Programming Problem for the 2002 Comprehensive Exam

Out: Monday 1/14/02 at 9:00 am
Due: Friday 1/18/02 at 5:00 pm

Department of Computer Science
Brown University
Providence, RI 02912

1.0 The Task

In my many travels through the Computer Science literature I have found a wide variety of mysterious things. Recently, while looking through a stack of old papers I discovered a proposal for a visual programming language that has some interesting features but that has never been implemented. My guess is that the visual aspects of the language are fairly simple and not programmatically interesting. However, the interpreter or run time system that would be needed for the language seemed quite interesting and somewhat nonintuitive. In order to find out whether this language is at all practical and reasonable, I'm going to have you code up an interpreter for it as this year's comps assignment.

2.0 Language Basics

The language consists of two types of components, connections between these components, and a set of rules for execution. The components are stores and processors. The components operate over data that consists of integers organized into tuples.

2.1 Constants

There are two primitive data types, integer and integer tuple. An integer constant is just that, a 32 bit value with the customary decimal representation. (If it is more convenient, you can change this as long as integers have at least 30 bits.) A constant integer tuple is also what you would expect, a nonempty ordered list of integers enclosed in brackets and separated by commas. Thus the following are legal constant data values:

```
3
17
[4]
[3,7,4,-324]
```

2.2 Variables

The language includes a set of twenty-six simple variables corresponding to the names `a` through `z`. All variables are restricted to taking on integer values.

2.3 Arithmetic Expressions

Arithmetic expressions are composed of integer constants and variables possibly combined in the standard way with the arithmetic operators `+` (add), `-` (subtract or unary minus), `*` (multiply), `/` (quotient), `%` (modulus), and `^` (exponentiation). The operators follow the normal precedence and associativity rules and parenthesis may be used as necessary.

To simplify things we will assume that modulus is not defined when either of the operands is negative, that quotient and modulus are not defined when the second operand is zero. We will also assume that the result of an operation that overflows is undefined (but that the operation itself is defined — that is, you don't have to worry about overflow in your code).

2.4 Conditional Expressions

A conditional expression is a Boolean formula consisting of two arithmetic expressions and a relational operator. The valid relational operators are `==`, `!=`, `<=`, `>=`, `<` and `>`.

2.5 Tuple Expressions

Tuple expressions are similar to the tuple data described above except that they contain arithmetic expressions that have integer values as their components. Thus the following are valid tuple expressions:

```
[x]
[34]
[x,y,x+3]
[x^2,x%3,y,5]
```

2.6 Stores

A *store* in the language is simply a set of data tuples. Here we mean set in the proper sense, i.e., a collection consisting of zero or more tuples, without duplication, with no implicit or explicit ordering on the elements. Each store in a program is given an initial value, possibly empty. Each store is also assigned a unique label, used primarily for debugging and making statements about the program.

The data in a store must be homogeneous in that all tuples in a store must have the same number of elements. Note that this number should be determinable from the program and may be considered part of the store specification.

2.7 Processors

A *processor* is a component that has its own set of variables (again restricted to `a` through `z`) and a set of conditional equations. The conditional equations must consist only of arithmetic equations over integer constants and elements of the processor's set of variables and a single relational operator. Note that the set of conditional equations might be empty. Each processor is assigned a label, again used primarily for debugging and referring to parts of the program.

2.8 Connectors

Processors and stores are linked by labeled *connectors*. A connector is a directed arc either from a store to a processor (an *input* connector) or from a processor to a store (an *output* connector). Each connectors is labeled with a single tuple expression. Note that the arity of the tuple expression can be used to determine the arity of the store to which it is connected. All connectors to a single store must have tuple expressions with the same arity.

The label on an input connector must be a tuple expression containing only variables. That is, the input label should be a tuple whose elements are all variables (no constants, no expressions). An instance of a variable in the label on an input connector is called a *defining instance* of that variable for the processor. Each variable must have a unique defining instance.

