Mathematical Expression Evaluator (Assessed Individual Coursework, Foundations 2 (F29FB), Spring 2019)

2019-01-10 (Thursday)

1 Individual work and attribution

- Except as permitted by your lecturers explicitly in writing, you must write any source code, report, and test data that you submit wholly by yourself, individually, as the sole author. English text must be in your own words. Code and test data must be in your own tokens and logic.
- Except as specified by your lecturers explicitly in writing, 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 screen, photograph, computer file, GitHub, etc. You must refuse and not request such sharing. You must make reasonable efforts to secure your work on this assignment against copying.
- To the extent that it is reasonably possible, you must properly attribute and reference the originator(s) (including yourself) of every bit of what you submit. You must explicitly mark the boundaries (beginning **and** end) of every bit that you are not author of.
- 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 Purpose and 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, implementing, and testing data structures to model mathematical entities.

2.1 Example of program behavior

Here is an example of what your program's input in the file `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)).
```

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.
```

3 Due dates, mark weighting, report length limits, and feedback

part:	1	2	3
mark weight:	10%	35%	55%
preliminary test data due in week:	3	5	8
full submission due in week:	4	7	12
report page limit:	1	2	3

- Unless a variation has been specified in writing by your lecturers, all deadlines will be at 15:15 (UK time) on Wednesday in Edinburgh and at 15:00 (UAE time) on Wednesday in Dubai.
- NOTE: It is explained later in the specification what should be covered by (1) your preliminary test data for part K and (2) the test data you include in your full submission for part K.

- WARNING: Late submission of preliminary test data is never accepted.
- WARNING: The page limit on reports will be enforced by automatically taking only the first K pages from the submitted PDF file.
- Each part's mark is 70% functional quality, 25% non-functional quality, and 5% preliminary test data.
- Written individual feedback for each part will consist of the 3 numeric marks and very brief comments on one or two points that affected the marks.
- For feedback, preliminary test data will be treated as if it were part of the later full submission.
- Continuous feedback will be available from the testing software provided by your lecturers.
- If non-obvious points affected many students' marks, brief explanations will be given to the whole course.
- Except where otherwise specified, standard MACS school coursework policies apply.

4 Recommended guidance: A procedure for doing the coursework

The steps listed below, if followed, give you a good chance of getting a good mark. This is not the only way to do the coursework. This section is recommended guidance, and is not required.

NOTE: Because of the Dubai IT Lab 1 situation, the guidance for Dubai students is a bit different.

1. All commands mentioned are to be run in a terminal running the bash shell on one of the Linux computers provided for HWU MACS CS students to use. Ensure you know what this means.
2. Configure Foundations 2 tool support. Run the command: "~jbw/f2 init ; source ~/.bashrc".

   Dubai: If the "f2" command is not already working on the computer you happen to be using, download it, name it "f2", ensure it is in the shell's current directory, and run the command "bash f2 init ; source ~/.bashrc".
3. Choose a new, empty directory (folder) to keep your work in. Let D stand for the full absolute path to this directory.
   - If you use Eclipse, D will likely be a subdirectory of your Eclipse workspace.

   Dubai: D needs to begin with `/media/` so it will be on your USB flash drive.
4. Set your directory. Run the command: "f2 set-dir D".

   Dubai: Skip this step. Instead, ensure the shell's current directory is D before running any commands mentioned below. The command "cd D" will do this.
5. Optional, but recommended: Get sample files. Run the command: "f2 get-samples".
6. Choose your programming language: Standard ML (SML) or Java. Let E stand for the usual file name extension for source code files in this language: "sml" or "java".
7. Set your language. Run the command: "f2 set-lang E".
8. There are six due dates: 1-prelim, 1-full, 2-prelim, 2-full, 3-prelim, 3-full.

   For each due date K-S where the part K $\in$ {1,2,3} and the subpart S $\in$ {prelim, full}, do the following:

   (a) Familiarize yourself with the additions to the input language in part K specified in the rest of this assignment.

   (b) If the subpart S = prelim, then create or update the file D/test-data.txt.

- This file must contain thorough tests for **every** student's program for all part-K features.

(c) If the subpart S = full, then create or update the files D/program.E, D/test-data.txt, D/report.pdf, and D/Makefile.
- The file D/test-data.txt must now contain tests that reach **every** reachable branch of D/program.E.
- Most students will modify at most 1 word in D/Makefile.

(d) Ensure your files satisfy the specification in the rest of this assignment.

(e) Pack the files and get feedback on the pack from testing. Run the command: "f2 pack-and-get-feedback K S".

