CS358 Principles of Programming Languages (Spring 2025)
Interpreter Implementation Project Guidelines (v2)

Version history:
v1: 4/11/25 Initial release, pre-approved domains, Milestone 1 instructions
v2: 4/25/25 Added Milestone2 instructions
v3: 5/19/25 Updated Milestone2 due; Added Milestone3 instructions

The main implementation project for the class is to build an interpreter for a simple language. Ultimately, this will include the following components:

1 Introduction

- An abstract syntax tree (AST) representation for programs in the language.
- A parser, to go from concrete syntax to ASTs.
- An evaluator, which actually executes programs.

We will build this up in pieces, with different parts due at different milestones. We will also expand the language as we go, so that later milestones will include more language features.

Overall, the *tentative* schedule is like this:

- Milestone 1 (due April 25): AST and evaluator for pure expressions
- Milestone 2 (due May 16 (subject to change)): Parser for expressions
- Milestone 3 (due June 6 (subject to change)): AST, evaluator and parser for functions, statements and expanded features
- Final version (due Finals week): Corrections only

2 Core Language

Everyone will implement the same core language. Details of this will be defined as we go, but for Milestone 1, it consists of a pure expression language containing:

- Numeric literals and arithmetic operations ($+$, $-$, etc.)
- Boolean literals and boolean operations (and, or, etc.)
- Relational operators ($==$, $<$, etc.)
- `let`-bindings and variables

A "program" is just an expression and the effect of running the program is just to display the value of that expression.

All the pieces needed for this will be shown in class or developed in the practice exercises, so building the core langauge will be largely a matter of combining things in the right way. Initially, we will write programs directly as abstract syntax; parsing will come in the second milestone.

3 Domain-specific Extension

In addition to the core language, you will design and implement a set of domain-specific extensions for a domain of your choice. The result will be a *domain-specific language (DSL)* tailored to writing programs for a particular purpose. You may choose from the domains described below, or pick a different domain *with prior permission of the instructor*.

For purposes of this project, it is important that your DSL have a useful notion of domain-specific *values* and *literals* and a pure expression sublanguage that describes how to combine these values using domain-specific *operators*. There needs to be an easy way to represent the values and implement the operators within Python (possibly using existing packages in PyPI, the Python package repository), and a way to inspect values externally for debugging and demo purposes.

Section 4 describes how these requirements can be met for the domains in the pre-approved list; if you want to pick a different domain, you'll need to describe how it meets these requirements. Note that there are many interesting DSLs (e.g., markup languages, build control systems, configuration languages) that do *not* have an obvious notion of expressions, and hence are not well-suited for this project.

And your full DSL may ultimately include additional domain-specific features besides expressions, including imperative features; if so, these will be developed in the later milestones.

4 Pre-approved Domains

These are the domains you can choose to use with no questions asked. Again, if you want to propose something different, please do—but you'll need to get instructor permission.

For each domain, we describe the *minimum* features you need to implement in order to get full credit for Milestone 1, but you are welcome (and encouraged) to add more operators if you wish! We include some possible additions for each domain, just to get your creative juices flowing. You can also choose to come back and extend your expression language in later milestones.

For Milestone 1, we will be working with *abstract* syntax only, but in designing your set of operators you may want to think ahead to what the *concrete* syntax should look like, to be prepared when we reach Milestone 2.

Strings This DSL is for simple string manipulation.

Values: Strings of (unicode) characters.

Literals: Quoted strings, using any convenient convention of escape sequences for non-printing characters.

Operators: (i) concatenate two strings; (ii) replace first instance of a given substring with another given string.

Implementation: Just use Python strings and string operators. See `https://docs.python.org/3/library/stdtypes.html#string-methods` to get started.

Result: Just output display resulting string to the terminal.

Possible additions: Lots of other string operators like substring selection, searching, case modification, etc. are easily available as Python library calls.

Images This DSL is for composing images out of smaller ones.

Values: Raster images (rectanges of pixel data).

Literals: Names of files containing raster images in `.jpg` or `.png` format.

Operators: (i) combine two images horizontally (fails if they have different heights); (ii) rotate an image by 90 degrees clockwise.

Implementation: Use the Python `Pillow` package. See `https://pillow.readthedocs.io/en/stable/handbook/tutorial.html` to get started.

Result of evaluation: Output the resulting image in a file called `answer.png`; if possible, automatically open a viewer (such as `xv` on linux or Preview on MacOS) to display it.

