COSC 455

Programming Languages: Design and Implementation

Fall 2025

**Lab Assignment #5**

*I like nonsense, it wakes up the brain cells. Fantasy is a necessary ingredient in living.*

* Dr. Seuss

**Name: Blessing Abumere / 10**

**Goals:** The intention of this lab exercise is to further familiarize you with [Backus-Naur Form](https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form) (BNF) and gain experience in understanding, designing and implementing simple parsers (i.e., syntax analyzers), as discussed in the previous [lecture](https://www.youtube.com/watch?v=ss2hULhXf04). In doing so, you should develop a better understanding of how parsers are designed and developed, as well as further familiarization with [VS Code](https://code.visualstudio.com/), [Rust](https://www.rust-lang.org/), [Slack](https://www.youtube.com/watch?v=5iJmFcMEuMg) and [GitHub](https://www.youtube.com/watch?v=0fKg7e37bQE). For help setting up the VS Code and Rust environment, see the Lab 2 description.

**Environment:** The VS Code IDE (or Rust Playground) using Rust must be used in this lab along with Slack to communicate and submit portions of the lab. No other programming language or submission will be allowed.

**Submission:** All lab tasks should be submitted via this document to Blackboard. ***Lab submissions not following this convention may not be graded.***

**Deadline:** Submitted via Blackboard by 11:59pm on Sunday, September 28, 2025

1. **A Simple Scanner and Parser.** Recall what the lexical and syntax analyzer does (review your [notes](https://i.imgflip.com/okxzx.jpg), if needed!). In a recursive-descent parser (i.e., a syntax analyzer), each non-terminal symbol has its own method in which the grammar rules are enforced.

In recursive-decent parsers, the algorithm/method of a non-terminal symbol with a single right-hand side rule is simple and proceeds as follows:

*For each terminal symbol on the right-hand side, that terminal symbol is compared with current token. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get the next input token. For each non-terminal, the syntax analyzer for that non-terminal method is called.*

For a non-terminal symbol whose rule is more than one right-hand side, the rule’s algorithm/method proceeds as follows:

*Each right-hand side is examined to determine the set of terminal symbols that can appear at the beginning of the sentence it can generate. By matching these sets against the next token of input, the syntax analyzer can select the correct path.*

Recall the simple [lolspeak](http://speaklolcat.com/) grammar from the previous lectures/labs:

<lolspeak> ::= <noun\_phrase> <verb> <noun\_phrase>

<noun\_phrase> ::= <article> <noun>

<verb> ::= lovez | hatez | ates

<noun> ::= dawg | kat | rat

<article> ::= a | teh

We will use and build upon this grammar in the simple compiler (i.e., recursive descent parser) we will implement in this lab.

**Task 1.** (1 point) I have provided a small compiler (main.rs), written Rust, for the simple [lolspeak](http://speaklolcat.com/) grammar. In the Rust Playground or VS Code, you should import this code into your IDE and examine it. You should notice that I have provided structures and implementations of a basic Compiler, a simplified Lexical Analyzer and partial implementation of a Syntax Analyzer in the main.rs file. Note, the Lexical Analyzer implementation provided cheats a bit and does not do character-by-character analyze, as usual, and simply tokenizes the input on spaces – this is simply for demonstration purposes.

In examining the code, **provide/annotate the source code with additional comments for documentation** and to demonstrate your understanding of Rust and the implementation. **Be sure to include your name in the code comments at the top of the source code.**

**Task 2.** (3 points) Using the algorithm/implementation description for recursive descent parsers provided in the lab description, implement the Syntax Analyzer methods for parsing the lolspeak grammar.

