EECS 3401 3.0 Intro to AI and LP — Dept. of Electrical Eng. & Computer Sci.
Winter 2017 — York University

Assignment 2

Total marks: 50.

Out: February 21
Due: March 6 at 23:59

Note: Your report for this assignment should be the result of your own individual work. Take care to avoid plagiarism ("copying"). You may discuss the problems with other students, but do not take written notes during these discussions, and do not share your written solutions.

1. [20 points] This exercise involves transforming a polynomial into a list of terms in normal form. We assume a polynomial is given as a sum and/or difference of

   - terms, each of which is either
     - a coefficient or
     - a variable raised to a power or
     - a coefficient * a variable raised to a power.
   - A coefficient is a constant expression.
   - A constant expression is
     - a numerical constant or
     - a symbolic constant or
     - an algebraic expression involving no variables.
   - A variable raised to a power is
     - a variable or
     - a variable ** non-negative integer.

   Expressions are parenthesized if necessary, following the usual rules for operator precedence (* has higher precedence than +, etc.).

   The same power may occur in two different terms and there is no restriction on the order of the terms. Here are three examples of polynomials in the variable `x` which illustrate all the cases above:

```
2
-100 + x*a - 2*x**7
x**3 - (a+b) - 0.5 * x**3 + x
```

The examples above show that, in the usual mathematical notation, polynomials don't have a completely regular structure: some terms have an exponent, some include a variable and some don't, etc. But programs which do symbolic algebra typically operate on algebraic expressions expressed in a *normal form* with a uniform structure, in order to simplify the calculations. This exercise asks you to define and test Prolog predicates to transform a polynomial into a normal form in which

- each term in the polynomial sum has a *coefficient*, *variable* and a non-negative integer *exponent*, with the coefficient preceding the variable;
- the polynomial itself is presented not as a sum or difference but as a *list* of terms which are understood to be summed;
- the terms are listed in descending order by exponent;
- terms with the same exponent are regrouped into a single term in the normal form.

For this purpose, define two Prolog predicates `transformed/3` and `transformedTerms/3` as follows:

- `transformed(Term, Variable, Result)` computes or tests a `Result` term which is algebraically equal to the `Term`, but is of the form

  ```
  Coefficient*Variable**Power
  ```

  For example, `transformed(-5, z, -5*z**0)` should succeed.
- `transformedTerms(Polynomial, Variable, Result)` computes a list of the terms in `Polynomial` with each term transformed into normal form. As an example,

  ```
  transformTerms(-100 + x*a - 2*x**7, x,
                 [-2*x**7, a*x**1, -100*x**0])
  ```

  should succeed.

Most of the work will go into the `transformed` predicate. In order to get a better feel for declarative programming, avoid introducing unnecessary variables into the clauses. For example,

```
transformed(-Var, Var, (-1)*Var**1).
```

is much better than the equivalent

```
transformed(Term, Var, Result):- Term = -Var,
                                 Result = (-1)*Var**1.
```

You may wish to reduce your programming by using built-in predicates, for example, `sort/2`. To learn about built-in predicates and the SWI-Prolog library, you can download the SWI reference manual available at `http://www.swi-prolog.org`. You can also get some information from the `help/1` and `apropos/1` predicates in SWI-Prolog. For example,

```
?- apropos(sort).
sort/2                  Sort elements in a list
msort/2                 Sort, do not remove duplicates
keysort/2               Sort, using a key
predsort/3              Sort, using a predicate to determine the order
merge/3                 Merge two sorted lists
merge_set/3             Merge two sorted sets

Yes

?- help(sort/2).
sort(+List, -Sorted)
  Succeeds if Sorted can be unified with a list holding the elements
  of List, sorted to the standard order of terms (see section 4.6).
  Duplicates are removed.  The implementation is in C, using natural
  merge sort.
```

Submit both your Prolog code in file `q1.pl` and your test results in the file `q1tests.txt`. Provide enough tests to convince yourself and the reader that your implementation is correct. Document your code appropriately.

2. [30 points] In this exercise, we use Prolog to implement a subset of an abstract process algebra which can be used to analyze concurrent processes. Expressions in the algebra describe the structure of a process constructed from primitive actions that can be carried out in a particular system. An expression in a process algebra can be tested to see if the process described by the expression has a particular property, for example, whether the process can be proved to eventually terminate. Each primitive action `A` in the process/system must be declared by asserting `primAction(A).`

   A process is then defined as one of the following:

   - `0` (the empty process – nothing left to do), a primitive action,
   - `A > P`: a sequence of a primitive action `A` followed by a process `P`,
   - `P1 ? P2`: a non-deterministic branching that either does process `P1` or process `P2`,

- `P1 | P2`: interleaved concurrent execution of process `P1` and `P2`.
- `P1 $ P2`: synchronized concurrent execution of processeses `P1` and `P2`.
- `ProcName`: a call to the procedure named `ProcName`.