The label on an output connector is an arbitrary tuple expression over the set of variables of the processor. That is, an output label can contain arithmetic expressions over these variables.

2.9 Programs

A *program* consists of a set of stores, processors, and connectors subject to the following constraints:

- Every processor has at least one input connector.
- Every store has at least one connector (either input or output).
- For each variable of each processor there is exactly one defining instance.
- Every store has an associated initial set of data.

3.0 Language Execution

The above defines the notion of a program in the language. Executing this program consists of executing a series of steps in a fair way. (Fair is used in the concurrent programming sense and is defined below.)

3.1 Processor Execution

We first define the notion of a processor executing. Conceptually a processor can execute if it can find tuples for each of its input connectors such that the conditional equations specified by the processor are satisfied and all output connections would be valid. Another way of looking at this is that there is a way of setting each of the variables of the processor so that there are distinct matching input tuples on each of the input connectors, so that the conditional equations associated with the processor are satisfied, and so that all output tuples that would result can be added to the corresponding stores without duplication.

More formally, a processor P is *enabled* if and only if the following conditions are met:

- There is a mapping S of the variables of the processor to constant data values.
- For each input connector I, let $\underline{S}(I)$ denote the constant data that results from the transformation replacing each occurrence of a variable in the label of I with the corresponding data value. Similarly, for each output connector O, let $\bar{S}(O)$ be the result of replacing all variables in O with the corresponding values in S and then performing all arithmetic operations.
- For each input connector I, there must be a tuple in the store of I that equals $\underline{S}(I)$. Moreover, if there are multiple input connectors to the processor from the same store, the corresponding $\underline{S}(I)$'s must be different (not equal). Intuitively, if the processor executes, it will remove the tuples $\underline{S}(I)$ from the corresponding stores. This constraint ensures that such tuples exist for each input connector.
- For each conditional equation C of P let $\tilde{S}(C)$ denote the result of replacing each variable in C with the corresponding data value. Then $\tilde{S}(C)$ must be well defined and must evaluate to true. $\tilde{S}(C)$ is well defined if and only if all operations apply to integers and if none of the arithmetic operations are undefined.
- $\bar{S}(O)$ is well defined, in the sense that arithmetic operations are only performed on integers and none of the arithmetic operations are undefined.
- For each output connector O, the result $\bar{S}(O)$ must not be in the corresponding store or must be equal to $\underline{S}(I)$ for some input connector I from the same store. Moreover, if there are two or more output connectors that are linked to the same store, they must each have different values. Intuitively, if the processor executes, it will store the tuples $\bar{S}(O)$ into the corresponding stores. In order for the execution to be valid, the corresponding tuples must not already be in the stores. This means that either they were not there initially or they were removed through one of the input connectors of the processor.

A processor executes by choosing a mapping S that satisfies the above and then doing the following as an atomic action:

- First, for each input connector I, the tuple $\underline{S}(I)$ is removed from the corresponding store.
- Then, for each output connector O, the tuple $\bar{S}(O)$ is added to the corresponding store.

3.2 Program Execution

A program executes by choosing a processor that can execute, executing that processor, and then repeating until no processor can execute.

The only constraint on program execution is one of fairness. While this is a bit complex to describe formally, it essentially means that if a processor *can* execute it eventually *will*. In particular, if at a point t in a program, processor P can execute, and at an infinite number of subsequent times $t' > t$, P can also execute, then fairness says that P must eventually be executed. (Note that with a deterministic choice of which processor to execute, you can construct programs where this may not hold true. For example, there could be another processor Q that is also executable and the system could always choose to execute Q rather than P.) Fairness can be achieved in various ways -- round robin scheduling, randomly choosing the next processor to execute, etc.

Fairness also extends to choosing the variable assignment when executing a processor. If there are multiple assignments S that can be used for executing P, then program execution must ensure that any assignment that is viable an infinite number of times will eventually be used. Again it is possible to construct programs where there are always at least two potential assignments that can be chosen so that this property is not guaranteed a priori. Note that assignments here are considered in the whole, not variable-by-variable. Thus if x can always be assigned A or B and y can always be assigned C or D, then fairness requires that each of the pairs (A, C), (A, D), (B, C), and (B, D) will occur as assignments for x and y.