To demonstrate its functionality, **provide a few test cases and screenshots of their executions in this document. Be sure to include test cases that are syntactically legal and illegal.**

**Task 3.** (2 points) Next, we will extend the simple lolspeak grammar to include [adjectives](https://en.wikipedia.org/wiki/Adjective) using the following definition for an *adjective*:

* An adjective can be the word *fat*, *hungry, happy* or *mean*

With this addition, the following is the refined definition of a *noun phrase* in our lolspeak grammar:

* A noun phrase is an article followed by an adjective followed by a noun

For example, the following are legal sentences in our extended lolspeak grammar:

* teh hungry dawg ates a mean kat
* teh happy kat secretly hatez teh fat rat

Using these new, extended definitions, update the BNF rules and **provide the new BNF grammar in this document**.

grammar lolspeak;

// lexical definitions

ARTICLE : 'a' | 'teh' ;

NOUN : 'kat' | 'dawg' | 'cheezburgr' ;

VERB : 'ates' | 'lovez' | 'hatez' ;

ADJECTIVE: 'fat' | 'hungry' | 'happy' | 'mean' ;

// syntax definitions

sentence : noun\_phrase VERB noun\_phrase;

noun\_phrase : ARTICLE ADJECTIVE NOUN;

With your new BNF grammar production rules for our extended lolspeak grammar and the above italicized description of how to implement the methods of a syntax analyzer for a production rule, implement your developed algorithms for our extended lolspeak grammar. To do so, you will need to modify/add to the LexicalAnalyzer and SyntaxAnalyzer struct/impl Rust sections.

Once you have developed, executed and tested your extended compiler, **provide a few test cases and screenshots of their executions in this document. Be sure to include test cases that are syntactically legal and illegal.**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Task 4.** (2 points) Finally, we will extend the simple lolspeak grammar to include adverbs using the following definition for an *adverb*:

* An adverb can be the word *accidently*, *quickly* or *secretly*.

With this addition, the following is the refined definition of a *sentence* in our lolspeak grammar:

* A sentence is a noun phrase, optionally followed by an adverb, followed by a verb, followed by another noun phrase.

For example, the following are legal sentences in our extended lolspeak grammar:

* teh happy dawg quickly ates a fat kat
* teh mean kat secretly hatez teh happy rat
* teh hungry rat hatez teh mean dawg

Using these new, extended definitions, update the BNF rules and **provide the new BNF grammar in this document**.

grammar lolspeak;

// lexical definitions

ARTICLE : 'a' | 'teh' ;

NOUN : 'kat' | 'dawg' | 'cheezburgr' ;

VERB : 'ates' | 'lovez' | 'hatez' ;

ADJECTIVE: 'fat' | 'hungry' | 'happy' | 'mean' ;

ADVERB : 'accidently' | 'quickly' | 'secretly' ;

// syntax definitions

sentence : noun\_phrase ADVERB? VERB noun\_phrase;

noun\_phrase : ARTICLE ADJECTIVE NOUN;

With your new BNF grammar production rules for our extended lolspeak grammar and the above italicized description of how to implement the methods of a syntax analyzer for a production rule, implement your developed algorithms for our extended lolspeak grammar. To do so, you will need to modify/add to the LexicalAnalyzer and SyntaxAnalyzer struct/impl Rust sections.

Once you have developed, executed and tested your extended compiler, **provide a few test cases and screenshots of their executions in this document. Be sure to include test cases that are syntactically legal and illegal.**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Once you have developed, executed and tested your extended compiler, **copy/paste your code to the end of this document for submission.**

use std::vec;

pub struct Compiler {

    pub current\_token: String,

}

impl Compiler {

    pub fn new() -> Self {

        Self {

            current\_token: String::new(),

        }

    }

}

pub struct LexicalAnalyzer {

    tokens: Vec<String>,

    pub articles: Vec<String>,

    pub adverbs: Vec<String>,

    pub adjectives: Vec<String>,

    pub verbs: Vec<String>,

    pub nouns: Vec<String>,

}

impl LexicalAnalyzer {

    pub fn new(input: &str) -> Self {

        //puts a bunch of provided words into words so we can use them

        let mut tokens: Vec<String> = input

            .split(' ')

            .filter(|s| !s.is\_empty())

            .map(|s| s.to\_string())

            .collect();

        tokens.reverse();

