CS4120/4121/5120/5121—Spring 2026
Programming Assignment 1
Implementing Lexical Analysis
Due: Wednesday, February 4, 11:59PM

This programming assignment requires you to implement a *lexer* (also called a *scanner* or a *tokenizer*) for the Eta programming language. As discussed in Lecture 2, a lexer provides a stream of *tokens* (also called *symbols* or *lexemes*) given a stream of characters.

0 Changes

- Update version control instructions.
- Update AI policy for the course.
- Changes to the submission instructions.

1 Instructions

1.1 Grading

Solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand and maximizes code sharing. A correct program compiles without errors or warnings, and behaves according to the requirements given here. A program with good style is clear, concise, and easy to read.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variable names and proper indentation. Keep your code within an 80-character width. If writing Java, most methods should be accompanied by Javadoc-compliant specifications, and class invariants should always be documented. Other comments may be included to explain nonobvious implementation details. Use similar best practices for other programming languages, but be sure to consult with the staff before choosing a language other than Java 11.

1.2 Partners

You will work in a group of 3–4 students for this assignment. Find your partners as soon as possible, and set up your group on CMSX so we know who has partners and who does not. Ed also has support for soliciting partners. If you are having trouble finding partners, ask the course staff, and we will try to find you a group in a fair way.

Remember that the course staff is happy to help with problems you run into. Read all Ed posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

1.3 Package names

Please ensure that all Java code you submit is contained within a package (or similar, for other languages) whose name contains the NetID of at least one of your group members. Subpackages under this package are allowed and strongly encouraged. They can be named however you would like.

1.4 Tips

This assignment is much smaller than future assignments: it is intended primarily as a warmup that gives your group the chance to practice working together. Later assignments will stress your ability to work effectively as a group, so now is a good time to set up the infrastructure and collaboration style that you will use for the rest of the semester. Some tips:

- Meet with your partners as early as possible to work out the design and to discuss the responsibilities for the assignment. Keep meeting and talking as the project progresses. Be prepared for your meetings. Be ready to present proposals to your partners for what to do, and to explain the work you have done. Good communication is essential.
- You should partition the assignment into parts that can be worked on largely separately. Avoid the temptation to do the assignment more "efficiently" by having a subset of the group do all the work. To succeed at the course project, your group needs to figure out how to work together effectively—and the sooner, the better.
- A good way to partition an assignment into parts that can be worked on separately is to agree as a group on, first, what the different modules will be, and further, exactly what their interfaces are, including detailed specifications. The individual modules can then be implemented independently with confidence that integrating them will be straightforward.

2 Design overview document

We expect your group to submit an overview document. The Overview Document Specification outlines our expectations. Writing a clear document with good use of language is important.

These are key topics to include in your design overview document:

- Have you thought about the key data structures in this assignment?
- Have you thought through the key algorithms and identified implementation challenges?
- Have you thought about your implementation strategy and division of responsibilities between the group members?
- Do you have a testing strategy that covers the possible inputs and the different kinds of functionality you are implementing?

3 Version control

Working with group members effectively is a key learning goal for this project. To facilitate this goal, you must use version control to manage your partnership. Large modern software is always managed with version control. While it may require some learning, using version control is a valuable skill to have. In the short term, you will reap the benefits as you proceed further into the project.

As always, making your code public would be a violation of academic integrity, so be sure to use a private repository. For this class, we would like you to use Cornell Github. You should be already added to an organization called cs4120-2026sp. Create a new `private` repository for your group **under this organization** (Check if your group mate has already created one. One repo per group.). You may choose any appropriate name for your repository, such as a customized group name. Make sure to add all–and only–your group members as collaborators. Additionally, set up .gitignore files appropriately to avoid committing large binaries, derived files, IDE files, and other unnecessary files.

The instructor and TAs have access to your repository and will look at your commit history during grading to ensure that everyone in the group contributed. If your commit history does not clearly reflect individual contributions, briefly mention it in your overview document. (For example, if you employ pair programming for elements of the assignment (not a bad idea!), you may wish to clarify this in your overview document, as only one member would appear on the commit history for that work.)

