CS 335 Semester 2023–2024-II: Project Description

24th Jan 2024

Due Your submission is due by Mar 3 2024 11:59 PM IST.

General Policies

- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.
- Each group will get a total of FOUR FREE DAYS to help with your project submission deadlines. You can use them as and when you want throughout the project. For example, you can submit Milestone 1 one day and ten hours late without penalty, and you will have two free days remaining to potentially utilize for the rest of the semester. The granularity is a day.

Description

The goal of this project is to implement a compilation toolchain, where the input is in Python language and the output is x86_64 code.

Milestone 1

In this milestone, you have to construct a scanner and a parser for a **statically typed** subset of the Python language. The output of your compiler should be a graphical representation of the abstract syntax tree (AST) of the input program. The leaves of the AST should be labeled by the token names and the lexemes within parentheses, e.g., `ID` (`myVarName`). You should come up with your token classes.

The following is an outline of recommended steps that you can follow for your implementation.

1. Read the lexical and grammar specifications for the Python 3.12 language carefully to identify **necessary** rules.

   The exact Python version is not important for the project. We have listed 3.12 since it is the latest stable version of Python. However, 3.12 grammar uses PEG which may be comparatively more difficult to understand.[1][2] Feel free to refer to older versions of the Python grammar (e.g., 3.6) if they are easier to understand.

2. Use Python 3.12 grammar to implement the parser using the **LALR** Bison parser generator. Other tools like ANTLR are not allowed.

   Simplify the grammar by (i) removing productions that are not required for our project, and (ii) cleaning productions that introduce ambiguity.

3. Integrate the lexer and parser interfaces so that the (1) parser can drive the lexer, and (2) the lexer returns tokens to the parser.

[1] PEG Parsers
[2] PEP 617 New PEG parser for CPython

4. Add actions to the grammar script so that the output of the parser is a DOT script representing the AST (not the parse tree) of the input program. The DOT script, when processed by the Graphviz tool called `dot`, should produce a PDF file with the diagram of the parse tree.

5. Submit FIVE non-trivial (e.g., $\geq$ 40 lines without comments) Python programs that work with your implementation.

6. Handle errors in the input Python programs. Print suitable error messages for the errors encountered. Your compiler can terminate compilation and exit after the first error is encountered.

Features. We will assume the input Python programs will include type hints (PEP 484), i.e., the input program will be statically typed. All uses of variables for the first time will include the type. All functions will also include their prototype (types of the formal arguments and the return type).

You are expected to implement support for the following language features.

- Primitive data types (e.g., `int`, `float`, `str`, and `bool`)
- 1D list (ignore dictionaries, tuples, and sets)
- Basic operators:
  - Arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`
  - Relational operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
  - Logical operators: `and`, `or`, `not`
  - Bitwise operators: `&`, `|`, `^`, `~`, $\ll$, $\gg$
  - Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=`, `^=`, `«=`, `»=`
- Control flow via `if-elif-else`, `for`, `while`, `break` and `continue` (ignore `pass`, `do-while` and `switch`)
  - Support iterating over ranges specified using the `range()` function. Supporting iterating over lists is optional.
- Support for recursion
- Support the library function `print()` for only printing the primitive Python types, one at a time
- Support for classes and objects, including multilevel inheritance and constructors. Ignore multiple inheritance (i.e., a class can have only one parent class).
- Methods and method calls, including both static and non-static methods
- Static polymorphism via method overloading

The input programs will respect the following constraints.

- The `main()` function will always be defined
- Ignore format specifiers in `print()` variants
- There will be a maximum of one method call on each line. Ignore call sequences on Object receiver types. You do not need to support statements like `obj.m1().m2().m3()`.

- Dunders are limited to `__name__`, `__main__`, and `__init__`

Ignore the following language features

`import ...` statements, file handling, enums, exceptions, support for various operations related to strings (e.g., concatenation), F-strings, list comprehension, iterators, generator expressions, method objects, Lambda expressions, higher-order functions and decorators, implicit and explicit type casting, function aliases, dynamic polymorphism via method overriding, Unicode characters, soft keywords, `del` keyword, `yield` statement, walrus (:=) operator, `with` statements, multiple assignments, slicing, returning multiple values from a function, positional and keyword-only arguments, variable length arguments to functions via `*args` and `**kwargs`

You are free to implement them as optional features in your project. It should be straightforward to support "dynamic polymorphism via method overriding" given that you are supposed to implement calls to `__init__()` in an inheritance tree.

Graphviz and DOT. Graphviz is a free graph visualization software. There are two components of Graphviz you will have to use to visualize the AST: (1) the language DOT for describing the AST, and (2) a tool called `dot`. If `graph.dot` is a DOT script describing the AST, then the following command generates a PDF containing the visualization of the AST.

```
$ dot -Tpdf graph.dot -o graph.pdf
```

The `dot` tool takes care of the tree layout. You only need to specify the tree structure (i.e., nodes, labels, and edges) with the DOT language. You can read more about the DOT language from http://www.graphviz.org/documentation/.

Other details. Your implementation must accept input file(s) and other options as command line parameters. Support the options `-input`, `-output`, `-help`, and `-verbose` at a minimum. The purpose and usage of the parameters should be obvious.

It may help your design if you know what to expect from Milestones 2 and 3. In Milestone 2, you are expected to generate 3AC IR for the input source program and add runtime support for making function calls. You will also need to (i) implement support for the symbol table and (ii) perform semantic analysis to do limited error checking on types and function signatures. In Milestone 3, you are expected to generate the correct x86_64 assembly from the 3AC which can be run via GAS on Linux. We will publish detailed descriptions in a few more weeks.

Submission. Submission will be through BOTH Canvas and CSE Git.

Canvas Create a zip file named "`cs335-project-<groupid>.tar.gz`" and upload it to Canvas. Alternate extensions like `.tar.xz` and `.zip` are allowed. Only one group member should submit the project on behalf of her project mates. The compressed file should contain a folder `milestone1` with the following contents:

- All your source files must be in `milestone1/src` directory. You are free to choose your implementation language.

- Create a directory `milestone1/tests` for your five test cases. Name the test files as "`test<serialnumber>.py`".
- Use `Makefile` (or equivalent build tools like `ant`) for your implementation. You are free to use wrapper scripts to automate building *and* executing your compiler.
- Your submission must include a `PDF` file under the `milestone1/doc` directory. The `PDF` file should describe the tools that you used and should include compilation and execution instructions. Document all command line options.

  You SHOULD USE LaTeX typesetting system for generating the PDF file.

Git Create a tag called milestone1 by the deadline. This is the tag that will be checked out and evaluated by the TAs.

You can create and push tags using the following commands.

```
git tag milestone1 -a
git push origin milestone1
```

Evaluation.

- Your implementation should follow the prescribed steps so that the expected output format is respected.
- We will evaluate your implementations on a Unix-like system, for example, a recent Debian-based distribution.
- We will evaluate the implementations with our inputs and test cases, so remember to test thoroughly.
  - Input test cases will use two spaces for indentation and will have a final newline
  - All input test cases will have an `if __name__ == "__main__":` block
- Our evaluation will mostly focus on correctness, the compilation time is not important.
- The TAs will meet with each group for evaluation. Make sure that your implementation builds and runs correctly.