(f) Optional, but recommended: Review the output from testing.

(g) Optional, but recommended: If you are unhappy with the quality of your files, repeat steps from A through F above as many times as necessary.

(h) Upload the pack file D/submission-K-S.tar.gz to VISION.

5 Helpful summary: Differences between parts

This section is redundant and merely repeats details from the specification.

- For part 1:
  - Expressions can only be: natural numbers, ordered pairs of natural numbers, sets of natural numbers.
  - It therefore makes no difference whether you use the simplified input.
- Part 2 adds:
  - Set and ordered pair expressions can contain any expression, including sets and ordered pairs.
  - Expressions can also be variables and tests of equality or membership.
  - The simplified input is actually simpler.
- Part 3 adds:
  - Expressions can be formed using additional operators.
  - The submission must contain text with specified reasoning about the diagonalize operator.

6 Specification

6.1 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 (whether input expression or result value) in some sense 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 (whether input expression or result value) as a tree/graph node such that for each member of the set there is a child of the node that represents it.

  NOTE: The requirement to use tree/graph nodes applies only to ordered pairs and sets.

  WARNING: The concept of "tree" taught in previous courses has a lot of stuff in it that you will not need. All you need are nodes that can have zero or more other nodes as children and can contain a couple of pieces of other data. 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.

- At every point in the execution of your program after the input is parsed by the JSON parser, your program must **NOT** hold data that represents any ordered pair or set (whether expression or value) as a string or as an array/list/sequence whose elements are all bytes/characters/symbols/tokens or in any other way that would be considered serialized/unparsed/marshaled/pickled.
- You must implement a subroutine (possibly as a bunch of methods spread across a number of classes) that traverses the representation of a mathematical expression or value and **during the traversal outputs** the individual tokens of a printed representation of that expression or value 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 by string literals in your program.
- You may use your programming language's function(s) for converting natural numbers to strings.
- You may use strings created by a JSON parser.

NOTE: Outputting and comparing strings are both allowed.

WARNING: For Java this means using + on strings is forbidden and nearly all uses of toString are forbidden, as well as anything that might do this implicitly, e.g., the print method of anything that is not a natural number.

NO BUILDING STRINGS! NO BUILDING STRINGS! NO BUILDING STRINGS! NO BUILDING STRINGS! NO BUILDING STRINGS! NO BUILDING STRINGS!

6.2 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.

6.2.1 Error handling

If your program thinks it has encountered an error, your program should follow this error-handling procedure:

1. For declarations handled before the error, ensure the output has already been written to the output file.

2. Write "." followed by 1 newline character to the output file. Write `ERROR:` 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 "." followed by 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 your program can exit **later** with an error status.
5. If your program was handling a declaration when the error was encountered, skip the rest of the handling of this declaration and process the rest of the input as if the skipped declaration had not been present.

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

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.

6.3 Common features of the input and output formats

- 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.

6.4 The input formats

6.4.1 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 the 2 JSON input files and must ignore the others.

Stronger programmers should use the standard (unsimplified) format to get test output that matches the original input. Weaker programmers should use the simplified format to reduce the amount of recursion that is needed.

Your lecturers will supply a program that converts the mathematical plain text input format to the 3 other formats.

NOTE: You must write and submit your test data as mathematical plain text.

6.4.2 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 MUST begin with the lowercase letter `x` which MUST be followed by an ASCII decimal natural number written with as few digits as possible (i.e, it starts with the digit `0` only if the entire number is written "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:

- a decimal natural number written in the standard way using only ASCII decimal digits and with as few characters as possible,
- an ordered pair constructor, written in the form (EXPRESSION, EXPRESSION), or
- a finite set constructor, written in the form {EXPRESSION, EXPRESSION, ... }.

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

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

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

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:

- A binary operator for set union, written EXPRESSION $\cup$ EXPRESSION.
- 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).`
- Optional, for a mark top-up bonus of up to 5%: Diagonalization, the only 4-parameter operator, written `diagonalize(EXPRESSION,EXPRESSION,EXPRESSION,EXPRESSION).`
- Domain calculation for binary relations, written `dom(EXPRESSION).`
- An unary operator for the inverse of a binary relation, written `inverse(EXPRESSION).`
- Optional, for a mark top-up bonus of up to 5%: A binary operator for calculating function spaces, written EXPRESSION $\nrightarrow$ 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.

6.4.3 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-`".

