Programming Assignment 1 - Implementing Lexical Analysis

Collaborators

- Michael Maitland, mtm68
- Scott Bass, sb2383
- Michael Tobin, mat292

Running the Program

Main class: `mtm68.Main.java`

Run `xic-build` (this requires having Maven installed). Then `xic [options...] <source-files>` becomes available.

Summary

For this assignment, we found the design and implementation of our lexer to be rather painless for the most part. The challenges we did face mostly had to deal with project setup and also small technicalities (such as managing escaped chars). Currently, the main shortcoming of our design lies in its error handling. We believe that we could add more explicit error messages that would make debugging a good deal easier later in the future.

Specification

For the most part, we found that this assignment did not really require us to make too many key decisions. There were, however, a few that are worth discussing…

One decision we made was to create a lex token for each punctuation symbol individually (i.e. one for [,], {, }, etc…) rather than create a single Symbol lex token with an attribute containing its specific form. We chose this approach, despite it requiring more work at this stage, as we figure that later it will result in much cleaner code (less attribute accessing) and easier matching in the parsing stage. Bass's past experience with lexing and parsing helped inform this decision.

Another decision dealt with how non .xi or .ixi files are handled when given as arguments to a xic call. We decided that rather than erroring out, we would ignore such files and print a simple statement acknowledging that we ignored it to command line.

Design and Implementation

Architecture

I think it is important to note we decided to use Maven to organize our project and build process. We chose this method of organization as we all have at least some experience with using Maven in the past.

The key classes we created for this assignment are the following...

- mtm68.Main.java
  - This is the entry point to our compiler. Here, leveraging the args4j library, we have set up the command line parsing. Compiler command options are private class fields annotated with the args4j @Option. The source files included in a command are picked up in a private List field annotated by @Argument. These values are then filled out appropriately after calling the command line parser on args[]. Then, they can be used freely throughout the rest of the code.
    - We decided to use args4j as we found the use of annotations to look very clean and maintainable. As a library, it is much less verbose than others like Commons CLI which we figure will make it much easier to come back to later when we need to add more command line features.

  - After parsing options and arguments, the code acts on the values retrieved. The source files enter the central pipeline (which currently only includes lexing but eventually will include the rest of phases such as parsing). For each source file, a SourceFileLexer is instantiated. Then, a list of the tokens from that file are generated using SourceFileLexer.getTokens. If the user provided the –lex option, this will be persisted to a .lexed file (either to the current directory or to a user-provided path if the -D option is present).
- mtm68.lexer.SourceFileLexer.java
  - SourceFileLexer is the wrapper class for the auto-generated Lexer. We created this to provide a clean lexing class for the rest of the codebase to interact with. Upon instantiation on a filename, SourceFileLexer creates a FileReader and then a Lexer on that FileReader. The most important method available is getTokens. This method constructs a List by repeatedly calling Lexer.getNextToken until null is returned. The list is then returned and the Lexer is closed.
- mtm68.lexer.Lexer.java & mtm68.lexer.lexer.flex
  - Lexer is the autogenerated class created by JFlex using lexer.flex. In our lexer.flex, we have a TokenType enum which contains all the different possible TokenTypes. We also have the Token class which contains a TokenType type, Object attribute, int lineNum, and int column. This fields are all instantiated when a Token is constructed. Most of our lexing rules are quite straightforward, simply creating a

a token when we see a symbol. The only complexities lie in integer lexing and string lexing. Integers can be written in three different ways: a raw int, a character literal, or a hex literal. This required constructing rules to match all three. We decided to add a seperate lex state for string parsing which is entered when a " character is reached and only exited when another " is reached. We decided to make it seperate as all characters in a string should not be lexed as seperate tokens but rather as a part of the one string.

Code Design

- For this assignment, we did not need any complex algorithms (outside of the DFA generation automatically handled by JFlex for lexing)
- There also wasn't a real need for any interesting or complex data structure usage either. Currently our tokens are stored in an ArrayList, however, depending on how we utilize that list for parsing, it might be more beneficial to change it to a LinkedList in the future.
- At this stage, we didn't feel we had to make any substantial tradeoffs so far.

Programming

- Most of the challenges we faced in this assignment actually had to do with project setup (using Maven correctly and setting up our dependencies). Once that was sorted, designing and implementing our ideas were not met with any great challenges.
- The following is the team coding breakdown for this assignment…
  - **Tobin:** Args4j command line parsing (Main.java)
  - **Maitland:** SourceFileLexer
  - **Bass:** StringUtils
  - **Group (Bass in driving seat):** lexer.flex, SourceFileLexerTests, other misc. changes

Testing

In order to test our code, we utilized a suite of JUnit tests. These tests ensure that characters are lexed correctly and the output is formatted correctly. We are confident that we have comprehensive coverage of the input space. In order to accomplish this, we made sure that every single viable character is tested at least once to be certain it is lexed correctly. We then did edge case tests for interesting lex rules such as with integers, strings, and hex literals. We have also created tests for errors at the lexing level to make sure that they are formatted correctly as well.

We do, however, believe we can improve our workflow by creating tests earlier in our design process (i.e. test-driven development). We did not face too great of a consequence for this assignment as it was relatively simple in nature but, as the assignments grow in complexity, this will likely help us a lot.

Work plan

We began by splitting up introductory tasks (Tobin - CLI, Maitland - Script/Project structure setup, Bass - JFlex exploration). Then we regrouped for dev-testing and peer review to ensure our project foundation was sound. Following this, Maitland designed the SourceFileLexer which allowed us to group up and design the lexer spec as a team. Once the lexer spec was designed, we created code to properly output the tokens to a file when required as well as code to handle lexing errors. From there, we divided testing and documentation evenly amongst the group.

This plan allowed for us to incrementally progress our codebase while making sure we are all on the same page as developers and that our code was all on the same page with our frequent checkups. This minimized any possibility of having to completely change a section of our code and allowed us to make minor tweaks when needed instead.

Known Problems

We believe our error messages could be more precise and helpful. Right now, there are a few exceptions that would be simply caught by a general catch-all catch block which prints a rather generic message. When we begin to add other phases to our parser, it will likely save us a lot of pain to make those messages more explicit.

Comments

We each have invested around 10 hours a piece into this assignment. A lot of this time was spent designing and figuring out our project set up (how to properly set up xic-build and Maven). To that end, we all agree that it would have been really useful if there was maybe some release code that gave either an example xic-build file or project setup which we could have referenced. We had to create and reference a few Ed posts to really get a good grasp of what was required from us. Although we didn't use Gradle, there was a very comprehensive Gradle setup post on Ed that we believe would have been really helpful to have been released with the assignment to begin with.