CS 343S: Domain Specific Language Design Studio **Spring 2025**

Internal Lab

Due: Thursday, May 1st, 11:59pm (`make submission.zip` and submit it via Gradescope)

Bugs: We make mistakes! If it looks like there might be a mistake in the statement of a problem, please ask a clarifying question on Ed.

In this assignment, you'll implement three tiny DSLs by customizing the semantics of Python **operators**, **functions**, and **blocks**; using **operator overloading**, **decorators**, and **context managers**. Then, you'll use all three techniques to implement a DSL for propositional formulas, via **deferred execution**. As a bonus problem, you can add GOTO to Python, by hacking the runtime.

Setup

1. Install **ffmpeg**. It is a library for audio/video manipulation and playback. On Arch Linux, Ubuntu, and macOS (homebrew) this is the `ffmpeg` package. You can test that the installation has succeeded by running `ffplay -version` and `ffmpeg -version` (the exact version is unimportant).

2. Install the Python `pycosat` package. Your OS package manager might have a package like `python-pycosat`. Or, you can do this with `pip`. You can test that installation succeeded by running `python3 -c 'import pycosat'`.

1 Sounds (operator overloading)

In this problem, you'll implement a library for combining sounds that overloads Python operators to achieve a clean syntax. We've provided a starter class `Sound` for you to complete in `sound.py`. When you're done, commented parts of `sound_ex.py` should work as expected.

To complete this problem, you'll use the `ffmpeg` binary, executed using Python's `subprocess` standard library module. We recommend reading the docstrings in the starter code and consulting `ffmpeg`'s man pages and online documentation. You may also find `NamedTemporaryFile` to be useful.

Written We've forced you to use the operators `|`, `&`, `*`, `@`, and `[]` to respectively represent sequencing, overlaying, repeating, modifying, and sampling sounds. What operators would you have chosen for these operations? Write your answers in `written.md`, along with a short rationale.

2 Timing (context managers)

Now you'll implement a context manager that prints the execution time of a block after that block finishes. Implement your context manager in `timeme.py`. When you're done, the example script `timeme_ex.py` should report that sleeping for 0.5 seconds and 0.1 seconds takes the expected amount of time.

3 Memoization (decorators)

Now, you'll implement a memoization decorator in `memoize.py` It considers the arguments to its function each time that function is called. If those arguments have not been used before, it saves them and the result in a cache. If those arguments have been used before, it looks up the result in the cache.

When you're done, the example script `memoize_ex.py` should report that `fib(36)` takes tens of microseconds to run and that `binary_tree_count(36)` takes hundreds of microseconds.

Note: you can't complete this problem until after `timeme.py`.

4 Propositional formulas (deferred execution)

Now, you'll tackle a more ambitious problem: a DSL for checking the satisfiability of propositional formulas. Ultimately, you need to get the example script `formula_ex.py` to execute with the expected results. Now, let's talk about the DSL you're implementing.

Here is an example program:

```python
from formula import sat

print(sat(lambda x: x & ~x))  # prints None (UNSAT)
print(sat(lambda x, y: x & y))  # prints {'x': True, 'y': True}
print(sat(lambda x, y, z: x & (y | z) ^ ((x & y) |(x & z))))  # prints None (UNSAT)
```

Formulas are written as Python functions. Propositional (Boolean) variables are arguments to the lambda, and the Boolean operators AND, OR, NOT, and XOR are respectively `&`, `|`, `~` , and `^` .

We've already given you classes `Term`, `Var`, `And`, `Nand`, `Or`, `Not`, and `Xor` that define an AST for formulas. Your tasks are as follows:

1. Implement operator overloading.
2. Implement conversion from a term to a logically equivalent term (i.e., a an equivalent term in the same variables) that uses only NANDs.
3. Implement conversion from a term to an equisatisfiable CNF formula (AND of ORs of possibly negated variables).
    - With this, you can give your formula to a SAT solver (they expect CNF). We've implemented the glue code for you in `solve`.
    - Hint: encode the whole formula, one NAND at a time. Introduce a new variable for the results of each NAND. The NAND $y = \texttt{NAND}(x_1, \ldots, x_N)$ can be represented as the following CNF:

$$
\begin{aligned}
&(\neg x_1 \vee \cdots \vee \neg x_N \vee \neg y) \\
\wedge\ &(y \vee x_1) \\
&\vdots \\
\wedge\ &(y \vee x_N)
\end{aligned}
$$

Here is a worked example of translating the following formula from NANDs to CNF: $\phi(x, y, z) = \texttt{NAND}(x, \texttt{NAND}(y, z))$. Crucially, our translation adds a new variable for the result of each NAND. We introduce n_1 for the result of the inner NAND and n_2 for the result of the outer NAND.

$$
\begin{array}{ll}
\quad (\neg y \vee \neg z \vee \neg n_1) & \textit{start the inner NAND} \\
\wedge\ (n_1 \vee y) & \\
\wedge\ (n_1 \vee z) & \textit{end the inner NAND} \\
\wedge\ (\neg x \vee \neg n_1 \vee \neg n_2) & \textit{start the outer NAND} \\
\wedge\ (n_2 \vee x) & \\
\wedge\ (n_2 \vee n_1) & \textit{end the outer NAND} \\
\wedge\ n_2 & \textit{assert that the formula is true}
\end{array}
$$

This CNF is *equisatisfiable* with ϕ. That means that some assignment to x, y, z satisfies ϕ if and only if there exists assignments to n_1, n_2 that also satisfies the CNF. To see an example of this equisatisfiability in action, consider

$$(x \mapsto \top, y \mapsto \top, z \mapsto \top)$$

Notice that this assignment satisfies ϕ because the inner NAND is $\bot$, to the outer one is $\top$. Also notice that by extending this assignment to

$$(x \mapsto \top, y \mapsto \top, z \mapsto \top, n_1 \mapsto \bot, n_2 \mapsto \top)$$

So, the forwards direction of the iff holds in this case. In fact, with this CNF, both directions of the iff hold in all cases.

4. Implement the `sat` function, which takes a Python function, constructs a formula based on that function, and solves the formula.

At each stage, you can test your progress using some of the examples in `formula_ex.py`.

5 Bonus: GOTO (interpreter hacks)

In this bonus problem (which is worth a lot of fun and just a few extra points), you implement go-to by hacking the Python runtime. Your task is to make the example program `bonus_goto_ex.py` executable, and get it to produce the expected result.

You may find the following facts useful:

- The function `sys.settrace` allows one to register a user-defined "tracing" function that then gets called before each line of code executes.
- The `f_lineno` member of a stack frame is writable.
- The function `tokenize.generate_tokens` tokenizes a file, giving access to the line number (and other metadata) for every token.

Written After implementing your solution, explain it in `written.md`.

6 Feedback

Written Answer the `Feedback` questions in `written.md`.

Submission

Build your submission zip with `make submission.zip` and submit it to Gradescope.