 62 statement executions.

Write your solutions using a text editor or word processor. Hand them in electronically as a PDF file named

`a16written.pdf`

per the instructions for your course. Note that you cannot simply change the file extension of a word processing document to `.pdf`. You must save the document as a PDF document.

1. Do Ben-Ari, Exercise 2.3.
   p1, p2, p3 execute 10 times with a final execution of p1, which is the final test for termination of the loop. That is a total of 31 executions. Similarly, there are 31 statement executions in process q. Therefore, your scenario must be a list of 62 statement executions.

2. Do Ben-Ari, Exercise 2.4.

See the first 15 minutes of the Lecture 3.7 video for Programming Paradigms on iTunes U for a discussion of this assignment.

Write your solutions using a text editor or word processor. Hand them in electronically as a PDF file named `a17written.pdf` per the instructions for your course. Note that you cannot simply change the file extension of a word processing document to `.pdf`. You must save the document as a PDF document.

1. Do Ben-Ari, Exercise 2.5.
   Assume weakly fair scenarios for Algorithms Zero A and Zero B. Do *not* assume weakly fair scenarios for Algorithm Zero C.

   For all five parts of the problem, assume that $f(2) = 0$ and that $f(i) \neq 0$ for all $i \neq 2$. Your solution should be a sequence of p's and q's that produce an incorrect result. Here is a sequence of p's and q's for Algorithm Zero A.

   p1 (*found* $\leftarrow$ *false*), q1 (*found* $\leftarrow$ *false*), p2, p3 ($i \leftarrow 1$), p4 (*found* $\leftarrow$ *false*), p2, p3 ($i \leftarrow 2$), p4 (*found* $\leftarrow$ *true*), p2 (end), q2 (end)

   This scenario does *not* produce an incorrect result, because process p found the zero at $f(2)$, and the processes terminated at p2 and q2 because *found* was *true*. Your task is to find a scenario that *does* produce an incorrect result.

   Be alert to the fact that one of the algorithms might deadlock, and identify which one that is. A deadlock is when both processes are stuck at an `await` statement and neither process can continue.

2. Do Ben-Ari, Exercise 2.9.

3. Do Ben-Ari, Exercise 2.10.

This assignment requires you to write a C++ program and a Java program. See

http://www.cslab.pepperdine.edu/warford/cosc450/cosc-450-Setup-for-Cpp.pdf

for the C++ setup, and

http://www.cslab.pepperdine.edu/warford/cosc450/cosc-450-Setup-for-Java.pdf

for the Java setup.

1. Study Ben-Ari, Chapter 3.

2. Write a C++ program named `Count3.cpp` that adds a third process to Ben-Ari, Algorithm 2.9, page 30, that also loops `m` times and increments `n`. Note that the program will produce correct results when the final value of `n` is `3*m`.

   Experiment with the program and with `CountA.cpp` to answer the following questions.

   (1) For what value of `m` does `CountA.cpp` begin to produce incorrect results?
   (2) For what value of `m` does `Count3.cpp` begin to produce incorrect results?
   (3) Would you expect there to be a difference between these two values?
   (4) Was there a difference between these two values?

   Put the answers to these questions in the comment section at the bottom of `Count3.cpp`.

   To hand in C++ programs in this course modify the source file in the distribution, duplicate the file to be handed in, prepend the duplicated file name with your two-digit course ID, and hand it in electronically. For this problem, hand in

   `Count3.cpp`

3. Write a Java program named `Count3.java` that adds a third process to Ben-Ari, Algorithm 2.9, page 30, that also loops `m` times and increments `n`.

   Experiment with the program and with `CountA.java` to answer the following questions.

   (1) For what value of `m` does `CountA.java` begin to produce incorrect results?
   (2) For what value of `m` does `Count3.java` begin to produce incorrect results?
   (3) Was there a difference between these values and the corresponding ones for C++?

   Put the answers to these questions in the comment section at the bottom of `Count3.java`.

   All the Java programs for this course are in the following distribution

   http://www.cslab.pepperdine.edu/warford/cosc450/cosc450JavaDistr.zip

   in an IntelliJ IDE directory. To hand in Java programs in this course modify the source file in the distribution, duplicate the file to be handed in, prepend the duplicated file name with your two-digit course ID, and hand it in electronically. For this problem, hand in

   `Count3.java`

1. Practice predicting the output of Sestoft, `Example99.java`, which will be a question on the Final exam. Use the `Example99` module in the `cosc450JavaDistr` software distribution.