Possible additions: Pillow supports dozens of image transformations.

Tunes This DSL is for composing simple major mode melodies algorithmically.

Values: Tunes, which are sequences of (pitch,duration) pairs. Pitches correspond to keys on the piano; durations are in seconds.

Literals: Tunes represented as an explicit sequence of (pitch name,duration) pairs, where pitch names C,D,E,F,G,A,B represent the keys of the major scale starting at middle C and durations are integers. The pseudo-pitch R can be used to represent a rest of the specified duration.

Operators: (i) concatenate two tunes so they are heard one after the other; (ii) transpose a tune up or down by a specified number of half-steps.

Implementation: Use the Python `midiutil` package. See `https://midiutil.readthedocs.io/en/1.2.1/` to get started.

Result of evaluation: Output the tune as a one-track midi file called `answer.midi`; if possible automatically open an app (such as GarageBand on MacOS or Windows Media Player) to play it.

Possible additions: accidentals, multiple simultaneous tunes (counterpoint!), chords(harmony!), dynamics, etc., etc.

Querying This DSL is for writing queries over a relational database, i.e. a very simplified form of SQL.

Values: Relational tables.

Literals: Names of `.csv` (comma-separated value) files containing tables. First row of csv file should contain column headings and remaining rows contain corresponding data.

Operators: (i) SELECT a set of named columns into a new table (error if any of the columns are not in the table); (ii) perform an (inner) JOIN operation on two tables by a specified column name (error if that is not a column name in both tables).

Implementation: Use the Python csv library to read and write files to a suitable Python internal data structure. See `https://docs.python.org/3/library/csv.html` to get started. The operators are easy to implement in Python.

Result of evaluation: Output the table value as a `.csv` file in the same format as the literals.

Possible additions: SELECT statement with WHERE clauses, other forms of JOIN, connect to an actual database baskend, etc.

Shell This DSL is for writing simple shell scripts to coordinate processing on unix platforms.

Values: Process descriptions, i.e. directives for running a pipeline of one or more unix processes, with possible IO redirections.

Literals: Strings representing unix program names, possibly with flags and arguments.

Operators: (i) connect two process descriptions with a pipe from one to the other; (ii) add a redirection for stdin, stdout, or stderr to a process description.

Implementation: Use the Python OS library to actually invoke the process description. See `https://docs.python.org/3/library/os.html` for details.

Result of evaluation: Can be printed as a string, but much more fun to actually `exec` it!

Possible additions: Additional operators to specify background execution, etc. See `www.ccs.neu.edu/home/shivers/papers/scsh.pdf` for one of the inspirations for this DSL idea (although the system described there is much more complex).

5 Milestone 1 Instructions

Here are the detailed requirements for the first Milestone, which is due at **10 PM**,**April 25.** This section is admittedly long, but should be mostly straightforward. If you have any questions, ask!

Domain Pick the domain you want to use for your project. You will be using the *same* domain for all the remaining milestones too. If this domain is not on the pre-approved list in the Section 4, you must obtain prior permission in writing from the instructor and/or TA. Obviously, it is to your advantage to get permission well before the due date.

Organization and Submission Create a directory `project/` containing a single file `interp.py`, which will contain all the code for this milestone. (Later milestones may involve adding additional files.)

Your `interp.py` file will contain AST node definitions for expressions, an `eval` method for interpreting expressions to values, and a `run` method for showing values to the user. The general form of this file should be similar to the `interp_arith_turtle.py` discussed in class on April 10 and posted in Canvas for week 2.

You *must* include a brief description of your domain-specific extension and how it is intended to be used. This should appear as a block comment near the bottom of your `interp.py` file.

In addition to this description, you *must* include a set of test expressions at the bottom of your `interp.py` file that show off the features of your domain-specific extension. Each test expression must be written using the (core + domain-specific) set of AST constructors, and embedded in a call to `run()`.

To submit, zip the contents of this directory into a file `project.zip` and upload this file to Canvas.

