Mathematical Expression Evaluator (Assessed Individual Coursework, Foundations 2 (F29FB), Spring 2018)

2018-01-25 (Thursday)

1 Individual work and attribution

- You must write the source code, report, and test data that you submit wholly by yourself, individually. English text must be in your own words. Code and test data must be in your own tokens and logic.
- Every bit of what you submit that you did not originate you must explicitly mark the boundaries of, attribute, and properly reference in writing, to the extent this is reasonably possible.
- Your submission may include source code provided by the instructor for this purpose. (Of course you must properly attribute this.)
- You must not share (whether receiving or giving, voluntarily or involuntarily) chunks of code or text you have written for this assignment with fellow students, regardless of how: paper, email, computer file, computer screen, etc. You must refuse and not request such sharing. You must keep your work on this assignment secure.
- Failing to abide by these instructions violates the university's rules against academic misconduct which forbid *failure to reference*, *collusion*, and *plagiarism*. (You are already responsible for knowing these rules and this is just a reminder.) Suspected violations will be reported to the MACS Discipline Committee. Students found to have violated these rules will be penalized. For third-year students, the penalties for a first offence can be severe, e.g., voiding this course and maybe also voiding another course.

2 Overall idea

You must write a program that reads expressions denoting mathematical entities, calculates which entity each expression denotes, and then writes the results. Your program must remember the results for later reuse.

A major goal is to ensure that you properly understand the mathematics used in the course. If you complete this coursework, you will have a good understanding of what a function is as well as how Cantor's diagonalization method works. You will practice designing and implementing data structures to model mathematical entities.

3 Due dates, mark weighting, and report length limitations

Part 1 This part will count for 10% of the mark. This part is due at 2018-01-24 T 15:15 (Wednesday, week 3).

You must not submit a report for part 1.

Assessment will be by a personal interview after the work is submitted. The interview is to be scheduled by 2018-02-01 (Thursday, week 4). In the interview I will inspect and test the work that was submitted and ask you questions. Feedback for this part will only be given during this interview.

Part 2 This part will count for 35% of the mark.

Your preliminary part 2 test data is due at 2018-02-07 T 15:15 (Wednesday, week 5).

Your complete part 2 submission (with your test data revised to provide better coverage of your program) is due at 2018-02-16 T 15:15:00 (Friday, week 6).

Your part 2 report must not exceed 2 pages.

Part 3 This part will count for 55% of the mark.

Your preliminary part 3 test data is due at 2018-03-16 T 15:15:00 (Friday, week 10).

Your complete part 3 submission (with your test data revised to provide better coverage of your program) is due at 2018-03-23 T 15:15:00 (Friday, week 11).

Your part 3 report must not exceed 4 pages.

The preliminary test data for parts 2 and 3 counts for 10% of the mark for these parts.

WARNING: Late submission of preliminary test data is not accepted.

The weighting of the marking criteria for the 3 parts is as follows:

	part 1	part 2	part 3
functionality	60%	40%	40%
explanation	10%	20%	20%
design	20%	20%	20%
presentation	10%	20%	20%

4 Differences between parts

- For part 1:
  - Expressions can only be: integers, ordered pairs of integers, sets of integers.
  - It therefore makes no difference whether you use the simplified input.
- Part 2 adds:
  - Sets and ordered pairs can contain the results of any expression, including sets and ordered pairs.
  - Expressions can also be variables and tests of equality or membership.
  - Sets must be printed with each member listed at most once in the printout.
  - The simplified input is actually simpler.
- Part 3 adds:
  - Expressions can be formed using additional operators.
  - The report must contain specified reasoning about the diagonalize operator.

5 Specification

5.1 Requirement to use trees/graphs and prohibition on string operations

Not only must your program implement the input/output behavior specified below, but your program must also internally implement this behavior as follows.

- You must represent each ordered pair as a tree/graph node which has 2 children that represent the left and right components of the pair. Similarly, you must represent each set as a tree/graph node such that for each member of the set there is a child of the node that represents it.

  NOTE: There is no requirement that all nodes belong to the same tree/graph, nor is there any requirement that they belong to different trees/graphs.