        Self {

            tokens,

            adverbs: vec!["accidently".into(), "quickly".into(), "secretly".into()],

            articles: vec!["a".into(), "teh".into()],

            adjectives: vec!["fat".into(), "hungry".into(), "happy".into(), "mean".into()],

            verbs: vec!["lovez".into(), "hatez".into(), "ates".into()],

            nouns: vec!["dawg".into(), "kat".into(), "rat".into()],

        }

    }

    /\*\* loops if something is a valid token

     \* returns true if its valid

     \* false otherwise

     \*/

    pub fn lookup(&mut self, word: &str) -> bool {

        self.articles.iter().any(|a| a == word)

            || self.adverbs.iter().any(|a| a == word)

            || self.adjectives.iter().any(|j| j == word)

            || self.nouns.iter().any(|n| n == word)

            || self.verbs.iter().any(|v| v == word)

    }

    pub fn start(&mut self) -> String {

        let candidate\_token = self.tokens.pop().unwrap\_or\_default();

        if self.lookup(&candidate\_token) {

            candidate\_token

        } else if !candidate\_token.is\_empty() {

            eprintln!(

                "A lexical error was encountered. '{}' is not a recognized token.",

                candidate\_token

            );

            std::process::exit(1);

        } else {

            eprintln!("A user error was encountered. The provided sentence is empty.");

            std::process::exit(1);

        }

    }

    pub fn next(&mut self) -> String {

        let candidate\_token = self.tokens.pop().unwrap\_or\_default();

        if self.lookup(&candidate\_token) {

            candidate\_token

        } else if self.tokens.is\_empty() {

            "".to\_string()

        } else {

            eprintln!(

                "A lexical error was encountered. '{}' is not a recognized token.",

                &candidate\_token

            );

            std::process::exit(1);

        }

    }

    pub fn is\_a\_article(&self, word: &str) -> bool {

        self.articles.iter().any(|a| a == word)

    }

    pub fn is\_a\_noun(&self, word: &str) -> bool {

        self.nouns.iter().any(|n| n == word)

    }

    pub fn is\_a\_verb(&self, word: &str) -> bool {

        self.verbs.iter().any(|v| v == word)

    }

    pub fn is\_a\_adjective(&self, word: &str) -> bool {

        self.adjectives.iter().any(|v| v == word)

    }

    pub fn is\_a\_adverb(&self, word: &str) -> bool {

        self.adverbs.iter().any(|adv| adv == word)

    }

}

pub struct SyntaxAnalyzer<'a> {

    //borrowed references to lexer and compiler

    lexer: &'a mut LexicalAnalyzer,

    compiler: &'a mut Compiler,

}

impl<'a> SyntaxAnalyzer<'a> {

    pub fn new(lexer: &'a mut LexicalAnalyzer, compiler: &'a mut Compiler) -> Self {

        Self { lexer, compiler }

    }

    pub fn next\_token(&mut self) {

        let tok = self.lexer.next();

        self.compiler.current\_token = tok;

    }

    pub fn lolspeak(&mut self) {

        self.noun\_phrase();

        self.adverb();

        self.verb();

        self.noun\_phrase();

    }

    /\* After task 3 a noun phrase is an article followed by an adjective followed by a noun \*/

    pub fn noun\_phrase(&mut self) {

        self.article();

        self.adjective();

        self.noun();

    }

    pub fn verb(&mut self) {

        if self.lexer.is\_a\_verb(&self.compiler.current\_token) {

            self.next\_token();

        } else {

            eprintln!(

                "A syntax error was encountered. '{}' was found when a verb was expected",

                self.compiler.current\_token

            );

            std::process::exit(1);

        }

    }

    pub fn noun(&mut self) {

        if self.lexer.is\_a\_noun(&self.compiler.current\_token) {

            self.next\_token();

        } else {

            eprintln!(

                "A syntax error was encounted. '{}' was found when a noun was expected",

                &self.compiler.current\_token

            );

            std::process::exit(1);

