Mathematical Expression Evaluator (Assessed Individual Coursework, Foundations 2 (F29FB), Spring 2017)

2017-01-10 (Tuesday)

1 Individual work and attribution

- You must write the source code, report, and test cases 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.
- You must explicitly attribute and properly reference in writing every bit of what you submit that you did not originate where it is reasonably possible to identify a specific source.
- 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, whether 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 make requests for such sharing. You must take steps to 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 0 This part will count for 10% of the mark. This part is due at 2016-01-24 T 15:15 (Tuesday, week 3).

Your part 0 report must not exceed 1 page.

Assessment will be by a personal interview after the work is submitted. The interview is to be scheduled by 2017-02-02 (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 1 This part will count for 35% of the mark. This part is due at 2016-02-14 T 15:15 (Tuesday, week 6).

Your part 1 report must not exceed 2 pages.

Part 2 This part will count for 55% of the mark. This part is due at 2016-03-21 T 15:15 (Tuesday, week 11).

Your part 2 report must not exceed 4 pages.

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

	part 0	part 1	part 2
functionality	60%	40%	40%
explanation	10%	20%	20%
design	20%	20%	20%
presentation	10%	20%	20%

4 Specification

4.1 Requirement to use trees 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 design and implement a tree data structure capable of representing all the mathematical values that can be built from integers using ordered pairs and finite sets.
- You must implement a subroutine that traverses your data structure and outputs a printed representation of the data structure using standard mathematical notation.

To help ensure that you genuinely make trees, you are absolutely forbidden from building or altering character arrays 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.
- If you use the JSON input format (which you must do unless you have permission to use the plain text format), you may use strings created by the JSON parser.
- You may create and alter strings or character arrays if you include in your submission an email message from me stating explicit permission for the way you create or alter strings or character arrays.

Note that outputting and comparing strings are both allowed.

4.2 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 when your program starts and will not change while your program runs.

Your program file will be run directly with no arguments.

When your program exits, the output file must exist in the current directory with the correct contents.

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

1. Calculate the value VAL that results from evaluating the expression EXP using the rules given below.
2. Remember the value VAL as the value of the variable VAR for use in later definitions.
3. Write in the output file an expanded definition which states that the variable VAR defined as the expression EXP has the value VAL, in the format defined below. The expanded definitions must be written in the output file in the same order as the definitions from the input file from which they are generated.

4.2.1 Error handling

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

1. Ensure output for definitions 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 of what your program thinks is wrong to the output file. This description must not contain any newline characters. Write 1 newline character to the output file.
3. Write an error message on the standard error channel (file descriptor 2 on a UNIX system like Linux).
4. Remember there was an error so that it can exit **later** with an error status.
5. Skip the rest of the erroneous definition and process the rest of the input as if the erroneous definition had not been present.

   (NOTE: This can be bad for real-world programs for reasons such as security. This is to raise your mark when you mistakenly think the input is wrong but your program can succeed on the rest of the input.)

If your program encounters any other error that would prevent it from being able to correctly write an expanded definition (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 whether your program thinks the blame lies with the input or itself.

Your program should exit with an error status (which means the exit status should not be zero on a UNIX system like Linux) exactly when it has followed the error-handling procedure at least once.

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

4.3 Common features of the input and output formats

- All input and output must be in plain text.
- Standard mathematical notation is used, except numbers that make variables distinct are not subscripted.
- The order in which members of a set value are listed does not matter.
- The empty set is written as `{}`.
- Whitespace is ignored except that it separates tokens.
- The character `#` begins a comment that extends to the end of the line and is treated as whitespace.

4.4 The input formats

4.4.1 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.

Normally your program must use either the JSON or simplified JSON input files. However, if you are an exceptional programmer and like a challenge, and you include with your submission an email from me explicitly giving you permission to write a parser for the text format, you may use either text format; this will not improve your mark (and indeed might lower it due to the extra work required).

For parts 1 and 2, the top score for "functionality" can not be achieved if you use a simplified format.

I will supply a program named `format-converter` that will convert the standard format to the 3 other formats, and you can use this to convert your own test cases to the format you use.

4.4.2 Mathematical plain text input format

The plain text file `input.txt` will contain a sequence of definitions 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.

For parts 1 and 2 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, ...}

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

For part 0, 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 1 and 2, expressions may also use these operations:

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

For part 2 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 (a binary relation satisfying a particular property), which is written `@is-function(EXPRESSION)`.
- Domain calculation for binary relations, written in the form `@dom(EXPRESSION)`.
- A binary operator for set union, written as EXPRESSION ∪ EXPRESSION.
- A binary operator for set difference (a.k.a. complement), written as EXPRESSION \ EXPRESSION.
- An unary operator for the inverse of binary relations, written in the form `@inverse(EXPRESSION)`.
- Diagonalization, written as `@diagonalize(EXPRESSION,EXPRESSION,EXPRESSION,EXPRESSION)`.