  WARNING: The concept of "tree" taught in previous courses has a lot of stuff in it that you will not need.

- At any point in the execution of your program after the input is parsed by the JSON parser, you must **NOT** represent any ordered pair or set as a string or as an array or list whose elements are all characters/symbols/tokens.

- You must implement a subroutine (possibly as a bunch of methods spread across a number of classes) that traverses the representation of a mathematical entity and **during the traversal outputs** a printed representation of that entity using standard mathematical notation.

To help ensure that you genuinely use tree/graph nodes, you are absolutely forbidden from building or altering character arrays/lists or strings at run-time, except as specified here:

- You may use strings created from string literals in your program.
- You may use your programming language's function(s) for converting integers to strings.
- You may use strings created by the JSON parser.

Note that outputting and comparing strings are both allowed.

5.2 Overall program behavior

Your program must read the entire contents of a single input file and produce a single output file with contents that are correct for that input.

The input file will be readable in the current directory and unchanging while your program runs.

Your program file will be run directly with no arguments.

Your program must exit with the output file existing in the current directory with the correct contents.

The input file will contain a sequence of declarations, in the format defined below. Each declaration defines a variable VAR to be some expression EXP. For each such declaration, your program must:

1. Calculate the value VAL or the absence of a value that results from evaluating the expression EXP using the rules given below.
2. Remember for use in later expressions that the variable VAR now has the value VAL or has no value.
3. Write in the output file an expanded declaration which states that the variable VAR defined as the expression EXP has the value VAL or has no value, in the format defined below. The expanded declarations must be written in the same order as the declarations from the input file from which they are generated.

5.2.1 Error handling

If your program thinks it has encountered an error in a declaration in the input, instead of writing an expanded declaration, your program should follow this error-handling procedure:

1. Ensure output for declarations before the error has already been written to the output file.
2. Write 1 newline character to the output file. Write `BAD INPUT:` to the output file (**not** followed by a newline character). Write a short description without newline characters of what is wrong to the output file. Write 1 newline character to the output file.
3. Write an error message on the standard error channel (file descriptor 2 on Linux).
4. Remember there was an error so that it can exit **later** with an error status.
5. Skip the rest of the erroneous declaration and process the rest of the input as if the erroneous declaration had not been present.

   (NOTE: This can be bad for real-world security. The purpose of continuing is to raise your mark.)

If your program encounters any other error that would prevent it from being able to correctly write an expanded declaration (e.g., an unexpected uncaught exception), your program should follow the same error-handling procedure as above except writing `ERROR:` instead of `BAD INPUT:`. Whether to use `BAD INPUT:` or `ERROR:` depends on where your program thinks the blame lies.

Your program should exit with an error status (which means a value from 1 to 127 on Linux) exactly when it has followed the error-handling procedure at least once.

Your program should never crash or abort except if it is run with unreasonable resource limits or an external component (e.g., the operating system or programming language implementation) malfunctions.

5.3 Common features of the input and output formats

- All input and output must be in plain text.
- Standard mathematical notation is used, except as indicated below.
- Variables are written as `x0`, `x1`, `x2`, … instead of x_0, x_1, x_2, ….
- The order in which members of a set construction expression or a set value are listed does not matter.
- The empty set is written as `{}` instead of $\varnothing$.
- Whitespace is ignored except that it is forbidden within tokens and required between alphanumeric tokens.
- The character `#` begins a comment that extends to the end of the line and is treated as a space.

5.4 The input formats

5.4.1 Choosing your input format

Your program's input will be in 4 files in the current directory:

- `input.txt` : mathematical plain text
- `input.json` : contains `input.txt` converted to JSON
- `simple-input.txt` : contains `input.txt` simplified
- `simple-input.json` : contains `input.txt` simplified and converted to JSON

Your program must read exactly 1 of these files and must ignore the others.

Your program must use either the JSON or simplified JSON input files.

(If you are an exceptional programmer and like challenges, and you include in your submission an email from me explicitly giving you permission to write a parser, you may use either text format.)

For parts 2 and 3, the top score for "functionality" can not be achieved if you use a simplified format. If you are a strong programmer, you might want to try for the top "functionality" score, otherwise the simplified format is recommended.

