COSC 455

Programming Languages: Design and Implementation

Fall 2025

**Lab Exercise #**[**3**](https://www.youtube.com/watch?v=aU4pyiB-kq0)

*Think left and think right and think low and think high.*

*Oh, the thinks you can think up if only you try.*

* Dr. Seuss

**Name:** Blessing Abumere **/ 10**

**Goals:** The intention of this lab exercise is to further familiarize you with Backus-Naur Form (BNF) and gain more experience in understanding, designing and implementing simple grammars using ANTLR (<http://www.antlr.org/>). In doing so, you should develop a better understanding of how program languages are designed, familiarity with parser generator tools and developing BNF/EBNF grammars.

**Environment:** Throughout this lab we will use the ANTLRWorks 1.5.2 tool that I have provided on Blackboard. This is an executable JAR file that you should be able to run directly from the browser; you may alternatively download it and double-click on the file to run. If this does not work on your setup, you may have to run it from a command line as follows:

- java -jar antlrworks.jar

ANTLR is a very powerful tool that can do many of the things we have discussed in lecture including, defining grammars in EBNF, producing parse and abstract syntax trees and generating the necessary source code for lexical and syntax analysis (Java, Python, Perl, C++, C#, Objective-C, Ruby, JavaScript, etc.). In this lab, we will only use a small part of the ANTLR tool.

**Slack Channel:** All/any questions, problems and/or announcements for this lab should be directed to the course’s #lab3 Slack channel.

**Submission:** All lab exercises should be submitted in a .doc/.docx or .odt file via Blackboard and have a naming convention of *FirstNameLastNameLab3*. For example, if your name is [Homer Simpson](https://www.youtube.com/watch?v=8-4P1WPE-Qg), you would submit a single file via Blackboard named *HomerSimponLab3.docx*. The file should contain this lab description with your answers to the questions. ***Homework submissions not following this convention*** [***may not be graded***](https://www.youtube.com/watch?v=cnaeIAEp2pU)***.***

**Deadline:** Submitted via Blackboard by 11:59pm on Friday, September 12, 2025

1. **ANTLR Warmup.** (2 points) Consider a language that allows for the set of all strings consisting of two uppercase letters followed zero or more additional characters, each of which can be either an uppercase letter or one of the digits 0 through 9. A BNF solution for this grammar could be:

<language> ::= <uppercase> <uppercase> <rest>

<rest> ::= <uppercase> <rest> | <number> <rest> | ε

<uppercase> ::= A | B | … | Z

<number> ::= 0 | 1 | … | 9

where ε represents the <empty> rule described in the previous [lecture](https://www.youtube.com/watch?v=ss2hULhXf04). For this question, we will use the existing BNF definition and translate it to an ANTLR-based grammar specification so that we can examine example parse trees and generate the parser for this language.

**Task 1.** Start ANTLR and create a new ANTLR 3 Grammar, name it Lab3Grammar and click on Ok. This will present a blank editor screen divided into 3 main parts: the upper right section is the editor where you define a grammar in EBNF, the upper left provides an outline view of lexical tokens and parser rules that you have defined, and the lower half provides a window for output. You should also notice the 4 tabs in the lower left corner: Syntax Diagram, Interpreter, Console and Debugger. We will use the first 3 in this lab.

**Task 2.** In any grammar, the first thing to do is determine what the legal tokens are so that we can define them in the grammar. To define lexical tokens in ANTLR, we must use all capital letters for the lexical definition. For example, using the language described above, we can define the legal tokens for upper-case letters in ANLTR with the following syntax:

UPPERCASE\_LETTER : 'A'..'Z';

This rule defines the legal lexical tokens for an upper-case letter to be all letters A thru Z. You should do the same for the legal number tokens in this language, naming the lexical token NUMBER. Save this grammar and fix any errors noted in the Console tab.

Once you have the 2 necessary lexical token rules, you should be able to view the generated syntax diagrams of each rule by highlighting a rule and selecting the Syntax Diagram tab in the lower left-hand corner. You should recall that the Syntax Diagram uses the same syntax as the textbook: square boxes for terminal symbols (i.e., tokens) and circular boxes for non-terminal symbols (i.e., parser rules). View the Syntax Diagram will be helpful while designing the later grammars so that you can visually see what your EBNF rules are describing.

