Middle East Technical University Department of Computer Engineering

CENG 444

Language Processors

Fall 2022-2023

Project 1 Submission Guidelines & Grading

Due date: December 4th, Sunday, 23:59

1 Introduction

In this phase, you will implement a parser for the Vox language according to the grammar and other semantic details laid out in the project repository. In order for us to be able to conduct black-box tests, your implementation should fulfill certain requirements specified in this text.

The grading scheme given beside the tasks are tentative, and are meant to guide you which parts you should focus on the most.

2 Files

Scripts and test cases are provided to you in the project repository to help with your implementation:

- `tester.py` to test your implementation with vox programs or token streams, along with 5 scanning and 5 parsing examples with correct outputs.
- `parser.py` to implement your parser with `sly.py`.
- `lexer.py` to implement your lexer with `sly.py`.
- `misc.py` to implement semantic error checking features.
- `ast_tools.py` for you to generate an AST and walk it with visitors.

3 Implementation

3.1 Scanning (10 pts)

Implement your lexer in `lexer.py`. Follow the directions within the comments. Your implementation should also fulfill the following:

- Your lexer class should keep track of line numbers in the `Lexer.lineno` attribute.
- If the lexer hits an erroneous character, this character should be emitted as a token of type `ERROR`. `ERROR` tokens are never longer than a single character.

3.2 Parsing (50 pts)

Implement your parser in `parser.py`. Your implementation should fulfill the following:

- The parser should only successfully parse the strings in the language specified in `grammar.txt` and no other (here, it is assumed that the parser does not do any error recovery).
- The parser should have at most one shift/reduce conflict, solved with preferring shift over reduce (*guess which one*).
- The parser should comply with the precedence rules hinted in the grammar. All types of rules with list-like items should evaluate from left to right (statement lists, vector elements, function call arguments, formal parameters etc.).
- It is impractical for parsers within compilers to fail immediately if the source code is not parsable, since the code might have more errors and it would be more informative if the user is notified of as many of these errors as possible. Due to this reason, some error recovery rules can be added to the grammar. Add error recovery rules such that the erroneous stream of tokens are discarded until we hit a `;` or }. If we hit EOF before these characters, the parser should have a fatal failure and return `None` (*it is easy to solve this issue as well, but very hard to specify without spoiling grammar implementation details, and that is your job*). These error rules should reduce to a `free-statement`.

Implement `process` and `generate_ast` in `misc.py`.

- `generate_ast` returns the AST tree specified in `ast_tools.py` (or `None` if error recovery fails). **Note that the AST is not a mirror of the parse tree.** Some parts of the parse tree are discarded in the AST for conciseness (the parentheses are not represented, as well as utility non-terminals such as `simpleStmt` or `function`). List type nodes in the tree should appear in the order they are seen in the source code, left to right.
- For more freedom in your implementation, you can output your custom intermediate representation or your custom AST in `process`, but you have to return the AST in `generate_ast`. You can simply return the AST that complies with submission rules in `process` as well, and let `generate_ast` be identity function (*that is what we did in our answer-key implementation*).

3.3 Semantic Checks

These kinds of errors can not be reported without context-sensitive analysis.

- To implement these checks easily with the AST of `ast_tools`, you can subclass the `ASTNodeVisitor` class and fill in its methods. An example implementation is made with `PrintVisitor`.
- Here are some resources for you to have an idea about semantic error checking part:
  - Compilers: Principles, Techniques and Tools, 2nd Ed, Ch. 2.7
  - Crafting Interpreters, Ch. 8
  - Modern Compiler Implementation in Java, 2nd Ed, Ch. 5

3.3.1 Multiple Variable Declarations (10 pts)

Implement `multiple_var_declarations` in `misc.py`.

3.3.2 Undeclared Use of Variables (20 pts)

Implement `undeclared_vars` in `misc.py`.

3.4 Test Cases (10 pts)

Write some test cases for your implementation.

- Write 5 test cases for your lexer, named `test{1..5}.txt`. These files do not need to resemble programs, think of them as streams of tokens. Obtain the outputs using:

```
./tester.py --save scan <filename>
```

- Write 5 test cases for your parser, named `test{1..5}.vox`. These files should be parsable (with/without error recovery). Obtain the outputs using:

```
./tester.py --save parse <filename>
./tester.py --save analyze <filename>
```

Include these test cases and their outputs in your submission.

3.5 Bonus: Error Hunt (2.5/5/10 pts)

It is quite difficult to point out an edge case for a good parser implementation. During grading, we will use the correct test case/output pairs you have sent along with our own test cases in the black-box tests. Strictly for "decent" submissions (meaning the particular submission seems to be complete in the big-picture):

- If one of your test cases causes an incorrect output in a decent submission, you receive 2.5 bonus points.
- If one of your test cases causes an incorrect output in 3 or less decent submissions, you receive 5 bonus points.
- If one of your test cases causes an incorrect output in a decent submission and it is the only incorrect output of this decent submission over all test cases, you receive 10 bonus points.

Test cases will be tried seperately for scanning, parsing, and semantic error checking. You can submit more test cases than required to increase your chances.

4 Submission

Submit your test cases, their outputs (without changing the name), `parser.py`, `lexer.py`, `misc.py` compressed in `eXXXXXXX.tar.gz` through ODTUCLASS.

5 Other Details

- Your code should be able to execute in a Python 3.8.10 virtual environment with only `sly.py` installed.
- Although your assignment will be checked in a black-box fashion, please comment parts of your code that you think is hard to understand.
- Please ask your questions regarding the assignment publicly in the google maillist for CENG444, unless your question reveals part of your solution. In that case, send a mail to `onem@ceng.metu.edu.tr`.
- **This is an individual assignment. Using any piece of code that is not your own is strictly forbidden and constitutes as cheating. This includes friends, previous homeworks, or the Internet. The violators will be punished according to the department regulations.**