I will supply a program that converts the standard format to the 3 other formats so you can write your test data in the standard format.

5.4.2 Mathematical plain text input format

The plain text file `input.txt` will contain a sequence of declarations of this form:

```
Let VARIABLE be EXPRESSION.
```

The variable names are `x0`, `x1`, `x2`, and so on. Variable names begin with the letter `x` which is followed by a decimal number which starts with the digit `0` only if the entire number is 0 (e.g., `x005` is forbidden).

For parts 2 and 3 only, an expression may be a variable.

An expression may be a constructor of mathematical values:

- an integer
- an ordered pair constructor, written in the form (EXPRESSION, EXPRESSION)
- a finite set constructor, written in the form {EXPRESSION, EXPRESSION, ...}

NOTE: The above 3 cases (integer, ordered pair, finite set) are what your tree/graph data structure implementation must handle.

For part 1, the expressions inside ordered pairs and sets are restricted to be integers only.

Expressions may use parentheses, so (EXPRESSION) is an expression meaning the same as EXPRESSION.

For parts 2 and 3, expressions may also use these operations:

- Equality testing, written EXPRESSION = EXPRESSION.
- Set membership testing, written EXPRESSION $\in$ EXPRESSION.

For part 3 only, expressions may also use these operations:

- Function application, written in the form EXPRESSION EXPRESSION by putting the function and its argument side by side (possibly with whitespace in between if needed to separate tokens).
- Testing whether a value is a function, written `@is-function(EXPRESSION)`.
- A binary operator for calculating function spaces, written EXPRESSION $\rightarrow\!\!\!\!\!+$ EXPRESSION.
- Domain calculation for binary relations, written `@dom(EXPRESSION)`.
- A binary operator for set union, written EXPRESSION $\cup$ EXPRESSION.

- An unary operator for the inverse of a binary relation, written `@inverse(EXPRESSION)`.
- Diagonalization, written `@diagonalize(EXPRESSION,EXPRESSION,EXPRESSION,EXPRESSION)`.

All binary operators are left-associative except for function space calculation which is right-associative. Precedence of binary operators from highest to lowest is: function application, function space calculation, union, membership, equality.

Here is an example of what `input.txt` might look like:

```
Let x0 be 1.
Let x212 be (1, 2).
Let x3 be {x212, (3, 4)}.
Let x10 be {(0, 4), (1, 6)}.
Let x8 be {{8}}.
Let x8 be {x8, {x8}}.
Let x17 be {1, 2, x8}.
Let x18 be {x17, (1, x17)}.
Let x19 be (x18, x17).
Let x96 be (0 ∈ {{0}, 2 = 3} = x8) ∈ {2, 1}.
Let x7 be x3 x0.
Let x20 be {x19} ∪ x18.
Let x23 be @dom(x10).
Let x24 be @is-function(@inverse(x10)).
```

5.4.3 Simplified input format

The file `simple-input.txt` in the current directory will contain the same declarations as `input.txt`, except that complex expressions will be broken up. An expression is complex if it has a subexpression that is neither a number nor a variable. For example, the declaration

```
Let x8 be ((0,1),(3,(4,5))).
```

will be turned into a sequence of declarations like this:

```
Let x32 be (0,1).
Let x33 be (4,5).
Let x34 be (3,x33).
Let x8 be (x32,x34).
```

In this example, the variables `x32`, `x33`, and `x34` will be fresh (not already occurring in the input).

Use the simple input format if you can't figure out how to process the complex input using recursion. With the simple format, your evaluation procedure doesn't need recursion. (You still need recursion (or a stack, or more advanced techniques) for properly doing the equality and set membership tests.)

You switch input format to simple by prefixing **both** input **and** output file names with "`simple-`".

5.4.4 JSON input format

Pre-parsed input will be in files in the current directory. The file `input.json` will contain a JSON version of what is in `input.txt` and `simple-input.json` will contain a JSON version of what is in `simple-input.txt`. It is fairly clear how the JSON format corresponds to the mathematical plain text format. Ask if you have questions.