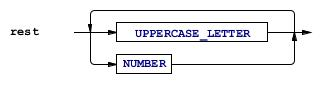
**Task 3.** Now that we have the lexical tokens defined for this grammar, we must define the syntax of the grammar through EBNF parser rules. To define parser rules in ANTLR, we must use lowercase letters to define the rules. In the language described above, any legal sentence must start with an upper-case letter and may or may not be followed by other characters. Based on this, we will define the start state, parser rule as follows:

language : UPPERCASE\_LETTER UPPERCASE\_LETTER rest?;

This defines a parser rule, named language, that requires two UPPERCASE\_LETTER tokens and may or may not be followed by whatever is defined in the non-terminal symbol rule named rest, which you will need to define. The ? syntax in this rule is ANTLR EBNF for saying that the rest syntax is optional in this rule. You can, eventually, confirm this by examining the Syntax Diagram as follows:



To complete this language, you must define a parser rule, rest, that finishes the definition of our language and should conform to the following syntax diagram:



Alternatively, ANTLR may display the syntax diagram as follows:



To do so, you will need to consider/utilize the following ANTLR syntax:

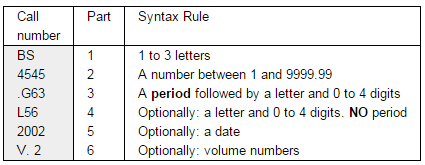
* \* denotes that a symbol is permitted 0 or more times. For example, (NUMBER)\* logically represents the rule “0 or more NUMBER tokens are legal”.
* + denotes that a symbol is permitted at least 1 or more times. For example, (NUMBER)+ logically represents the rule “there must be at least 1 NUMBER token, possible followed by more NUMBER tokens”.
* | denotes a logical or. For example, NUMBER (‘+’ | ‘-‘) NUMBER logically represents the rule “there must be a NUMBER token followed by either the plus or minus token followed by the NUMBER token.

Provide your full ANTLR EBNF grammar here.

**Task 4.** Once you have implemented the entire grammar, you should test it with some example inputs. First, we will use the input “ABC123DEF”. To do so, click on the Interpreter tab in the lower left hand corner, select “language” from the dropdown menu to note it as the start state, enter in the input string in the lower-left edit, and hit the “play” button located next to the dropdown menu where you selected “language”. This will generate a parse tree on the right-hand side to confirm that this input sentence is a correct, parsable sentence in our defined grammar.

Provide a screen shot of your parse tree. To do so, right click on the parse tree, select Export as a Bitmap image, save it and copy it here. Repeat this process for the input “123ABC”. Provide a screen shot of your parse tree.

1. **Library of Congress Classification System (LCCS).** (2 points) Consider the following language description, defined by the Library of Congress (LoC), to organize book collections topically:



*The main sub-units of a call number are as follows:*

***Topic: Parts 1 and 2****. The initial letters designate a broad topic. The numbers progressively narrow the topic. This is explained in more detail below. For simplicity, assume Part 2 ranges from 0 to 9999 (no decimal portion).*

***Cutter: Parts 3 and 4****. The topical portion of the call number is followed by a cutter number (named after Charles Cutter). The cutter number usually encodes the author (technically, the main entry.) So within a given topic books are usually sorted by author. Cutter numbers can also be used to further subdivide a topic, and one call number can have two cutters (usually topical use followed by main entry use). This is not common.*

***Other parts****. The call number may conclude with a publication year or volume number. For volume number, it must be in the format of a ‘v’ (lower case) followed by a period, followed by a one or two digit number.*

According to the LCC,S each of the six parts should have a line break; however, for this example, you can assume there is a space between each part. For example, the following would be legal call numbers using this classification scheme:

[PR 6051 .D3352 H5 1989](http://southcat2.usouthal.edu/vwebv/holdingsInfo?bibId=238201)

PS 3551 .S5 I2 2008

Likewise, the following would be illegal call numbers using this classification scheme:

12 JOSH .D24 K4 2016

JD 1234 A32 X7 1979

**Task 1.** Define the legal lexical tokens in the grammar for this language.

**Task 2.** Develop the parser rules for this grammar. As you develop/troubleshoot your grammar, be sure to utilize the Syntax Diagram to visualize the rules you are defining.

**Task 3.** Provide your full ANTLR EBNF grammar here and a screenshot for the parse trees generated by the example inputs provided previously.

1. **A Signed Even Number Grammar.** (2 points) Consider the following language description of a signed even number: A signed even number optionally has a “+” or “-“ to denote its sign followed by one more number of which the last number *must* be even.