Language For this milestone, the language you interpret will just consist of expressions. The expression language must contain at least the following expression forms:

Core language:		
arithmetic	`Lit(int)`, `Add`, `Sub`, `Mul`, `Div`, `Neg`	as in `interp_arith.py`
boolean	`Lit(bool)`, `And`, `Or`, `Not`	as in Week 2 exercises
binding/variables	`Let`, `Name`	as in either of the above
equality comparison	`Eq`	see below
relational comparison	`Lt`	see below
conditional	`If`	see below; similar to Week 3 exercises
Domain-specific extension: ...	...	see below

Note the following points:

- There are three kinds of values: integers, booleans, and domain-specific. The language is dynamically typed. The `eval` method can return any of these kinds of value.
- The arithmetic operators take integer operands and yields an integer result. An exception should be raised if any of the operands is not an integer, or if division by zero is attempted.
- The boolean operators take boolean operands and yields a boolean result. `And` and `Or` should be *short-circuiting*, meaning that right operand is not evaluated if the boolean result is already determined by the left operand; Pythons own `and` and `or` operators are short-circuiting. An exception should be raised if any of the operands is not a boolean.
- The equality comparison `Eq(`l`,`r`)` evaluates l and r to values and yields `true` if these values are equal and `false` otherwise. The operand values can be of any type. For integers and booleans the definition of equality is obvious. For domain-specific values, the definition is up to you! If the types of l and r are different, their values should always be considered unequal.
- The relational comparison `Lt(`l`,`r`)` evaluates l and r to integer operands and yields `true` if $l < r$ and `false` otherwise. An exception should be raised if either of the operands is not an integer.
- The conditional operator `If(`b`,`t`,`e`)` takes one boolean operand b representing the value to test and t and e represent the "then" and "else" branches. If b evaluates to `true`, then t is evaluated and its value becomes the result of the `If`; otherwise, e is evaluated and its value becomes the result of the `If`. (Be careful not to evaluate *both* branches!) An exception should be raised if b is not a boolean. The types of t and e can be integers, booleans, or domain-specific values, and do not need to be the same.

Domain-Specific Extension For your choice of domain, you must define at least one new type of value, at least two literals of this type, and at least two operators that involve this type. For examples, see the descriptions for the pre-approved domains in Section 4. The literals and operators must be represented by new AST constructors, which can be given any names you like (as long as they do not conflict with the core operators described above). Your operator implementations should raise an exception if the types of their operands are wrong, or if the operands are otherwise illegal for some reason. In addition, you must define what equality comparison means for your new domain type.

You are welcome to include additional types, literals, and/or operators beyond the required minimum, but this is not necessary to obtain full points. It is better to get the required minimum working correctly than to only complete part of a more ambitious extension. You will also have the opportunity to add further expression features in later Milestones.

Running the interpreter Your `run` method should be modeled on the one in `interp_arith_turtle.py`: depending on the kind of value returned by `eval` it should show it in a suitable way. For some domains this means just displaying a string representation on the terminal; for others, it may mean invoking an auxilary application to play music or show a drawing, etc. Since these applications differ across OS platforms, you should *always* provide an easy way to capture the result value into a temporary file, so that we can inspect it and display it using an appropriate application for our own environment.

Extra Packages If implementing evaluation for your domain requires import of any packages beyond those in the Python standard library, you *must* indicate this in a comment at the *top* of your `interp.py` file. Be sure to specify the relevant package *version*.

You are strongly advised to install packages into a virtual environment (indeed, recent Python releases including the one installed on the CS Linux systems require this), using `pip`. For more information on how to do this, see `https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/`. For information on how to access your virtual environment from inside VS Code, see `https://code.visualstudio.com/docs/python/environments`.

Testing We will provide a test driver for the *core* features (only) of the project language in a file `test_phase1_core.py`. Similar to the practice exercises, you can run this just using:

```
python3 test_phase1_core.py
```

You should make sure that your `interp.py` can indeed by tested by this driver, e.g. that it has no syntax errors preventing execution, even if some of the tests fail.

Naturally, you should also test your domain-specific extension thoroughly, but there is no need to include all these tests in your submission—just includes those that demonstrate your domain features nicely.

Scoring Your delivered `interp.py` will be scored according to a point-based rubric. Roughly half the points will be given for the core, and the other half for your domain-specific extension. Unlike the weekly practice exercises, you will only get points for things that work, i.e., no points just for effort. Detailed rubrics can be found on Canvas later.

You are *required* to use the style we discussed in class and in weekly exercises to define the interpreter, instead of an object-oriented approach.

In particular, please make sure that your submission is free of syntax errors. You will lose significant points if we cannot even run and test your code because of such errors.

You will be given the opportunity to fix problems as part of your Milestone 2 submission, and by doing this you can potentially get up to 50% of the points you lose in Milestone 1.