5.5 Output format

Your program's output must be a plain text sequence of expanded declarations of one of the two following forms (where VARIABLE, EXPRESSION and VALUE are replaced by their printed representations):

```
Let VARIABLE be EXPRESSION which is VALUE.

Let VARIABLE be EXPRESSION which has no value.
```

The tokens `Let`, `be`, `which is` or `which has no value`, and `.` (period, a.k.a. full stop) must appear in exactly that form. Do not alter spelling or capitalization or substitute other symbols. The VARIABLE and EXPRESSION part must be reproduced from what was in the declaration in the input. The new portion that your program must calculate is the VALUE or the absence of a value.

The EXPRESSION part can differ from the original input text in whitespace, unneeded parentheses, and order of subexpressions of set constructions.

In parts 2 and 3, when printing VALUE, members of any set must be listed at most once.

The output must be placed in the current directory in the file `output.txt` if the input file was `input.txt` or `input.json`, or in `simple-output.txt` if the input file was `simple-input.txt` or `simple-input.json`.

5.6 Evaluation rules

If an expression's value is not defined by the following rules, then the expression **has no value**.

The meanings of the value constructors are as follows:

- If EXP is the integer INT, then EXP has value INT, i.e., integers evaluate to themselves.
- Pair construction: If EXP is (E1, E2), and E1 has value V1, and E2 has value V2, then EXP's value is the pair (V1,V2).
- Set construction:
  - If EXP is { }, then EXP's value is the empty set { }.
  - If EXP is {E1}, and E1 has value V1, then EXP's value is the singleton set {V1}.
  - If EXP is {E1, EXPSEQ} where EXPSEQ is a comma-separated sequence of one or more expressions, and E1 has value V1, and {EXPSEQ} has value V2, and $\{V1\} \cup V2$ evaluates to V3, then EXP has value V3.

If there are one or more **preceding completed** declarations of a variable VAR (in the same input file), and the most recent one establishes that VAR has value VAL, then VAR evaluates to VAL (otherwise VAR **has no value**).

For deciding equality and set membership, integers, ordered pairs, and sets are considered to be distinct. This means every set is not equal to any integer.[1] Similarly, every ordered pair is not equal to any set, and every ordered pair is not equal to any integer.

The meanings of the operations are as follows:

- Equality testing: If EXP is E1 = E2:

[1]This is not the case in how mathematics is often built, but we will pretend this is true for this assignment. For example, often the natural number 2 is represented by the set $\{\emptyset,\{\emptyset\}\}$, so in that case $2 = \{\emptyset,\{\emptyset\}\}$ would be a true statement.

  - If E1 has value V1 and E2 has value V2, then EXP's value is 1 (meaning true) or 0 (meaning false) depending on whether V1 = V2.
  - If one of E1 or E2 has a value and the other has no value, then EXP's value is 0.
  - If each of E1 and E2 has no value, then EXP's value is 1.

- Set membership testing: If EXP is E1 $\in$ E2, and E1 has value V1, and E2 has value V2, then EXP's value is 1 (meaning true) or 0 (meaning false) depending on whether V1 $\in$ V2. For deciding this, integers and ordered pairs are considered to have no members, so the result is 0 if V2 is an integer or an ordered pair.
- Function application: If EXP is E1 E2, and E1 has value V1, and V1 is a set containing only ordered pairs, and E2 has value V2, and V1 contains an ordered pair (V2, V3), and V1 does not contain an ordered pair (V2, V4) where V3 $\neq$ V4, then EXP has the value V3.

  WARNING: This definition allows useful results when V1 is a binary relation but not a function.
- Function testing: If the subexpression has a value, this operation's value is 1 (meaning true) or 0 (meaning false) depending on whether the subexpression's value is a function.

  WARNING: Use the definition of function from the lecture notes. Do not vary from that definition.
- Function space: If EXP is E1 $\nrightarrow$ E2, and E1 has value V1, and E2 has value V2, and both V1 and V2 are sets, then EXP's value is the set V3 that contains only and exactly every function f such that dom(f) $\subseteq$ V1 and ran(f) $\subseteq$ V2.
- Domain calculation: If the subexpression's value is a set containing only ordered pairs, then this operation's value is the set of the first components of every such ordered pair.