4.0 Syntax

This framework (which we have simplified somewhat) was designed to be used as the basis for a concurrent visual programming language. We'll now describe a graphical representation of programs in order to show a few simple example programs in a way that might be understandable. *This graphical representation is just to make it easy to show you some examples; you do not need to implement anything graphical.* The graphical syntax introduces a number of complexities that are not dealt with directly in the above framework, but which make the visual descriptions of programs more compact.

After introducing the visual syntax and a number of programs, we describe what will actually be input to your system in terms of a textual syntax.

4.1 Visual Representation

A visual representation of a program in this language is relatively simple. Each store is represented as a circle, each processor as a box, and each connection as a labeled directed arc. Thus the overall program is represented as a graph.

Stores may be tagged with their corresponding label. If a store has a non-empty initial set of constant data, the set is listed on the graph near (or in) the store (or somewhere else,

tagged by the label of the store), either as an explicit list or as a set equation yielding the data. For example, $\{[i, i] | (0 < i < 5)\}$. represents the constant tuple `[1,1]`, `[2,2]`, `[3,3]` and `[4,4]`. This notation represents a visual extension; the textual syntax that you will deal with will use only explicit tuple sets as values.

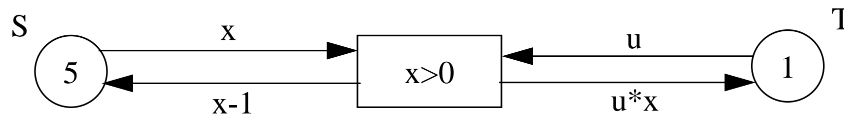
Processors may also be tagged with their corresponding label. The set of conditional equations associated with the processor is specified as a list of equations inside the processor box. The processor's set of variables is declared implicitly as the collection of all variables appearing on input connectors to the processor.

Connectors are directed arcs (i.e., they have an arrow head showing which way they go) that are labeled with their corresponding tuple expression. Again, we will use the shorthand of a single integer expressions to represent the tuple expression containing the single element with the corresponding value.

A number of visual abbreviations are allowed. First, a connector can have multiple tuple expressions. This implies that it actually represents multiple connectors, each of which is labeled with one of those tuple expressions. Second, a connector may be bidirected (i.e., have arrow heads at both ends). This implies that there is both an input and output connector connecting the store and processor and the two have the same tuple expression as their label. Third, the same variable can appear in several input connectors for a single processor. This implies that the variables are actually distinct and that there are additional conditions in the processor that require these distinct variables to be equal. Fourth, we will allow labels containing arithmetic expressions to be used on input connectors as long as all the variables in the expression are defined on some other input connector. This is equivalent to replacing the expression with a single new variable and then adding a conditional expression in the processor that requires this variable to be equal to the given expression. Finally, we will let simple integer expressions be used in stores and as connector labels to represent singleton tuples. Thus the value or label `5` would actually represent the tuple `[5]` and the label `u*x` would actually represent the tuple expression `[u*x]`.

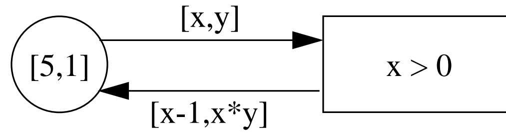
4.2 Sample Programs

The following simple sample program shows how this language can be used to implement a simple sequential factorial function:

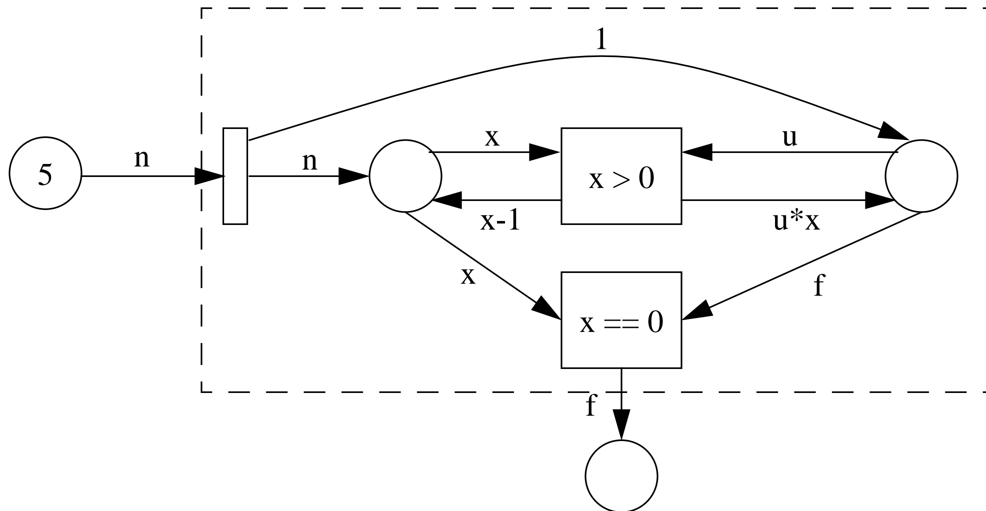


Note that the all the values in Program 1 are actually singleton tuples, not integers or integer expressions. Execution of this program is rather simple since at any one time there is at most one tuple in each of S and T and that tuple determines x and u.

The next program shows the same thing done using tuples and a single store:

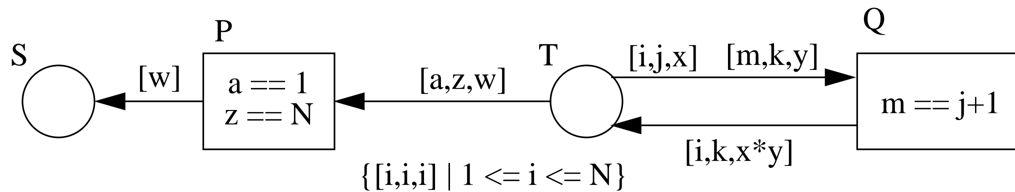


A slightly more complex variant of this takes a single input an generates a single output:



Note that this example shows how subroutines (indicated by the dotted square) could be specified.

A more complex version of factorial that works concurrently can be written as:



Note that this example illustrates the use of two labels on one connector and the use of a set expression to initialize a store. Here the doubly-labeled connector is equivalent to two separate connectors, one with the label `[i,j,x]` and the other with the label `[m,k,y]`. The store in the middle, for `N=5` would be initialized with the tuples `[1,1,1]`, `[2,2,2]`, `[3,3,3]`, `[4,4,4]`, and `[5,5,5]`.

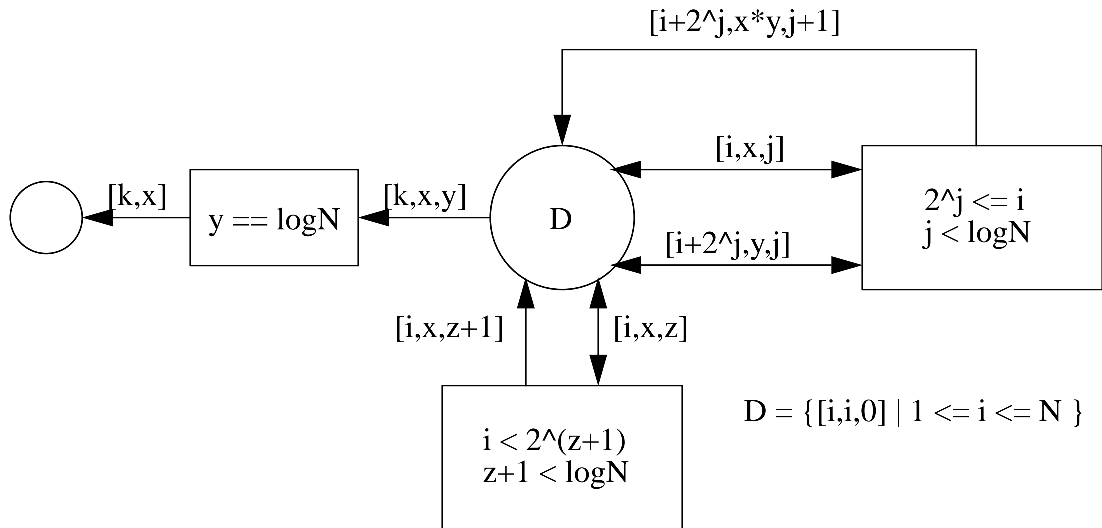
Execution of this example is a bit more complex (and nondeterministic). At first, only processor Q can execute, but it can execute with any of the combinations:

```
[1,1,1] and [2,2,2]
[2,2,2] and [3,3,3]
[3,3,3] and [4,4,4]
[4,4,4] and [5,5,5]
```