6 Milestone 2 Instructions

Here are the detailed requirements for the second Milestone, due at **10 PM**,**May 16 (subject to change).**

Organization and Submission At this milestone, your project should consist of three files in your `project/` directory:

- a revised version of `interp.py`
- a grammar file `expr.lark`
- a file `parse_run.py` that combines and tests the first two

Most of the fresh work in this Milestone will be in the grammar and driver files. However, you will need to make some additions to `interp.py` to support functions in the core language, and you may also wish to correct any problems detected in your Milestone 1 submission. As before, this file should provide a `run` method that consumes an AST, evaluates it, and shows the result in a domain-appropriate way. See more instructions below on how to request credit for fixing problems from Milestone 1.

The `expr.lark` file should contain a Lark specification for the concrete grammar of the full language from Milestone 1, including your domain-specific extensions, and some small additions to support functions. The general form of this file should be similar to the `expr.lark` file discussed in class and posted to Canvas in week 5.

The `parse_run.py` file should contain the code to create a Lark grammar from the `expr.lark` file, and define a function `parse_and_run(s:str)` that transforms a parse tree into an AST expression tree and then invokes the interpreter in `interp.py` to run that tree. The general form of this file should be similar to the `parse_run_arith.py` file discussed in class and posted to Canvas in week 5, but should not invoke a `driver` loop at the bottom.

Instead, you *must* include a set of test expressions at the bottom of your `parse_run.py` file that show off the concrete syntax of your domain-specific extension. Each test expression must be written as a concrete string and embedded in a call to `parse_and_run()`.

To submit, zip the contents of this directory into a file `project.zip` and upload this file to Canvas. Please make sure that you zip the files in context of the `project` directory, not just the individual files, so that when unzipped a new copy of directory `project` is created. For example, the following command should do the trick when run from the directory *above* `project`:

```
zip project.zip project/parse_run.py  project/interp.py  project/expr.lark
```

Language For this Milestone, the core language should be extended to include (single-argument) functions with AST constructors `letfun` and `app` as defined in the `interp_fun.py` file discussed in class and posted to Canvas in week 6.

Concrete Grammar The concrete grammar for the core language should be the same as developed in class in weeks 5 and 6. Specifically, you should use the concrete syntax for arithmetic expressions from `expr.lark` in week 5, for boolean expressions from `parse_bool1.py` in week 5, and for function definition and application from `expr_fun.lark` in week 6. Use == and < for the `Eq` and `Lt` relational operators, and `if` e_1 `then` e_2 `else` e_3 for the AST form `If(`e_1`,`e_2`,`e_3`)`.

Your combined grammar should use the following precedence order (highest to lowest) for the core expression forms:

```
id parenthesized-expr application let-in-end letfun-in-end
- (unary)
* /
+ -
```

```
== <
!
&&
||
if-then-else
```

All binary operators are left-associative, except for == and < which are non-associative.

In addition, you must invent appropriate concrete formats for for your domain-specific literals and operators from Milestone 1, and write a combination of Lark terminal definitions and production rules to describe them. Then you must combine these with the core grammar to produce a single unified grammar for your extended language. For example, for the Turtle graphics domain described in Week 2, we might make these grammar additions

```
expr: ...
    | expr "++" expr -> append  # for Append constructor
    | expr "@" expr  -> repeat  # for Repeat constructor

atom: ...
    | "^" int  -> forward    # for Forward constructor
    | "<" int  -> left       # for Left constructor
```

It is up to you to decide on the precedence of your added domain-specific operators. Some choices will produce more pleasant results than others, but the only requirement for receiving full credit is that your resulting grammar is non-ambiguous.

Some notes on using Lark:

- The boolean literals `true` and `false` (and perhaps other domain-specific literals you introduce) have the same lexical form as ordinary `id` tokens. The grammar and Lark invocation in `parse_bool1.py` show one way to handle this problem using priorities on terminals, but this approach turns out not to scale well. A better approach is to parse `true`, `false`, etc. as ordinary `id`'s, and then detect them during the `Transform` process, converting them to `Lit` forms instead of `Id` forms at that point. Using this approach, an appropriate invocation of Lark looks like this:

  ```
  parser = Lark(Path('expr.lark').read_text(),start='expr',
                parser='earley',ambiguity='explicit')
  ```