  WARNING: Unlike some definitions of domain, this definition gives a useful result when the first argument evaluates to a binary relation that is not a function.
- Inverse: If the subexpression's value is a set V1 containing only ordered pairs, then this operation's value is the set V2 containing only ordered pairs such that for every value V3 and V4 it holds that (V3,V4) $\in$ V1 exactly when (V4,V3) $\in$ V2.

  WARNING: Unlike some definitions of inverse, this definition gives a useful result when the first argument evaluates to a binary relation that is not a function.
- Diagonalization: If the four subexpressions have the values V1, V2, V3, and V4, and V1 is a set, then this operation's value is a function F such that dom(F) $\subseteq$ V1 and for every i such that i $\in$ V1 it holds that:
  - If V2(i)(i) has no value, then F(i) = V4.
  - If V2(i)(i) has a value and V3(V2(i)(i)) has no value, then F(i) has no value.
  - If V3(V2(i)(i)) has a value, then F(i) = V3(V2(i)(i)).
- The others should all be obvious. Ask if they are not.

NOTE: Equality testing is the only operation that always has a value. All other operations have no value if even one of their subexpressions has no value. Function application has no value in additional circumstances.

If you have not yet attempted an operator, then all uses of that operator must have no value. Furthermore, you should begin your output file with a special "NOT ATTEMPTED" comment so that the test framework can avoid wasting your and my time with voluminous details on how the unattempted operator does not work. For example, if you have not attempted diagonalization for part 3, you would indicate that like this:

```
# NOT ATTEMPTED: diagonalize
```

REMINDER OF IMPORTANT VITAL REQUIREMENT: You **MUST** represent each of your internal intermediate results (which are integers, ordered pairs, and finite sets) as tree/graph nodes.

5.7 Example of program behavior

For the example `input.txt` given above, your program should write to `output.txt` something like this:

```
Let x0 be 1                         which is 1.
Let x212 be (1, 2)                  which is (1, 2).
Let x3 be {x212, (3, 4)}            which is {(1, 2), (3, 4)}.
Let x10 be {(0, 4), (1, 6)}         which is {(0, 4), (1, 6)}.
Let x8 be {{8}}                     which is {{8}}.
Let x8 be {x8, {x8}}                which is {{{8}}, {{{8}}}}.
Let x17 be {1, 2, x8}               which is {1, 2, {{{8}}, {{{8}}}}}.
Let x18 be {x17, (1, x17)}
  which is {(1, {1, 2, {{{8}}, {{{8}}}}}), {1, 2, {{{8}}, {{{8}}}}}}.
Let x19 be (x18, x17)
  which is ({(1, {1, 2, {{{8}}, {{{8}}}}}), {1, 2, {{{8}}, {{{8}}}}}},
            {1, 2, {{{8}}, {{{8}}}}}).
Let x96 be (0 ∈ {{0}, 2 = 3} = x8) ∈ {2, 1}
  which is 0.
Let x7 be x3 x0
  which is 2.
Let x20 be {x19} ∪ x18
  which is {(1, {1, 2, {{{8}}, {{{8}}}}}),
            ({(1, {1, 2, {{{8}}, {{{8}}}}}), {1, 2, {{{8}}, {{{8}}}}}},
             {1, 2, {{{8}}, {{{8}}}}}),
            {1, 2, {{{8}}, {{{8}}}}}}.
Let x23 be @dom(x10)                          which is {0, 1}.
Let x24 be @is-function(@inverse(x10))        which is 1.
```

5.8 Explaining why/whether/when diagonalization works (part 3 only)

For part 3 your report must additionally explain why diagonalization works, when it does work. Your explanation must cover at least these points:

- Under what conditions does `@diagonalize(E1,E2,E3,E4)` have as its value a function F such that F does not belong to the range of E2? List as many relevant conditions as you can. Avoid listing unnecessary or redundant conditions.
- Why is this the same as the method of "diagonalization" introduced by Cantor? Is it only the same under some conditions? If so, what conditions?

Although your implementation will only work for inputs that are finite, consider all cases where the inputs can be or can contain infinite sets. Explain your reasoning.