Procedures are defined by asserting `defproc(ProcName,Body)` where `ProcName` is a symbol that is the procedure's name and `Body` is a process expression that is the procedure's body. When the procedure's name occurs in a process expression, it can be replaced by procedure's body. Procedures can be recursive, for example:

```
defproc(iterDoSomething, doSomething > iterDoSomething ?  0).
```

which performs the primitive action `doSomething` 0 or more times.

We impose the following restrictions on recursive procedure definitions: their body cannot contain the concurrent execution constructs; and they must always perform at least one primitive action before making a recursive call.

Among the process composition operators we assume that sequence `>` has highest precedence, followed by nondeterministic branch `?`, then interleaved concurrency `|`, and finally synchronous concurrency `$`. Parentheses can be used to override this. You can obtain the right precendence in Prolog by using the following declarations:

```
:- op(700,xfy,>).
:- op(800,xfy,?).
% | is predefined as xfy with precedence 1100
:- op(1120,xfy,$).
```

The execution of processes can be defined in terms of transitions. Let `P1-A-P2` mean that process `P1` can do a single step by performing action `A` leaving process `P2` remaining to be executed. We can define this relation as follows:

- `0 - A - P` is always false.
- `A - A - 0` holds (where `A` is a primitive action), i. e., an action that has completed leaves nothing more to be done.
- `(A > P) - A - P` (where `A` is a primitive action), i.e., doing a step of a sequence `(A > P)` involves doing the initial action `A` leaving `P` to be done afterwards.
- `(P1 ? P2) - A - P` holds if either `P1 - A - P` holds or `P2 - A - P` holds.
- `(P1 | P2) - A - P` holds if either `P1 - A - P11` holds and `P = (P11 | P2)`, or `P2 - A - P21` holds and `P = (P1 | P21)`

- `(P1 $ P2) - A - P` holds if both `P1 - A - P11` holds and `P2 - A - P21` holds and `P = (P11 | P21)`
- `ProcName - A - P` holds if `ProcName` is the name of a procedure that has body `B` and `B - A - P` holds.

We can define this in Prolog as follows:

```
A-A-0 :- primAct(A).
(A > P)-A-P :- primAct(A).
(P1 ? P2)-A-PR :- P1-A-PR ; P2-A-PR.
(P1 | P2)-A-(P1R | P2) :- P1-A-P1R.
(P1 | P2)-A-(P1 | P2R) :- P2-A-P2R.
(P1 $ P2)-A-(P1R $ P2R) :- P1-A-P1R, P2-A-P2R.
PN-A-PR :- defproc(PN,PB), PB-A-PR.
```

We can also define a predicate `final(P)` that holds when process `P` may legally terminate. The definition in Prolog is as follows:

```
final(0).
final(P1 ? P2):- final(P1); final(P2).
final(P1 | P2):- final(P1), final(P2).
final(P1 $ P2):- final(P1), final(P2).
final(P):- defproc(P,B), final(B).
```

An execution of a process is a sequence of transitions, which we will represent by a list `[P1, A1, P2, A2, ...]`, such that for all `i > 0`, `Pi - Ai - Pi+1`. A complete execution is an execution where the last process is `final` or cannot make any further transitions.

Let's look at a few simple examples:

- `(a1 > a2 > a3)` has only one complete execution: `[(a1 > a2 > a3), a1, (a2 > a3), a2, a3, a3, 0]`
- `((a1 > a2) | a3)` has 3 complete executions:
  `[((a1 > a2) | a3), a1, (a2 | a3), a2, (0 | a3), a3, (0 | 0)]`,
  `[((a1 > a2) | a3), a1, (a2 | a3), a3, (a2 | 0), a2, (0 | 0)]`, and
  `[((a1 > a2) | a3), a3, ((a1 > a2) | 0), a1, (a2 | 0), a2, (0 | 0)]`;
  interleaved concurrency interleaves the actions of the component processes.

- `(a1 $ a1)` has one complete execution: `[(a1 $ a1), a1, (0 $ 0)]`; when we use synchronous concurrency, both component processes advance.
- `(a1 $ a2)` has no executions; synchronous concurrent processes can only advance if they perform the same action.
- `p1` where `defproc(p1, a1 > p1)` has the infinite execution `[p1, a1, p1, a1, ...]`.