4 Lexer

We encourage you to use a lexer generator such as JFlex in your implementation, but it is not required. If you do use a lexer generator, you may wish to consider using the adapter pattern to aid you in your implementation. A example grammar file for JFlex, `example.flex`, is included in the release folder that you might find useful to peruse.

5 Command-line interface

A command-line interface is the primary channel for users to interact with your compiler. As your compiler matures, your command-line interface will support a growing number of possible options.

A general form for the command-line interface is as follows:

```
./etac [options] <source files>
```

For this assignment, at least the following three options must be supported:

- `--help`: Print a synopsis of options.
  A synopsis of options lists all possible options along with brief descriptions. No source files are required if this option is specified. Invoking `etac` without any source files should also print a synopsis. To see an example of a synopsis, run `javac` from the command line.

- `--lex`: Generate output from lexical analysis.

  For each source file named `filename.eta`, a diagnostic output file named `filename.lexed` is generated to provide the result of lexing the source file. Each line in the output file corresponds to each token in the source file in the following format:

  `<line>:<column> <token-type>`

  where `<line>` and `<column>` indicate the beginning position of the token, and `<token-type>` is one of the following:

  - `id <name>` for an identifier
  - `integer <value>` for an integer constant
  - `character <value>` for a character constant, where `value` excludes enclosing quotes
  - `string <value>` for a string constant, where `value` excludes enclosing quotes
  - `<symbol>` for a symbol such as parentheses, punctuation, and operators
  - `<keyword>` for a keyword, including names and values such as `int` and `true`

  Non-printable and special characters in character and string literal constants should be escaped in the output, as well as Unicode character escapes as described in the Eta Language Specification, but ordinary printable ASCII characters (e.g., "`d`") should not be. Comments and whitespace should not appear in the output.

  A lexical error should result in the following line in the output file:

  `<line>:<column> error:<description>`

  where `<description>` details the error. All valid tokens prior to the location of the error should be reported as above.

  Table 1 shows a few examples of expected results.

- `-D <path>`: Specify where to place generated diagnostic files.

  If given, the compiler should place generated diagnostic files (from the `--lex` option), in the directory relative to this path. The default is the current directory in which `etac` is run.

  For example, if this path is `p` and the file to be generated is `a/r/se.lexed`, the compiler should place this file at `p/a/r/se.lexed`.

To parse command-line arguments, it is common to use a library instead of implementing such functionality manually (although it's perfectly fine and not difficult to implement it manually). Below are some common libraries for various languages that you might find useful:

- **Java**: Commons CLI, Argparse4j, args4j
- **OCaml**: `Command` module in Jane Street Core (see the Real World Ocaml chapter)
- **Haskell**: optparse-applicative
- **Scala**: scopt, scallop

6 Test harness

The test harness is a framework for testing your implementation and for grading your submissions. Named `eth`, the test harness has been tested to work in a 64-bit linux virtual-machine environment. The VM distribution contains instructions for setting up the VM on your machine.

Content of input file	Content of output file

`use io`
 	
`main(args: int[][]) {`
`  print("Hello, Worl\x{64}!\n")`
`  c3po: int = 'x' + 47;`
`  r2d2: int = c3po   //  No Han Solo`
`}`
	`1:1 use`
	`1:5 id io`
	`3:1 id main`
	`3:5 (`
	`3:6 id args`
	`3:10 :`
	`3:12 int`
	`3:15 [`
	`3:16 ]`
	`3:17 [`
	`3:18 ]`
	`3:19 )`
	`3:21 {`
	`4:3 id print`
	`4:8 (`
	`4:9 string Hello, World!\n`
	`4:31 )`
	`5:3 id c3po`
	`5:7 :`
	`5:9 int`
	`5:13 =`
	`5:15 character x`
	`5:19 +`
	`5:21 integer 47`
	`5:23 ;`
	`6:3 id r2d2`
	`6:7 :`
	`6:9 int`
	`6:13 =`
	`6:15 id c3po`
	`7:1 }`

`x:bool = 4all`
`x = ''`
`this = does not matter`
	`1:1 id x`
	`1:2 :`
	`1:3 bool`
	`1:8 =`
	`1:10 integer 4`
	`1:11 id all`
	`2:1 id x`
	`2:3 =`
	`2:5 error:Invalid character constant`

Table 1: Examples of running `etac` with `--lex` option

The VM setup script will download and install the test harness, along with other necessary components, such as Java, JFlex, Maven, and ant, to build and run your compiler. The full list of currently available features can be found in `root-bootstrap.sh` in the VM distribution. Additional components may be requested on Ed.

