

CS358 Principles of Programming Languages – Winter 2025

Interpreter Implementation Project Guidelines (v2)

Version history:

v1: 1/13/25 Initial release with pre-approved domains

v2: 1/18/25 Added Milestone1 instructions and section numbering

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.
- A static checker, that verifies one or more correctness properties of programs *without* evaluating them.

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 1/29): AST and evaluator for pure expressions
- Milestone 2 (due 2/19): Parser for expressions
- Milestone 3 (due 3/5): AST, evaluator and parser for functions, statements and expanded features
- Final version (due 3/17): More expanded features, and static checker for everything

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 language will be largely a matter of combining things in 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 (rectangles 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 backend, 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 **Noon, Wednesday, January 29**. 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 Jan. 15 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	<code>Lit (int), Add, Sub, Mul, Div, Neg</code>	as in <code>interp_arith.py</code>
boolean	<code>Lit (bool), And, Or, Not</code>	as in Week 2 exercises
binding/variables	<code>Let, Name</code>	as in either of the above
equality comparison	<code>Eq</code>	see below
relational comparison	<code>Lt</code>	see below
conditional	<code>If</code>	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; Python's 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 auxiliary 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 be 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.

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 Phase 2 submission, and by doing this you can potentially get up to 50% of the points you lose in Phase 1.