1. Study Ben-Ari, Sections 4.1–4.5.

2. Implement Algorithm 3.6: Second attempt in C++.
   The source file to modify is `Algorithm-3-6.cpp` in the `cosc450CppDistr` software distribution. Insert random delays in such a way that the result is usually incorrect with different results on each run. Uncomment the `cout` line in `main()`. For this problem, hand in

   `Algorithm-3-6.cpp`

3. Implement Algorithm 3.6: Second attempt in Java.
   The source file to modify is `Algorithm0306.java` in the `cosc450JavaDistr` software distribution. Insert random delays in such a way that the result is usually incorrect with different results on each run. Uncomment the `println()` line in `main()`. For this problem, hand in

   `Algorithm0306.java`

1. Study Warford, Vega, and Staley, *A Calculational Deductive System for Linear Temporal Logic*.

   https://dl.acm.org/doi/10.1145/3387109

2. Fill in the blank entries in the table below. From the table, do you believe the *until* operator $\mathcal{U}$ is associative? Explain why or why not.

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	…
p	F	F	T	T	T	T	T	F	…
q	F	T	F	T	T	F	F	F	…
r	F	F	F	T	F	F	T	F	…
$p \mathcal{U} q$	F								…
$q \mathcal{U} r$	F								…
$p \mathcal{U} (q \mathcal{U} r)$	F								…
$(p \mathcal{U} q) \mathcal{U} r$	F								…

   For the following proofs, note that there is no case analysis proof technique in temporal logic.

3. Prove (6) Distributivity of $\circ$ over $\equiv$: $\quad \circ (p \equiv q) \equiv \circ p \equiv \circ q$.

4. Prove (8) Falsehood of $\circ$: $\quad \circ \mathit{false} \equiv \mathit{false}$.

5. Prove (20) Right zero of $\mathcal{U}$: $\quad p \mathcal{U} \mathit{true} \equiv \mathit{true}$.

6. Prove (45) Expansion of $\Diamond$: $\quad \Diamond p \equiv p \vee \circ \Diamond p$.

7. Prove (72) Absorption of $\Box$: $\quad \Box \Box p \equiv \Box p$.

8. Prove (77) Strengthening of $\Box$: $\quad \Box p \Rightarrow \Diamond p$.

   You may hand in the written exercises on paper or electronically. If you hand them in electronically they must be in a PDF document named

   `a21written.pdf`

9. Implement Dekker's algorithm in C++.
   The source file to modify is `Dekker.cpp` in the `cosc450CppDistr` software distribution. Execute process p 15 times in a loop with control variable `i`. In the noncritical section, output

   `Process p, i=`*i*

   where *i* is the value of `i`. Execute process q 10 times in a loop with control variable `j`. In the noncritical section, output

   `Process q, j=`*j*

   where *j* is the value of `j`. To show that Dekker's algorithm really works, put a random 10ms delay between *every* statement in the preprotocol section and the postprotocol section as well as between the assignments to `temp` and `n` in an attempt to produce incorrect results. The final value should be 25. For this problem, hand in

`Dekker.cpp`

10. Implement Dekker's algorithm in Java.
The source file to modify is `Dekker.java` in the `cosc450JavaDistr` software distribution. Execute process p 15 times in a loop with control variable `i`. In the noncritical section, output

    `Process p, i=`*i*

    where *i* is the value of `i`. Execute process q 10 times in a loop with control variable `j`. In the noncritical section, output

    `Process q, j=`*j*

    where *j* is the value of `j`. To show that Dekker's algorithm really works, put a random 10ms delay between *every* statement in the preprotocol section and the postprotocol section as well as between the assignments to `temp` and `n` in an attempt to produce incorrect results. The final value should be 25. For this problem, hand in

    `Dekker.java`

1. Study Ben-Ari, Sections 6.1–6.10.

2. Prove (119) Monotonicity of $\Diamond$: $\quad \Box(p \Rightarrow q) \Rightarrow (\Diamond p \Rightarrow \Diamond q)$.
   Hint: Start with the entire expression and get it equivalent to *true*. Use (3.59) three times. Then, eliminate $\Box$ with (60) Dual of $\Box$. Manipulate the resulting expression containing $\Diamond$ as the only temporal operator using (3.47b) De Morgan, (52) Distributivity of $\Diamond$ over $\vee$, (3.44b) Absorption, (3.28) Excluded middle, and (3.29) Zero of $\vee$.