`eth` is installed in the `eth` directory and can be invoked from the command line. A general form for the `eth` command-line interface is as follows:

```
eth [options] <test script>
```

Two options are of particular interest:

- `-compilerpath <path>`: Specify where to find the compiler
- `-testpath <path>`: Specify where to find the test files

For the full list of currently available options, invoke `eth`.

An `eth` test script specifies a number of test cases to run. The directory `eth/tests/pa1` contains a sample test script (`ethScript`), along with several test cases. `ethScript` also lists the syntax of an `eth` test script.

To request `eth` to build your compiler, a command `build` can begin the test script. Upon receiving a `build` command, `eth` will invoke command `etac-build` in the compiler directory. `etac-build` is responsible for building the compiler in its entirety. Currently, `eth` will execute all commands in a test script, even if the build fails. Therefore, if you already have your compiler built by some other means, later tests can still succeed even if your `etac-build` fails or is unavailable. For each test case, `eth` will invoke command `etac` in the compiler directory.

The result of running the test harness is summarized in the last line of the output, which looks like this:

```
ethScript: ? out of 16 tests succeeded.
```

The details for each test case can be found prior to the last line.

`eth` was used successfully in the last iteration of the course, but bugs are always possible. Please report errors, request additional features, or give feedback on Ed.

7 Submission

Before submitting, you should first prepare your repository. Make sure that your repository is complete and ready to be graded. In particular, it should include all of the following:

- All source code needed to compile and run your project, including any required third-party libraries.
- All test cases and any scripts used to run them.
- A design overview document, as described in §2, placed in the root of your repository.
- A `README` file explaining how to navigate, build, run, and test your code (this may overlap with the design overview).

- Removal of unnecessary files, such as large binaries, generated files, IDE metadata, or OS metadata.

Once your repository is ready, go to your repository on the Cornell GitHub website and create a new release (found in the right sidebar). Use the programming assignment title—e.g., `Programming Assignment 1: Radio Recommender System` as the title of your release, and set the tag to `submissions`. Publish your release and download the release ZIP file. Finally, submit the release ZIP file on CMSX before the deadline of Wednesday, February 4, 11:59PM. **All of these steps must be completed on time.**

Your repository should be self-contained, meaning that graders should be able to clone it and build your project without additional setup. Using Git submodules is allowed; if you do so, ensure that `git clone --recursive your_repo` retrieves everything needed to build and run your project.

For smaller libraries, it is often easy and effective to include the source code directly, but be sure to make clear what is library code, e.g. by package name. JAR files also work well, but sometimes do not include Javadoc and are less well integrated with your IDE. Your mileage may vary.

If you use a lexer generator, please include the lexer input file, e.g., `*.flex`. Your `etac-build` should use this file to generate source code, and you should not submit the corresponding generated source code file (e.g. `.java`).

Finally, remember that clear and well-written documentation—including commit messages, the design overview, and the `README`—does not just help your graders; they also help you and your partners.

8 AI Policy

You may use AI tools such as ChatGPT to assist with small, well-scoped pieces of code. However, you should not use AI tools to generate large portions of your code or core functionality. Using AI too much is a common pitfall and tends to backfire in this course.

In your design overview document, please write down where and how you used AI tools, if at all.

> **The Vibe Mirage.**
>
> Who needs to write code anymore—don't we just ask generative AI to do the work for us? This approach may sound like it will save time and effort, but it rarely does. AI-generated code often looks convincing until it is tested carefully and reveals subtle bugs that cost far more time than they save. Because each stage of the project builds on your existing code, your foundations must be solid. Relying on AI can undermine your understanding of your own code, and that understanding is critical for success.