All binary operators are left-associative. Precedence of binary operators from highest to lowest is: function application, set difference, 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 x21 be x20 \ {x17}.
Let x23 be @dom(x10).
Let x24 be @is-function(@inverse(x10)).
```

4.4.3 Simplified input format

The file `simple-input.txt` in the current directory will contain the same statements 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 definition

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

will be turned into a sequence of definitions 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-`".

4.4.4 JSON input format

Pre-parsed input will be in files in the current directory with the following names:

```
input.json
simple-input.json
```

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 plain text format, so I am not documenting it. Ask if you have questions.

4.5 Output format

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

```
Let VARIABLE be EXPRESSION
  which is VALUE.
```

The tokens `Let`, `be`, `which is`, and `.` 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 definition in the input. The new portion that your program must calculate is the VALUE.

In parts 1 and 2, when printing VALUE, members of any set must be listed at most once.

If the value is not defined, then `undefined` must be printed.

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`.

4.6 Evaluation rules

Integers evaluate to themselves.

A variable evaluates to the value given it by the most recent **preceding** definition of that variable in the same input file, **or has no value** (i.e., is undefined) if there is no such definition.

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 the left subexpression has value VL, and the right subexpression has value VR, this operation's value is 1 (meaning true) or 0 (meaning false) depending on whether VL = VR.

[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 {∅,{∅}}, so in that case 2 = {∅,{∅}} would be a true statement.

- Set membership testing: If the left subexpression has value VL, and the right subexpression has value VR, this operation's value is 1 (meaning true) or 0 (meaning false) depending on whether VL $\in$ VR.

  For deciding this, integers and ordered pairs are considered to have no members, so the result is 0 if VR is an integer or an ordered pair.

- Function application: If the left subexpression has the value VL, and VL is a set containing only ordered pairs, and the right subexpression has the value VR, and VL contains an ordered pair (VR, V1), and VL does not contain an ordered pair (VR, V2) where V1 $\neq$ V2, then this operation has the value V1.

  WARNING: This is more general than the operation defined in the lecture notes, because a non-function can be used provided it is functional on the inputs it is used with.

- 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.

- 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: In the lecture notes, "domain" is only defined for functions, whereas here it is also defined for sets of pairs that are not functions.

- 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) is undefined, then F(i) = V4.
  - If V2(i)(i) is defined and V3(V2(i)(i)) is undefined, then F(i) is undefined.
  - If V3(V2(i)(i)) is defined, then F(i) = V3(V2(i)(i)).

- The others should all be obvious. Ask if they are not.

Note that if a subexpression of an operation has no value (i.e., is undefined), then the operation has no value.

If you have not yet implemented an operator, then all uses of that operator must have no value.

REMINDER OF IMPORTANT VITAL REQUIREMENT: You **MUST** implement your internal intermediate results (which are integers, ordered pairs, and finite sets) using your tree data structure.

4.7 Example of program behavior

For the example file `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 x21 be x20 \ {x17}
  which is {(1, {1, 2, {{{8}}, {{{8}}}}}),
             ({(1, {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.
```

4.8 Explaining why/whether/when diagonalization works (part 2 only)

For part 2 your report must additionally explain why diagonalization works, when it does work. Your explanation must address at least these points:

- Under what conditions does `@diagonalize(E1,E2,E3,E4)` return a function F such that F does not belong to the range of E2? Exhaustively list all needed conditions. 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.

5 General conditions

5.1 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 close any formal mathematical content comes to fulfilling the requirements.
- EXPLANATION: This judges how well each student explains and evaluates how and why their work satisfies or fails to satisfy the specification.
- 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.
- PRESENTATION: This judges how well the work makes clear how well it satisfies the requirements.

5.2 Files to submit

5.2.1 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 where PART is replaced by the part number. The name must not differ in any way from what is specified here.
  - If your program is an executable script that does not need any compiling, then its main source code file must be named exactly "`program-part-PART`" (**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-part-PART.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.

5.2.2 Report

- Your report must be in a file named "`report-part-PART.pdf`" where PART is the part number. 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 ink marks at least 2.5 cm from the edge of the paper.
- 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.

5.2.3 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 invoking GNU `make` in a directory containing your submitted files must ensure the existence of a directly runnable file in that directory named "`program-part-PART`" where PART is replaced by the part number. The runnable file's name must have **no** extension.

5.2.4 Test cases

- You must submit test cases for testing your program and other students' programs.
- Your test cases must be in the mathematical plain text input format (**NOT** the JSON format).
- Test cases 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 cases to yourself.)
- Your test cases submitted for a particular part must not use features that are not required until a later part.
- Ideally your test cases should exercise every branch in your program.
- Your test cases should include all of the cases your program fails on with comments explaining why.
- Your test cases must contain comments.
- You **MUST** give attribution for every test case you did not author.
- Avoid duplicating test cases provided to you by the instructor. Only submit test cases provided by the instructor if you need to add comments (e.g., to explain failures), and if you do you must give attribution.
- Let us say that a test case 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 cases in the same file that should not depend on the results of the poisonous test case. For example, if your program crashes on a test case then the test case is poisonous. A test case is probably poisonous if handling it causes your program to corrupt its data structures. Your program can produce the wrong result for a test case without it necessarily being poisonous. The idea is that non-poisonous test cases 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 cases in a file named "`test-cases-part-PART.txt`", where PART is replaced by the part number. Your program must not crash on this input.
  - You must submit each of your poisonous test cases in its own separate file with the name "`test-cases-part-PART-poison-NUM.txt`" where NUM is a number (1, 2, 3, . . .).
- The student whose test cases crash the most other students' programs will get their "functionality" score raised (if possible). The same applies to the student whose test cases cause the most other students' programs to behave wrongly (either crash or produce incorrect output). Resource-limit-exceeded crashes due to absurdly big test cases do not count.
- Part or all of your test cases might 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. If you prefer other students to see your name, then express this wish in a comment in your main test case file.

5.2.5 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, . . .).

5.2.6 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 case 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.

5.3 Explanatory content

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

- What was required?
- 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 language.)
- 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.

5.4 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.

5.5 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 a human-readable plain text file.

5.6 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.

5.7 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 named `submit` that does the needed copying.

- 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.

5.8 Testing and assessment

- The assignment is **NOT** to handle just test cases 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.