6.4.4 JSON input format

Easier-to-parse input files will also be 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`. Both JSON files will look overall like this:

```
{ "type" : "declaration-list", "declaration-list" : [JDEC_1, ..., JDEC_k] }
```

Each JSON-format declaration `JDEC` looks like this, where `NAT` is some natural number:

```
{ "type" : "declaration", "declared-variable" : NAT, "expression" : JEXP }
```

Each JSON-format expression `JEXP` is in one of the 3 following shapes, where `OPER` is a JSON string:

```
( "type" : "expression", "expression-case" : "natural-number",
  "natural-number" : NAT }

{ "type" : "expression", "expression-case" : "variable",
  "used-variable" : NAT }

{ "type" : "expression", "expression-case" : "operation",
  "operator" : OPER, "arguments" : [JEXP_1, ..., JEXP_k] }
```

It should be fairly clear how the JSON format corresponds to the mathematical plain text format. Ask if you have questions.

6.5 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 in the indicated order. 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 is allowed to differ from the original input text in whitespace, unneeded parentheses, and order of subexpressions of set constructions.

VALUE is printed as an expression not mentioning variables and built only using the constructors for natural numbers, ordered pairs, and sets. 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`.

6.6 Evaluation rules

For this coursework, we consider each value to be exactly one of: a natural number, an ordered pair of two values, or a set of values. The logical domain of discourse (the range of possible meanings for expressions) is the values together with the one additional possibility of "not having a value".

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

The evaluation rules for the value constructors are as follows:

- If EXP is the natural number NAT, then EXP has value NAT, i.e., natural numbers evaluate to themselves.
- Ordered pair construction: If EXP is (E1, E2), and E1 has value V1, and E2 has value V2, then EXP's value is the ordered pair (V1,V2).
- Set construction:
  - If EXP is {}, then EXP's value is the empty set {}, which is the only set value that has no members.
  - If EXP is {E1}, and E1 has value V1, then EXP's value is the singleton set {V1}, which is the the only set value that has V1 as its sole member.
  - 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 V3 = {V1} ∪ V2,[1] then EXP has value V3.

If there are one or more **earlier, 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, natural numbers, ordered pairs, and sets are considered to be distinct. This means every set is not equal to any natural number.[2] Similarly, every ordered pair is not equal to any set, and every ordered pair is not equal to any natural number.

[1] This means V3 is the only set value such that for every value V4 it holds that V4 ∈ V3 if and only if either V4 = V1 or V4 ∈ V2.

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

The evaluation rules for the other operations are as follows:

- Equality testing: If EXP is E1 = E2:
  - 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, natural numbers and ordered pairs are considered to have no members, so the result is 0 if V2 is an natural number 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.

  WARNING: The size of the result is huge: |E1 $\nrightarrow$ E2|=(|E2|+1)$^{|E1|}$.
- 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 the function F such that dom(F) $\subseteq$ V1 and for every 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)).

  NOTE: The function applications here (V2(i), V2(i)(i), V3(V2(i)(i)), and F(i)) are all evaluated following the rule for function application given above.
- Union should be obvious. Ask if it is 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.

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

6.7 Formally stating why/whether/when the `diagonalize` operator "works" (part 3 only)

For part 3 your report must additionally formally state why the `diagonalize` operator "works", when it does "work". Your formal statement must cover at least these points:

- Under what conditions does the `diagonalize(E1,E2,E3,E4)` operation as defined in this coursework 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 the `diagonalize` operator 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.

WARNING: This work is mandatory. Your submission must include the formal statement described above, regardless of whether you have attempted or completed implementation of the `diagonalize` operator.

WARNING: This is asking about the `diagonalize` operator described above. The only aspect of the method of "diagonalization" that is relevant here is how it is related to the `diagonalize` operator.

7 Specification: General conditions

7.1 Files to submit

7.1.1 Source code (program, makefile, test data)

- Your source code must be properly commented, tidy, and well written.
- Indentation must follow rules that are rigorous, logical, and reasonable.
- Lines longer than 100 characters should be avoided when reasonably possible while also keeping logical indentation and line breaks.

7.1.1.1 Program

- Your program should reside in as few files as possible (ideally 1 file) and with as few subdirectories as possible (ideally no subdirectories at all).

  WARNING: For Java this means it is best to not have a package declaration. If you let Eclipse put your code in a package, you'll need to drag your file out of the package with the mouse.
- Your main program file must be named exactly "`program.EXTENSION`" where .EXTENSION is the file name extension most commonly used for source code in your programming language. Your main program file must be named exactly as specified here with no differences whatsoever!
- Your program must build (if needed) and run on the Linux computers provided for HWU MACS CS students to use.
- All identifiers must have names that are meaningful to readers other than the author.