One possible sequence of processor executions would be:

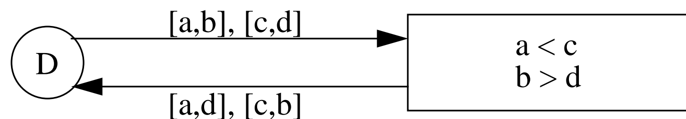
i	**j**	**x**	**m**	**k**	**y**	**a**	**z**	**w**	**T**	**S**
									[1,1,1] [2,2,2] [3,3,3] [4,4,4] [5,5,5]	
3	3	3	4	4	4				[1,1,1] [2,2,2] [5,5,5] [3,4,12]	
1	1	1	2	2	2				[5,5,5][3,4,12][1,2,2]	
3	4	12	5	5	5				[1,2,2][3,5,60]	
1	2	2	3	5	60				[1,5,120]	
						1	5	120		[120]

A more complex (and possibly faster) concurrent factorial would be:



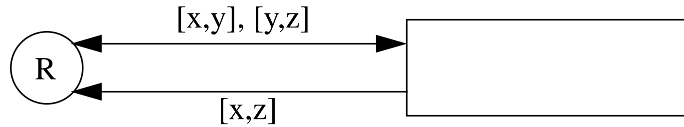
This program again shows a number of abbreviations. First, it uses bidirectional connectors to represent both an input and an output connector with identical labels. (These have the effect of using a tuple from the store for the computation without actually removing it.) Second, it shows the use of expressions on an input connector (actually on a input-output connector, but used for both input and output). Note that the variables used in the expression, `i` and `j`, are both defined by another input connector. Finally, it shows the use of a store label (`D`) and to allow the placement of the initial value outside of the store. Note that `N` here is a constant (it would be set to some value when actually inputting or running the program, and hence `log N` is constant as well.

Programs to do things other than computing factorial can also be written. For example, the following sorts the array A_i, `i = 1... n`:



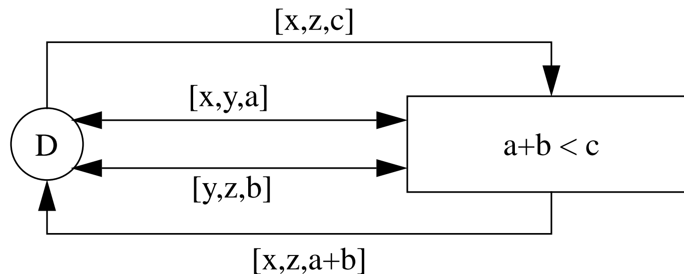
$D = \{[i,A_i] \mid 1 <= i <= n\}$

Next, the following computes the transitive closure of a binary relation `R`:



Note that the processor in this program can always find inputs, but when the transitive closure has been completely computed, there will be no place to put the output, so that the processor will no longer be enabled.