- Once you have finished designing and testing your grammar, you can check that it is unambiguous by attempting to build an `lalr` parser rather than an `earley` parser. This will only succeed if your grammar is indeed in the class of LALR(1) grammars, which in particular means it is unambiguous. To do this, you must first install the `interegular` extension to the Lark package by doing:

  ```
  pip3 install lark[interegular]
  ```

  Then change your Lark invocation to look like this:

  ```
  parser = Lark(Path('expr.lark').read_text(),start='expr',
                parser='lalr',strict=True)
  ```

If this call succeeds, it means your grammar is OK. If it fails, you will get error messages about "shift/reduce" or "reduce/reduce" conflicts. We don't recommend trying to debug the grammar based on these errors, because that requires understanding how LR parsing works, which is beyond the scope of this course. But if you get errors, you can go back to using the Earley parser invocation and do more testing to see where the ambiguities may be coming from.

Scoring and Re-scoring Your delivered code and grammar will be scored according to a point-based rubric, similar to Milestone 1. As a reminder, you will only get points for things that work, i.e., no points just for effort.

If you lost points in Milestone 1, you can regain up to 50% by fixing them in this Milestone. To obtain these credits for any particular rubric item where you lost points, you must:

- Make appropriate changes in the `interp.py` file
- **Mark your changes** with a comment, specifying which rubric item you are addressing.
- Include the rubric item on a list of changes you include in a comment at the top of the `interp.py` fie.

You will *not* receive credit unless you tell us clearly which rubric items you (claim to) have fixed!

If your changes result in a higher point total for a rubric item, you will get a credit of 50% of the point difference. For example, suppose you orignally got 2/5 points for a particular rubric item. You fix the code, but not quite correctly, and now it is worth 4/5 points. The difference is 2 points, but you only get 50% of that, so the net effect is to improve your Milestone 1 score by 1 point.

You will get a similar chance to improve your Milestone 2 scores at Milestone 3, and so forth. But this is your *last chance* to improve your Milestone 1 scores!

7 Milestone 3 Instructions

Here are the detailed requirements for the third Milestone, due at **10 PM**,**Monday, March 10.**

Organization and Submission At this milestone, your project should again consist of (at least) three files in your `project/` directory:

- a revised version of `interp.py`
- a grammar file `expr.lark`
- a file `parse_run.py` that combines, tests, and demonstrates the first two

You will need to make changes and additions to all three files for this Milestone. You may also wish to correct any problems detected in your Milestone 2 submission; see more instructions below on how to request credit for this.

You must also include any auxiliary files (e.g. images) that are needed for your test programs to run.

To submit, zip the contents of this directory into a file `project.zip` and upload this file to Canvas. Please make sure that you zip the files in context of the `project` directory, not just the individual files, so that

when unzipped a new copy of directory `project` is created. For example, the following command should do the trick when run from the directory *above* `project`:

```
zip project.zip project/parse_run.py  project/interp.py  project/expr.lark
```

Core Language For this Milestone, the core language should be extended as follows:

- Make variables mutable, by binding them to `Location`s as in the `interp_imp.py` file discussed in class and posted to Canvas in Week 7. This will require changes in the existing implementations of `Name`, `Let`, `Letfun` and `App`. It is fine to store the `Closure` representing a function value in a `Location`, even though we won't allow these values to be mutated.
- Add an assignment operator `Assign` (concretely written x `:=` e), as in `interp_imp.py`. However, for this project assignment should be an expression, not a statement (there are still no statements). The result value of the assignment expression should be the value of e. Assignment should be valid on any `let`-bound variable in scope. Attempting to assign to an unbound variable or to a function name (bound by `letfun`) should raise an exception.
- Add an expression sequencing operator `Seq` (concretely written with a semicolon e_1 `;` e_2), as in the `interp_arr.py` discussed in class and posted to Canvas in Week 8.
- Add a new operation `Show` (concretely `show` e) that evaluates e to a value v and then shows v in a suitable way, just as your `run` function already does. (For example, a string can just be displayed on the terminal, but an image should be drawn in a new window, etc.) In effect, this allows you to write programs that "show" their results at multiple points during expression evaluates, rather than just once at the end.
- Add a new operation `Read` (concretely `read`) that prompts the user for an integer input and returns that integer. Raise an exception if what the user provides is not an integer.