**Task 1.** Define the legal lexical tokens in the grammar for this language. You will likely want to define separate lexical rules for the numbers 0-9 as well as the even numbers.

**Task 2.** Develop the parser rules for this grammar. As you develop/troubleshoot your grammar, be

sure to utilize the Syntax Diagram to visualize the rules you are defining.

**Task 3.** Provide your full ANTLR EBNF grammar here and a screenshot for the parse trees generated by the following inputs: [42](https://www.youtube.com/watch?v=aboZctrHfK8), 567 and +890.

1. **A Simple HTML Grammar.** (2 points) Let us consider a small subset of HTML with the following description:

* All legal HTML documents must start with the <html> tag and end with the </html> tag
* Within an HTML document, the following tags are legal:
  + The paragraph tag, starting with <p> and ending with </p> can have *text* in between as well as a bold tag
  + The bold tag (<b>…</b>) can only have *text* within its tags
  + Anywhere within the <html> tags can have *text*
  + *Text* is defined as any letter (upper or lower-case) and any number

Thus, a legal example of our html code would be:

<html>

<p> This is a <b>paragraph</b></p>

This is normal text

</html>

For this simple HTML language, construct the ANTLR EBNF grammar. To do so, you may want to consider lexical tokens similar to the following:

HEAD\_BEGIN : '<html>' | '<HTML>';

HEAD\_END : '</html>' | '</HTML>';

You should also define text exactly as follows:

TEXT : ('A'..'Z' | 'a'..'z' | '0'..'9' | '\t' | ' ' | '\r' | '\n'| '\u000C')\*;

This defines text to be any letter or number as well as accommodating for any tab, newline or space.

Develop the ANTLR EBNF grammar for this language and provide your full ANTLR EBNF grammar here. Additionally, provide a screenshot for the parse trees generated by the following 2 inputs:

#1:

<html>

<p> ANTLR is so <b> fun </b> </p>

COSC455 is fun too

</html>

#2:

<html>

This </p> does not follow the grammar <p>

</html>

1. **Simple Rust Implementation of Question 1 Syntax Checking. (2 points)** Using the syntax description from Question 1, complete the small Rust program, below, to programmatically check the syntax for the grammar described in Question 1. That is, using the provided code, in VS Code or Rust Playground for simplicity (see <https://play.rust-lang.org/>), complete the 3-5 lines of code in the q1\_parser function needed. This function should process the string parameter 1 character at a time, using the provided utility functions, to check adherence to the grammar’s syntax and return true if so, and false otherwise. I have provided 2 test cases, along with their assertions (read about Rust assert\_eq [here](https://doc.rust-lang.org/std/macro.assert_eq.html)). A successful execution should produce no output; a failing execution will produce failing test results to the console.

While AI/LLM (i.e., [*vibe coding*](https://en.wikipedia.org/wiki/Vibe_coding)) usage is allowed for this, the implementation for this is well within your abilities if you have completed Lab 2. Further, despite any vibe coding done, you are responsible for fully understanding your implementation, the implementation should stay within the Rust features we have seen in class/lab, and you should be able to answer similar questions, on paper, in an exam setting.

Provide your full Rust code here and a few screenshots showing the execution/output console.

Rust starter code:

fn main() {

let test1 = "ABC123DEF".to\_string();

let test2 = "123ABC".to\_string();

// Assertions for the test cases

assert\_eq!(q1\_parser(test1), true);

assert\_eq!(q1\_parser(test2), false);

}

// Utility method that takes a character and returns true if digit

// or upper case letter.

fn is\_uppercase\_or\_digit(c: char) -> bool {

is\_uppercase\_letter(c) || (c >= '0' && c <= '9')

}

// Utility method that takes a character and returns true if it is

// an upper case letter.

fn is\_uppercase\_letter(c: char) -> bool {

c >= 'A' && c <= 'Z'

}

fn q1\_parser(text: String) -> bool {

// make String into char vector

let characters\_array: Vec<char> = text.chars().collect();

// for each character in the character vector do...

// where i starts at 1 and increments for each iteration

for (i, character) in characters\_array.iter().enumerate() {

// YOUR CODE GOES HERE

}

}

**Note:** There are 3-5 [Easter eggs](https://en.wikipedia.org/wiki/Easter_egg_(media)) (this isn’t one of them) in this lab. If you find one, let me know via direct message on Slack.