Finally, given a set `D` of arc costs `[x,y,c]` where `x` and `y` are the end points and `c` is the cost, the shortest path problem can be computed using:



Note that D here must contain values for all possible pairs of `x` and `y`, using some large value `M` for those cases where `x` and `y` are not directly connected.

4.3 Textual Syntax

While the graphical syntax is designed to be easy to read by humans, the textual syntax, the one that you will be using, is designed to be easy to read by machine. In this section we introduce a sample XML-based syntax as an illustration of what input might be like. You may use this syntax or design and use your own (which you should document).

The textual syntax needs to describe the stores, processors, and connectors. It does each of these in a separate section of the XML:

```
<PROGRAM>
   <STORES> ... </STORES>
   <PROCESSORS> ... </PROCESSORS>
   <CONNECTIONS> ... </CONNECTIONS>
</PROGRAM>
```

4.3.1 Stores

Each store is defined using a STORE element within the second-level STORES element. The STORE element looks like:

```
<STORE ID="store_label" TUPLESIZE="size">
   <TUPLES> ... </TUPLES>
</STORE>
```

where the TUPLESIZE attribute gives the arity of tuples that can be placed in the store. The tuples comprising the data of the store are defined using corresponding XML expressions defined using a LIST element containing INT elements. For example, the values `[1,3]`, `[2,3]` and `[3,5]` would be represented as:

```
<TUPLES>
   <LIST><INT VALUE="3"/><INT VALUE="5"/></LIST>
   <LIST><INT VALUE="2"/><INT VALUE="3"/></LIST>
   <LIST><INT VALUE="3"/><INT VALUE="5"/></LIST>
</TUPLES>
```

4.3.2 Processors

Each processor is defined using a PROCESSOR element within the second-level PROCESSORS element. The PROCESSOR element looks like:

```
<PROCESSOR ID="proc_label">
   <VARIABLES> ... </VARIABLES>
   <CONDITIONS> ... </CONDITIONS>
</PROCESSOR>
```

Here the elements that are defined within the CONDITIONS element are general expressions (defined below). The variables associated with the processor are defined as VARIABLE elements within the VARIABLES element. These look like:

```
<VARIABLE NAME="a" />
```

4.3.3 Connections

Each connection is defined using a CONNECTION element within the second-level CONNECTIONS element. The CONNECTION element for an input connection looks like:

```
<CONNECTION TYPE="IN" SOURCE="store_label" TARGET="proc_label">
   expression
</CONNECTION>
```

while the similar output CONNECTION element looks like:

```
<CONNECTION TYPE="OUT" SOURCE="proc_label" TARGET="store_label">
   expression
</CONNECTION>
```

In both cases, the element within the CONNECTION element is a general expression element that describes the label associated with the connection. Note that our prior restrictions mean that an input connection will only contain constants and variable names and no operators.

4.3.4 Expressions

Expressions are defined in our XML notation in prefix form using EXPR, VAR, INT, and LIST elements. INT and LIST elements are as defined for specifying the tuples of a store. VAR elements are used to reference a variable and look like:

```
<VAR NAME="a"/>
```

EXPR elements are used for all of the expression operators. They take the form:

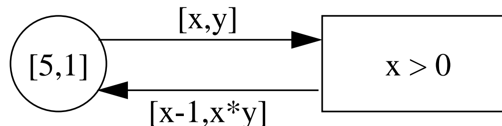
```
<EXPR OP="operator"> operands... </EXPR>
```

For example, the expression `[i+2^j,y,j]` would be represented as:

```
<LIST>
   <EXPR OP="+">
      <VAR NAME="i"/>
      <EXPR OP="^">
         <INT VALUE="2"/>
         <VAR NAME="j"/>
      </EXPR>
   </EXPR>
   <VAR NAME="y"/>
   <VAR NAME="j"/>
</LIST>
```

4.3.5 A Complete Example

Consider the previously shown factorial program:



It would be represented in our notation as:

```
<PROGRAM>
   <STORES>
      <STORE ID="A" TUPLESIZE="2">
         <TUPLES>
            <LIST><INT VALUE="5"/><INT VALUE="1"/></LIST>
         </TUPLES>
      </STORE>
   </STORES>
   <PROCESSORS>
      <PROCESSOR ID="B">
         <VARIABLES>
            <VARIABLE NAME="x"/>
            <VARIABLE NAME="y"/>
         </VARIABLES>
         <CONDITIONS>
            <EXPR OP="&gt;"><VAR NAME="x"/><INT VALUE="0"/></EXPR>
         </CONDITIONS>
      </PROCESSOR>
   </PROCESSORS>
   <CONNECTIONS>
      <CONNECTION TYPE="IN" SOURCE="A" TARGET="B">
         <LIST><VAR NAME="x"/><VAR NAME="y"/></LIST>
      </CONNECTION>
      <CONNECTION TYPE="OUT" SOURCE="B" TARGET="A">
         <LIST>
            <EXPR OP="-"><VAR NAME="x"/><INT VALUE="1"/></EXPR>
            <EXPR OP="*"><VAR NAME="x"/><VAR NAME="y"/></EXPR>
```

```
            </LIST>
         </CONNECTION>
      </CONNECTIONS>
   </PROGRAM>
```

5.0 Your Task

Your task is to create a program that provides the run time engine for this concurrent framework. There are several steps that you must complete.

First you should devise an input format for your system based on the above textual graphical representation. We described an XML-based syntax above, but you are free to modify this or to choose another format that might be easier to parse or read based on your implementation language. You should then write code that reads the program in, checks that it is a legal program (according to the previous criteria), and prints an error and terminates if it is not.

Second, you should provide a means for executing the program according to the rules cited above. Your system should run the program until no more processors can be executed. Your execution must be fair.

Third, you should provide output that will include dumping the state of all stores at the end of execution.

Fourth, you should (for your sanity and mine) provide a means of tracing program execution. Here you should indicate each step of the program, showing which processor fired, which tuples were used, and which tuples were generated.

Fifth, you should run your system on the examples provided above. XML for these examples will be available in the directory `/u/spr/comps/2002/data`.

Finally, you should create your own program for this framework that is independent of the test examples. You should run your system on your new program. (Actually, you might want to create this program first since doing so will help you understand more fully how the language works and thus how you might design and implement your implementation.)

Algorithmic efficiency is not a primary consideration and brute force solutions will be considered acceptable. However, extra credit will be given for innovative solutions that cut down on execution (or search) time.

6.0 Mechanics

You may write this program in any programming language available on any of the Department of Computer Science's systems. (For grading purposes, your program will have to be able to be run on one of our systems.) You can use any libraries that are considered part of that language. For example, the STL is considered part of C++; all classes in java.* or javax.* are considered part of Java. Standard libraries that are not directly related to the

assignment (such as an XML parser) can also be used. You may use any machines that you normally have access to at Brown or you may use your own personal machine. If you have any questions as to what is or isn't valid, you should ask Dr. Reiss before proceeding.

You should hand in your program (source and executable), sample runs, and whatever output you have to demonstrate your program works. If possible, you should encode your own input program in the above XML format. You must also include a brief write-up that describes the algorithms used, the overall program structure, and how to run your program. Handins should be done electronically if at all possible into the directory `/map/aux1fred/comps/handins/###` where `###` is the random number you assign yourself. You should have permissions to create this directory and should protect it accordingly. All directories will be locked at the moment the exam is due and you will be graded on what is there at the time. Written handins, while strongly discouraged, will be accepted provided they are physically given to Dr. Reiss before the deadline.

Grading will be based on your choice of data structures and algorithms as well as on the correctness, performance, style, and readability of your program and documentation.

All work that you hand in must be your own. You should not discuss your work or ideas with anyone else. You should not share code or ask others for advice on your code. You should not read other's code (either those here or code obtained elsewhere) that duplicates in any way what you are to write. You may ask Dr. Reiss questions either in person or via email. Any answers or information that should be known to all will be posted in the comps news group which is your responsibility to read.

Good luck and have fun.