CS4120/4121/5120/5121—Spring 2020
Programming Assignment 1
Implementing Lexical Analysis
Due: Monday, February 3, 11:59PM

This programming assignment requires you to implement a *lexer* (also called a *scanner* or a *tokenizer*) for the Xi 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

- None yet; watch this space.

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 CMS so we know who has partners and who does not. Piazza 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 Piazza 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 this is a good time to set up the infrastructure 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.
- One way to partition an assignment into parts that can be worked on separately is to agree on, first, what the different modules will be, and further, exactly what their interfaces are, including detailed specifications.

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, we would like you to use version control in managing your partnership. You may choose to use any system you like; common industry standards include Git, Subversion, and Mercurial.

As always, making your code public would be a violation of academic integrity, so be sure to use a private repository. Cornell Github is one option well suited for this class, since you are allowed unlimited private repositories for free.

You must submit a file `pa1.log` containing your group's commit history. This is not extra work, as version control systems already provide this functionality. 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 delve further into the project. In the long run, any large piece of modern software is always managed with version control.

Modifying the log file in any way will be considered a violation of academic integrity. If you feel there needs to be a significant clarification, instead briefly mention it in your overview document. (For example, if you employ pair programming for elements of the assignment, 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 this 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:

```
xic [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 `xic` 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.xi`, 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, 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 **??** 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 `xic` 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. Below are some common libraries for various languages that you might find useful:

- **Java**: Commons CLI, Argparse4j, args4j, many more are listed here
- **OCaml**: `Command` module in Jane Street Core (see Real World Ocaml chapter here)
- **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 `xth`, 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. (If this link is down, the VM has not yet been released. It is coming soon.)

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 Piazza.

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

```
xth [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 `xth`.

Content of input file	Content of output file

`use io`
 	
`main(args: int[][]) {`
`  print("Hello, Worl\x64!\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`
	`4:29 )`
	`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 `xic` with `--lex` option

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

To request `xth` to build your compiler, a command `build` can begin the test script. Upon receiving a `build` command, `xth` will invoke command `xic-build` in the compiler directory. `xic-build` is responsible for building the compiler in its entirety. Currently, `xth` 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 `xic-build` fails or is unavailable. For each test case, `xth` will invoke command `xic` in the compiler directory.

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

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

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

`xth` 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 Piazza.

7 Submission

You should submit these items on CMS:

- `overview.txt/pdf`: Your overview document for the assignment. This file should contain your names, your NetIDs, all known issues you have with your implementation, and the names of anyone you have discussed the homework with. It should also include descriptions of any extensions you implemented.
- A `zip` file containing these items:
  - *Source code*: You should include everything required to compile and run the project. Please ensure that the directory structure of your source files is maintained within the archive so that your code can be compiled upon extraction.
    If you use a lexer generator, please include the lexer input file, e.g., `*.flex`. Your `xic-build` should use this file to generate source code, and you should not submit the corresponding generated source code file (e.g. `.java`).
    Your build process must not download anything from the internet. If your code depends on any third-party libraries, they must be included in the submission.
    Do not submit compiled versions of your own code (submitting precompiled libraries is OK).
  - *Tests*: You should include all your test cases and test code that you used to test your program. Be sure to mention where these files are in your overview document.

  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.

Do not include any derived, IDE, or SCM-related files or directories such as `.class`, `.jar` `.classpath`, `.project`, `.git`, and `.gitignore`, unless they are precompiled versions of third party libraries.
It is strongly encouraged that you use the `zip` CLI tool on a *nix platform, such as the course VM. Do not use Archive Utility or Finder on macOS as they include extraneous dotfiles, and do not use a Windows tool which does not maintain the executable bit of your `xic` and `xic-build`.
It is suggested that you write a small (shell) script to pack your submission zip file, since you will be using it repeatedly throughout the course.

- `pa1.log`: A dump of your commit log from the version control system of your choice.