   You may hand in the proof on paper or electronically. If you hand it in electronically it must be a PDF document named

   `a22written.pdf`

   per the instructions for your course.

3. Implement the bounded buffer producer-consumer problem with C++ counting semaphores.
   See Ben-Ari, Problem 6.9. The source file to modify is `Algorithm-6-19.cpp` in the `cosc450CppDistr` software distribution.

   Implement the bounded buffer as a circular queue having five elements.

   The `producer` thread should:

   - Execute in a loop 15 times producing the values 10, 20, ..., 150 to store in the circular queue.
   - Print the message "Produced *x*" on a new line where *x* is 10, or 20, etc. just after leaving its critical section (after the `signal()` statement).

   The `consumer` thread should:

   - Execute in a loop 15 times consuming the values.
   - Print the message "Consumed *x*" on a new line where *x* is the value retrieved from the buffer just after leaving its critical section (after the `signal()` statement).

   To test the efficacy of your code, put random delay statements inside and outside the critical sections. To insure that there is no interleaving in your output statements, which are not atomic, incorporate a `mutex` binary semaphore to implement mutual exclusion for them.

   In addition to handing in your correct program, do an experiment by commenting out the `wait` and `signal` statements for accessing the buffer. Do *not* comment out the `mutex` operations on the output statements. Then run the program a few times and select a run that exhibits erroneous behavior. Copy the output of the run into the comment field provided at the bottom of the `.cpp` source file. Below the printout of the erroneous run, explain (1) where the first error occurred, (2) what the error was, and (3) what happened to cause the error.

   For this problem, hand in

   `Algorithm-6-19.cpp`

   CAUTION: Do not hand in your erroneous listing with the `wait` and `signal` methods commented out. Hand in your correct solution of the bounded buffer producer-consumer problem.

1. Do Ben-Ari Exercise 4.2 to prove Lemma 4.10 and use it to prove that Dekker's algorithm satisfies mutual exclusion. You may hand it in on paper or electronically. If you hand it in electronically it must be a PDF document named

   `a23written.pdf`

   per the instructions for your course.

2. Implement the dining philosophers solution of Ben-Ari, Algorithm 6.11, in Java with semaphores. The source file to modify is `Philosopher.java` in the `cosc450JavaDistr` software distribution.

   Name your class Philosopher with the following three attributes:

   - A private integer `processID` with a constructor to initialize its value
   - A private static fair semaphore `room` allocated with `new` and initialized to 4
   - A private static array of semaphores `fork` allocated with `new`

   In the main program, first allocate each fork semaphore in a loop, initializing it with the Java Semaphore constructor. Next, allocate an array of five philosophers (with `new`). Then, initialize the array of philosophers in a loop (with `new`) giving each philosopher its proper processor ID. Start each philosopher process in a loop and `join()` them in a loop.

   In your implementation of `run()`, loop through the activity of thinking and eating three times.

   - For p1, output "Philosopher *n* is thinking." where *n* is the philosopher id (0..4) on a new line. Then, sleep with a random 20 ms delay.
   - Immediately after p2, output "Philosopher *n* entered the dining room." on a new line.
   - Put a random 20 ms delay between p3 and p4 to try to force a deadlock.
   - For p5, output "Philosopher *n* is eating." on a new line. Then, sleep with a random 20 ms delay.
   - Immediately after p8, output "Philosopher *n* left the dining room." on a new line.
   - At the conclusion of the meal, output "The Philosophers are finished eating." on a new line.

   Run your program a few times to verify that several philosophers are eating concurrently. Show the following in the documentation section at the bottom of the source file.

   (1) Copy the output from one of the runs and paste it into the documentation section.
   (2) From the output in (1), what was the maximum number of philosophers in the room and who were they when the maximum occurred?
   (3) Change the initial value of the room semaphore from 4 to 5, allowing all five philosophers into the room at once. Run your program a few times to verify that it deadlocks. Copy the output from one of the deadlock runs and paste it into the documentation section.
   (4) Experiment with the length of the random delay between p3 and p4. What is the ms delay that will cause a deadlock to occur in roughly half of the runs?
   (5) Comment out the random delay between p3 and p4. Can you ever get a run to deadlock?
   (6) Explain your answer to (5).