        }

    }

    pub fn article(&mut self) {

        // if current token is an article

        if self.lexer.is\_a\_article(&self.compiler.current\_token) {

            self.next\_token(); //asks lexer to get next token

        } else {

            eprintln!(

                "A Syntax error was encountered. '{}' was found when an article was expected",

                self.compiler.current\_token

            );

            std::process::exit(1);

        }

    }

    //task 3

    pub fn adjective(&mut self) {

        if self.lexer.is\_a\_adjective(&self.compiler.current\_token) {

            self.next\_token();

        } else {

            eprintln!(

                "A Syntax error was encountered. '{}' was found when an adjective was expected",

                &self.compiler.current\_token

            );

            std::process::exit(1);

        }

    }

    //task 4

    /\*\* adverb is optional so the process doesnt exit if it doesnt exist \*/

    pub fn adverb(&mut self) {

        if self.lexer.is\_a\_adverb(&self.compiler.current\_token) {

            self.next\_token();

        }

    }

}

fn main() {

    //let test1 = "a kat lovez teh dawg";

    let sentence = "teh hungry rat hatez teh mean dawg"; //"teh mean kat secretly hatez teh happy rat"; //"a kat teh dog";

    //create instances of lexical analyzer

    let mut compiler = Compiler::new();

    let mut lexer = LexicalAnalyzer::new(sentence);

    compiler.current\_token = lexer.start(); //where the work starts for lab 5.

    let mut parser = SyntaxAnalyzer::new(&mut lexer, &mut compiler);

    parser.lolspeak();

    if !lexer.tokens.is\_empty() || !compiler.current\_token.is\_empty() {

        eprintln!("A syntax error was encountered. Additional tokens found after the sentence.");

        std::process::exit(1);

    }

    println!("The sentence '{}' follows the lolspeak grammar!", sentence);

}

1. **More Rust.** (2 points) As mentioned in class and in prior labs, this semester will utilize the Rust programming language for a significant project implementing some aspects of a compiler/parser and will be used throughout to demonstrate various programming language aspects. While I will not directly teach Rust, we will (yes, I am suffering along with you ☺) utilize the excellent Rust Essential Training LinkedIn Learning course, found at <https://www.linkedin.com/learning/rust-essential-training>. Note that LinkedIn Learning is completely free to all TU students (you must sign in using your TU email address). This is course is 6+ hours, but much of it assumes very little prior programming experience and I have found that watching/following along at 1.25-1.5x speed works for me. We will also tackle this in smaller chunks over the next few weeks to not overwhelm.

From this lab, I included some new, funky Rust syntax (e.g., impl<'a> SyntaxAnalyzer<'a> and/or you maybe encountered compiler errors related to *borrowing* in your simple compiler solution from Question 1 that you didn’t, yet, understand because these features have not been covered in the LinkedIn Learning modules we’ve gone through. Now that you’ve completed the simple compiler from Question 1, let’s fill in the gap of this Rust understanding. To do so, please complete video Chapters 6, 7, 12, and 13 (Variable Scope, References, Traits, and Lifetimes). This should help you understand more of the implementation I provided you as well as prepare from new features (e.g., traits) that we will see soon.

While the payoff for completion of this task will mostly come when we get to implementing the simplified compiler/parser soon, to demonstrate your completion of this task for now, for simplicity please utilize the Rust Playground (<https://play.rust-lang.org/>) to work through the demonstrations/examples in the videos for your understanding. That is, I should see some of the examples in your playground from these chapters.

**Once you have completed this task, provide a link to your Rust Playgroud here.** To do so, within the playground select Share > Embed Code in Link and then copy the link by clicking on the little clipboard icon and pasting it below. Without this, you will not receive credit for this portion of the lab.

[Challenge 7](https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=8aca3304337cfff97d8eed030b4272ad)

[Challenge 12](https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=7cdc59d954bc2c5c2337d0ad05ffa838)

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

[**Packers**](https://www.youtube.com/watch?v=avRHH9uAL4Q)**!**