The sequencing operation should have the lowest possible precedence, and should be right-associative (just to make it easier to define tests; the associativity doesn't really matter). The assignment and show operations should have the next lowest precedence, the same as `if-then-else`. The `read` operation should have highest precedence, the same as the existing atoms. (Note: it is easiest to parse `read` as an `ID` and special-case it during the transformation from parse tree to AST, as we have previously recommmended for `true` and `false`.)

Domain-specific Extensions For this Milestone, you must enhance your domain-specific extensions and produce an interesting demonstration of their capabilities in the form of one or more (concrete) test programs.

At a minimum, you **must** add *two* new domain-specific operators that produce and/or consume domain-specific values (e.g. strings, images, etc.). This will require additions to the interpreter, parser, and `ToExpr` transformer class. As before, you can decide on the precedence and associativity of your new domain-specific operators.

In most cases, all operands to the operators should be `Exprs` (as opposed to literals); your operator implementations should check that these expressions evaluate to values of the appropriate type and raise an exception if not.

Since the core language now contains imperative features (assignment, I/O, explicit sequencing), you may choose to make your new operators imperative as well. But pure functional operators are fine too. For example, if your domain is string manipulation, you might add an imperative `reverse` operator that mutates its argument, or a pure one that returns a fresh string. For example, if `s` is the string `"abc"`, the pure operator might look like this:

```
let t = reverse s in show s; show t end
```

and show `abc cba`, while the mutating operator might look like this:

```
let t = s in reverse s; show s; show t end
```

and show `"cba cba"`. But if you choose mutating operators, be careful that your underlying Python representation of domain-specific values is mutable! If not, you'll need to change your representation to one that is. Probably the simplest way to do this is to put a mutable "box" around each value. For example, Python strings are *not* mutable, so to support mutating operators, you could change the representation for *your* strings to something like a singleton (Python) list of (Python) strings (similar to the `Loc` type used to support mutable variables in `interp_imp.py`). This will have consequences for the meaning of assigment and equality comparison, as illustrated by the second code example above, and discussed in Week 8's classes.

You are encouraged to get more creative, by adding additional operators and perhaps additional kinds of domain-specific values. However, this is not required in order to get full credit.

Top-level Tests and Driver Organization Your `parse_run.py` file **must** include one more more concrete test expressions that demonstrate the usefulness of your domain-specific extensions. At least one of these tests **must** include some meaningful interaction with the outside world by use of `show` and `input` expressions. You should try to make these "demo" programs as informative and interesting as possible. Think up a problem you'd like to solve using your language and try it; you'll learn a lot about the pros and cons of your language design that way!

To ease testing and evaluation of your submission, your `parse_run.py` **must** define the following functions:

- `parse_and_run(s:str)` works as before.
- `just_parse(s:str)` just attempts to parse and generate the AST for for concrete expression `s`, and returns that AST or `None` if the parse fails. (This allows separate testing of the parser.)
- `driver()` works as before. But there should **not** be an invocation of `driver()` at the bottom of the file. We recommend that you include the multi-line input and editing support facilities for input introduced in `parse_run_arr.py` from Week 8.
- `main()` runs your demo test(s) (by invoking `parse_and_run` on them).

Again, nothing should run when your `parse_run.py` file is loaded, so that we can choose which of the top-level functions to invoke.

Don't forget to include any auxiliary files, e.g. images, that are needed to make your `main()` function correctly.

Scoring and Re-scoring Your delivered code and grammar will be scored according to a point-based rubric, similar to Milestones 1 and 2. As a reminder, you will only get points for things that work, i.e., no points just for effort.

If you lost points in Milestone 2, you can regain up to 50% by fixing them in this Milestone. As before, to obtain these credits for any particular rubric item where you lost points, you must:

- Make appropriate changes in the relevant file(s).
- **Mark your changes** with a comment, specifying which rubric item you are addressing.
- Include the rubric item on a list of changes you include in a comment at the top of the relevant file(s).

You will *not* receive credit unless you tell us clearly which rubric items you (claim to) have fixed!

As before, if your changes result in a higher point total for a rubric item, you will get a credit of 50% of the point difference. For example, suppose you orignally got 2/5 points for a particular rubric item. You fix the code, but not quite correctly, and now it is worth 4/5 points. The difference is 2 points, but you only get 50% of that, so the net effect is to improve your Milestone 2 score by 1 point.

You will get a similar chance to improve your Milestone 3 score at the final submission. But this is your *last chance* to improve your Milestone 2 score! And it is too late to improve your Milestone 1 score at this point.