6 General conditions

6.1 Marking criteria

- **Functionality**: This judges how well the non-explanatory content fulfills the specification.

  For a program this is based primarily on its performance, which is judged both dynamically (by testing) and statically (by reading the code).

For test data this is judged primarily on how thoroughly it tests whether a program fulfills its specification.

For a report this is judged primarily on how well any formal mathematical content meets requirements.

- **Design**: This judges the overall conception of how everything fits together, i.e., the *gestalt* of the work: how the student arranged for the overall fulfillment of the specification to emerge from the parts.
- **Explanation**: This judges how well each student explains and evaluates how and why their work satisfies or fails to satisfy the specification.
- **Presentation**: This judges how well the work makes clear how well it satisfies the requirements.

6.2 Files to submit

6.2.1 Source code

- Your source code should reside in as few files as possible (ideally 1 file) and with as few subdirectories as possible (ideally no subdirectories at all).
- Your main source code file must be named exactly as specified here with no differences whatsoever!
  - If your program is an executable script that does not need any compiling, then its main source code file must be named exactly "`program`" (**without** any extension).
  - If your program needs to be compiled or somehow assembled before it can be run, then its main source code file must be named exactly "`program.EXTENSION`" where .EXTENSION is a file name extension commonly used for source code in your programming language.
- Your program must build (if needed) and run on the HWU MACS CS department's Linux computers.
- Your source code must be properly commented, tidy, and well written.
- All identifiers must have meaningful names.
- Indentation must follow rigorous rules.
- Lines longer than 100 characters should be avoided when reasonably possible while also keeping logical indentation and line breaks.

6.2.2 Report

- Explanatory content that does not more sensibly belong in another file can go in your report.
- Your report must be in a file named "`report.pdf`". The name must be exactly as specified here. Your report must be in the standard Adobe PDF format.
- Your report must be formatted for A4 paper, with all marks at least 2.5 cm from the edge of the page.
- If you have space at the end within your page limit try using a bigger font to make it easier to read.
- Your report must **not** have a title page.
- Your report must be well written, clear, and easy to read.

6.2.3 Makefile

- If your source code needs to be compiled before it can be run, then you must submit a makefile.
- A makefile must be named exactly "`Makefile`" with no variation whatsoever. Capitalize and spell the name exactly as indicated.
- A makefile must be in the top-level directory of your submission, not in a subdirectory.
- If you have a makefile, then running GNU `make` in a directory containing your submitted files must ensure the existence of a directly runnable file in that directory named "`program`". The runnable file's name must have **no** extension.

6.2.4 Test data

- You must submit test data for testing your program **and also** other students' programs.
- Your test data must be in the mathematical plain text input format (**NOT** the JSON format).
- Test data you submit must be valid input **without** syntax errors. (You should test your program on invalid inputs (this will be in JSON format for most of you), but keep such test data to yourself.)
- Your test data submitted for a particular part must not use features that are not required until a later part.
- Ideally your test data should exercise every branch in your program. (See "Test Case Design" in the F28SD (Software Design) notes.)
- Your test data should include all of the cases your program fails on with comments explaining why.
- Your test data must contain meaningful and helpful comments.
- You **MUST** give attribution for every bit of test data you did not author.
- Avoid submitting test data you did not author. Only submit such test data if you need to add comments (e.g., to explain failures), and if you do you must give attribution.
- Let us say that some test data is *poisonous* for your program if handling it can cause your program to enter a bad state that can prevent your program from correctly handling later test data in the same file that should not depend on the results of the poisonous test data. For example, if your program crashes on some test data then the test data is poisonous. Test data is probably poisonous if handling it causes your program to corrupt its data structures. Your program can produce the wrong result for test data without it necessarily being poisonous. The idea is that non-poisonous test declarations that do not share variables between them can exhibit both successes and failures while safely coexisting in the same file.
  - Submit all your non-poisonous test data in a file named "`test-data.txt`". Your program must not crash on this input.
  - Submit all of your poisonous test data in a file named "`test-data-poison.txt`".
- The student whose test data crashes the most other students' programs will get their "functionality" score raised (if possible). The same applies to the student whose test data causes the most other students' programs to behave wrongly (either crash or produce incorrect output). Resource-limit-exceeded crashes due primarily to huge test data or output do not count and might disqualify test data from this award.
- Your test data will normally be provided to other students to help them understand their errors and also to help them improve their later submissions. By default this will be anonymous, which will be automatically implemented by removing comment lines from the beginning of the file until a non-comment line is seen. So begin your test data like this:

```
# Author: Given Names Surname

# (Notice the blank line just above after the name comment!)
# Here go comments explaining the test data.  ...
Let x1 be (0,1). # ← beginning of actual test data
```