   For this problem, hand in

`Philosopher.java`

CAUTION: Do not hand in your erroneous listing with the room semaphore initialized incorrectly. Hand in your correct solution of the dining philosophers problem with the original 20 ms delay between p3 and p4.

1. Study Ben-Ari, Sections 7.1–7.5, 7.8, 7.9, 7.11.

2. Study the paper by Buhr, Fortier, and Coffin, *Monitor Classification*, Sections 1, 2, 3.

   http://www.cslab.pepperdine.edu/warford/cosc450/Monitor-Buhr.pdf

3. Study class handout: Notes on monitors.

   http://www.cslab.pepperdine.edu/warford/cosc450/cosc-450-Notes-on-Monitors.pdf

4. Study Sestoft, Chapter 16.

5. Implement the dining philosophers solution of Ben-Ari, Algorithm 7.5, in C++ with a monitor.
   The source file to modify is `Philosopher.cpp` in the `cosc450CppDistr` software distribution.

   Even though Algorithm 7.5 is not starvation-free, starvation will never occur because our runs are finite. Any two neighboring philosophers who conspire to starve the philosopher in the middle will eventually terminate, enabling the philosopher in the middle to continue.

   Your solution should consist of a monitor named `ForkMonitor` with the following methods.

   - `takeForks(int philID)`
     - The first statement after the `unique_lock` should output "Philosopher *p* is taking forks." where *p* is the philosopher's `philID`.
     - The last statement should output "Philosopher *p* is eating."
   - `releaseForks(int philID)`
     - The first statement after the `unique_lock` should output "Philosopher *p* is releasing forks." where *p* is the philosopher's `philID`.
     - The last statement should output "Philosopher *p* is thinking."

   The main program has the following specification.

   - `philosopherRun(int philID)`. Loop three times. Inside the loop, the philosopher should
     - random delay for 60ms,
     - pick up his forks,
     - random delay for 60ms,
     - put his forks down.
   - `main()`
     - Start five philosophers.
     - Join five philosophers.
     - Output "The Philosophers are finished eating."

   Run your program several times to insure that it produces different results each time and does not deadlock. Show the following in the documentation section at the bottom of the source file.

   (1) Copy the output from one of the runs and paste it into the documentation section.
   (2) From the output in (1), what is the relationship between the statements "Philosopher *p* is releasing forks" and "Philosopher *p* is thinking"?
   (3) C++ uses Mesa semantics. Would the same relationship hold if the monitor used Hoare semantics?

(4) Explain your answer to (3).
(5) Does the same relationship in (2) hold between the statements "Philosopher *p* is taking forks" and "Philosopher *p* is eating"?
(6) Explain your answer to (5).
(7) From the output in (1), list the first eight philosophers, in chronological order, who started to eat.
(8) From the output in (1), identify the first philosopher who was blocked trying to pick up his fork.

For this problem, hand in

`Philosopher.cpp`

The final exam is cumulative, but with an emphasis on the concurrency paradigm.

1. There will be one question from the functional paradigm and one question from the declarative/logic paradigm. These questions will be taken from the exercise sets.

2. One question will give the code for Sestoft Example 90 and some three-digit input strings and will ask to predict the precise output of the computer run.

3. The following proof (among others) will be on the exam. It is part of the proof that Dekker's algorithm is starvation-free, which is in the slides. You will be given Dekker's algorithm.

   Here is Lemma 4.10
   (4.2) $turn = 1 \vee turn = 2$
   (4.3) $p3..5 \vee p8..10 \equiv wantp$
   (4.4) $q3..5 \vee q8..10 \equiv wantq$

   Here is Lemma 4.11, (L4.11): $\Box wantp \wedge \Box turn = 1 \Rightarrow \Diamond\Box\neg wantq$

   Here is Lemma A: $p2 \wedge \neg\Diamond p8 \Rightarrow p3, p4$ forever

   Assuming the above lemmas, prove Theorem 4.12 Dekker's algorithm is starvation free.

4. Know the specifications of the operations for semaphores and monitors (but not including those for Java), and the semaphore invariants (Theorem 6.1).