7.1.1.2 Makefile

- You must submit a makefile named exactly "`Makefile`" with no variation whatsoever. Capitalize and spell the name exactly as indicated.
- Your makefile must be in the top-level directory of your submission, not in a subdirectory.
- Running GNU `make` in a directory containing your submitted files must ensure the existence of a directly runnable file (possibly a symbolic link) in that directory named "`program.exe`", together with any necessary supporting files.

7.1.1.3 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 of course test your program on invalid inputs (this will be in JSON format unless you have permission to write a parser), but keep such test data to yourself.)
- Your test data file must be named exactly "`test-data.txt`" with no variation whatsoever.
- Your test data for each part will be submitted in two stages: preliminary and final.
  - Your preliminary test data for part K should give good coverage of the part-K specification and contain thorough tests for **every** student's program for all part-K features.
    WARNING: Late submission of preliminary test data is never accepted.
  - Your final test data for part K is the test data you include in your full submission for part K. In addition to fulfilling the requirements for preliminary test data, your final test data must also provide good coverage of your actual submitted program. It must contain tests that exercise **every** branch in your program that is reachable for a valid input. You should write a comment in each program branch identifying 1 test that reaches it or an explanation of why no test reaches it. (See "Test Case Design" in the F28SD (Software Design) notes. Search for "test coverage" on the web.) Your final test data should include all of the cases that make your program fail with comments explaining why.
- Your test data will normally be provided to all students in the course to help them understand their errors and also to help them improve their later submissions.
- 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.
- Because your test data is source code, it must contain meaningful and helpful comments.
- 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.

  Put test data in this order in your test data file:
  - test data that is not poisonous for your program
  - poisonous test data that corrupts your program
  - poisonous test data that crashes your program

7.1.2 Report

- Your report must contain an anti-plagiarism statement in which you demonstrate that you correctly understand the relevance to this coursework of these words:

  individual, author, attribute, reference, collusion, plagiarism, penalty

  The underlined portion of each word must appear in your statement.

- Text that does not more sensibly belong in another file must 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 should **not** have a title page.
- Your report must be well written, clear, and easy to read.

7.1.3 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, ...).

7.1.4 Forbidden files

- Do not submit any files (including directories) whose names contain "`.class`", "`done`", "`.exe`", "`expected`", "`input`", "`.jar`", "`manifest`", "`out`", "`status`", "`test`", or "`wrapper`" 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), except for the sample makefile supplied by your lecturers.
- Submit **ONLY** human-readable plain text files (e.g., program source code) or PDF reports.

7.2 Explanatory content (mainly the report)

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

- In mathematical terms, what was required? (Avoid just parroting the assignment.)
- In mathematical terms, to what extent does your work correctly achieve what was required, and how does it achieve this, and how do you know it does? (Avoid just narrating your source code or restating your testing results.)
- In mathematical terms, in what ways does your work fail to achieve what was required, and why? (Avoid just listing your crashes and wrong outputs.)
- In mathematical terms, 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 or about your time-management skills.)

- In mathematical terms, what are the most significant properties that are true or not true of your work (for good or ill)?
- In mathematical terms, 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.

7.3 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 for each part must include an email from your lecturer explicitly giving you permission to do so.
  - You must ensure that your choice of language is installed system-wide on the Linux computers provided for HWU MACS CS students to use.
- You must ensure that any 3rd-party libraries you need are installed system-wide on the Linux computers provided for HWU MACS CS students to use.
- You must not use libraries that solve exactly major parts of the assignment.

7.4 Use of student ID in place of name

- You must use your student ID in everything you submit for this course in place of your name, your computer account name, your email address, etc.

7.5 Submission

- You must pack your files to be submitted (even if there is only one file) in a directory inside a gzip-compressed GNU-tar archive file. If the part K $\in$ {1,2,3} and the subpart S $\in$ {prelim, full}, the bash shell command "`~jbw/f2 pack-and-test K S`" will do this for you. You must upload the archive file to VISION.
- Do **NOT** email your lecturers coursework submissions! Emailing your work does **NOT** count as submitting it.

7.6 Marking criteria

- **Functional quality**: This is judged by criteria like the following.
  - **Performance**: This judges how well the non-explanatory content fulfills the specification.
    For a program this is based primarily on its input/output behavior, 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.

- **Non-functional quality**: This is judged by criteria like the following.

  - **Explanatory evaluation**: 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 is arranged so that it is easy to see how well it satisfies the requirements.

7.7 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 your lecturers 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.

7.8 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.

7.9 Plain text

- Plain text must consist of characters in the UTF-8 encoding of the Unicode character set.
- Plain text 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)).
- Plain text 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.