Let's now look at some more interesting examples.

Example 1:

This is a simple example of processes that can deadlock; the processes try to acquire two locks in different orders.

Actions: `acquireLock1, acquireLock2, releaseLock1, releaseLock2, doSomething`

Process definitions:

```
defproc(deadlockingSystem, user1 | user2 $
                           lock1s0 | lock2s0 | iterDoSomething).
defproc(user1, acquireLock1 > acquireLock2 > doSomething >
               releaseLock2 > releaseLock1).
defproc(user2, acquireLock2 > acquireLock1 > doSomething >
               releaseLock1 > releaseLock2).
defproc(lock1s0, acquireLock1 > lock1s1 ? 0).
defproc(lock1s1, releaseLock1 > lock1s0).
defproc(lock2s0, acquireLock2 > lock2s1 ? 0).
defproc(lock2s1,releaseLock2 > lock2s0).
defproc(iterDoSomething, doSomething > iterDoSomething ? 0).
defproc(oneUserSystem, user1 $ lock1s0 | lock2s0 | iterDoSomething).
```

The process `deadlockingSystem` may deadlock. The single user version `oneUserSystem` cannot deadlock.

Example 2:

In this example, there is producer process that generates data and a consumer process that consumes it. The data is stored in a buffer can handle up to 3 items. The buffer can overflow and underflow. One can use synchronization actions to avoid this.

Actions: `produce, consume, underflow, overflow, notFull, notEmpty`

Process definitions:

```
defproc(producerConsumerSyst,
           producer | consumer | faults $ bufferS0).
defproc(producer, notFull > produce > producer).
defproc(consumer, notEmpty > consume > consumer).
defproc(faults, underflow ? overflow).
defproc(bufferUF, notFull > produce > bufferUF ?
                  produce > bufferUF ?
                  consume > bufferUF).
defproc(bufferS0, notFull > produce > bufferS1 ?
                  produce > bufferS1 ?
                  consume > underflow > bufferUF).
defproc(bufferS1, notFull > produce > bufferS2 ?
                  produce > bufferS2 ?
                  consume > bufferS0 ?
                  notEmpty > consume > bufferS0).
defproc(bufferS2, notFull > produce > bufferS3 ?
                  produce > bufferS3 ?
                  consume > bufferS1 ?
                  notEmpty > consume > bufferS1).
defproc(bufferS3, produce > overflow > bufferOF ?
                  consume > bufferS2 ?
                  notEmpty > consume > bufferS2).
defproc(bufferOF, produce > bufferOF ?
                  consume > bufferOF ?
                  notEmpty > consume > bufferOF).
defproc(producerConsumerSystBuggy,
           producerB | consumerB | faults $ bufferS0).
defproc(producerB, produce > producerB).
defproc(consumerB, consume > consumerB).
```

a) Define a Prolog predicate `run(P,R)` that holds iff `R` is a complete execution of process `P`. Also define a `print_run(R)` predicate that prints executions in a readable way. Test this (at least) on the `oneUserSystem` and `deadlockingSystem` examples.

b) Define a Prolog predicate `has_infinite_run(P)` that holds iff process `P` has an infinite run (this happens only if there is a there is a cycle in the configuration graph). Test this (at least) on the examples above.

c) Define a Prolog predicate `deadlock_free(P)` that holds iff process `P` cannot reach a deadlocked configuration, i.e., a configuration where the process is not `final` but cannot make any further transition. Test this (at least) on all the examples above.

d) Define a Prolog predicate `cannot_occur(S,A)` that holds iff there is no execution of process `P` where action `A` occurs (an instance of checking a safety property). Test (at least) `cannot_occur(P,overflow)` on the two versions of the producer-consumer example.

e) Define a Prolog predicate `whenever_eventually(S,A1,A2)` that holds iff in all executions of process `P`, whenever action `A1` occurs, action `A2` occurs afterwards (a instance of checking a liveness property). Test (at least) `whenever_eventually(P,produce,consume)` on the two versions of the producer-consumer example.

For all the parts of the question, provide enough tests to convince yourself and the reader that your implementation is correct. The tests can involve very simple processes where it is easy to see what should happen. Submit both your Prolog code in file `q2.pl` and your test results in the file `q2tests.txt`. Document your code appropriately.

To hand in your report for this assignment, put all the required files in a directory `a2answers` *and submit it electronically by the deadline. To submit electronically, use the following Prism lab command:*

```
submit 3401 a2 a2answers
```

Your Prolog code should work correctly on Prism.