If you prefer other students to see your name, then express this wish in a comment in your test data file, e.g., begin like this:

```
# (This line at the file start will be automatically removed.)

# (Notice the blank line just above after the dummy comment!)
# Author: Given Names Surname
# I prefer that the test data I provide is not anonymous.
Let x1 be (0,1). # ← beginning of actual test data
```

6.2.5 Permission emails

- Each permission email giving you explicit permission to vary the assignment must be submitted in a file named "`permission-email-NUM.txt`" where NUM is a number (1, 2, 3, ...).

6.2.6 Forbidden files

- Do not submit any files (including directories) whose names contain "`done`", "`expected`", "`input`", "`output`", "`status`", or "`test`" as substrings, except as specified above for test data files. Avoid Java class names containing these strings (because Java will make a `.class` file named after each class).
- Do not submit any files that are wholly authored by others (e.g., 3rd-party libraries).
- Submit **ONLY** human-readable plain text files (e.g., program source code) or PDF reports.

6.3 Explanatory content (mainly the report)

Your report and other files should show a deep critical analytic understanding that includes things like:

- What was required? (Do not just parrot the assignment.)
- To what extent does your work correctly achieve what was required, and how does it achieve this, and how do you know it does?
- In what ways does your work fail to achieve what was required, and why?
- Why did you do your work this way rather than the alternatives? Which choices were significant and what do you think of them now? (This is **not** asking why you chose your programming tools.)
- What are the most significant properties that are true or not true of your work (for good or ill)?
- What are the significant aspects or parts of your work and what is significant about how they combine?

Concentrate more on the non-obvious and significant, and less on restating things that are easy to see.

Cover the entirety of what was required, not just what has changed since the previous part.

6.4 Programming language and libraries

- You may program in Java or SML.
- If you want to use another programming language, you must do these things:
  - Your submission must include an email from me explicitly giving you permission to do so.
  - You must verify or ensure that your choice of language is installed system-wide on the HWU MACS CS department's Linux computers.
- You must verify or ensure that any 3rd-party libraries you need are installed system-wide on the HWU MACS CS department's Linux computers.

6.5 File formats for input/output

- All files that are input to or output from your program must be human-readable plain text files, or (semi-human-readable) plain text JSON files generated from human-readable plain text files.

6.6 Plain text

- Plain text must consist of characters in the UTF-8 encoding of the Unicode character set.
- You must not use the characters U+000D ("carriage return" (CR)) and U+FEFF ("zero width no-break space" (ZWNBSP), also known as "byte order mark" (BOM)).
- You must only use character U+0009 ("character tabulation", also known as "horizontal tabulation" (HT) and "tab") in a makefile and there only as the first character on a line.

6.7 Submission

- You must submit your files by copying them into a directory that will have been created for you on the HWU MACS CS department's NFS file systems.

  I will provide a program that does the needed copying.
- Except where explicitly indicated otherwise, I follow the default MACS school policy for late coursework (i.e., 10% of the maximum mark is deducted for each working day late, and work turned in after 5 days gets a mark of zero).
- Do **NOT** email me coursework submissions! Emailing me your work does **NOT** count as submitting it.

6.8 Testing and assessment

- The assignment is **NOT** to handle just test data made available to you in advance. Marks are **NOT** based on this. You must fulfill the specification.
- Software I make available to you in advance to help you test your program might not implement the specification correctly. You are responsible for understanding the specification and determining for yourself whether any particular input or output data is valid or whether any particular result is correct.