Assignment 1

Course: CMPT 340 – Programming Language Paradigms
Name: Connor Morrison
NSID: tvi340
Student Number: 11374770
Date: January 27, 2025

Problem 1.1 in 'Programming Language Pragmatics, Fourth Edition':

a) Lexical Error – Error_a.rs
The lexical error example demonstrates an error that occurs during the scanning phase of the program compilation. In the example, we use an invalid character '$' in an expression. This is a good example of a lexical error because the scanner, which breaks the input into components, immediately recognizes '$' as an invalid character in this context in Rust. The scanner doesn't need to understand the syntax or semantics of the program to detect this error - it simply knows that '$' is not a valid token in this position. Lexical errors are typically the simplest type of error to detect, since they involve only the individual characters of the source code rather than their program logic or relationships.

b) Syntax Error – Error_b.rs
The syntax error example shows a violation of Rust's grammar rules. The missing semicolon after the assignment statement breaks the expected structure of a Rust statement. This is an excellent example of a syntax error because it demonstrates how the parser, which checks if the sequence of tokens follows the language's grammar rules, catches the error. The individual components are valid (unlike a lexical error), but their configuration violates Rust's syntactic rules. Syntax errors present a higher level of analysis than lexical errors, as they are concerned with the structural relationships between components rather than the validity of individual characters. We can detect these errors because we have a complete understanding of the language's grammar and can determine when token sequences don't match the valid language rules.

c) Static Semantic Error – Error_c.rs
The static semantic error example illustrates a type violation that can be caught during semantic analysis. Attempting to assign a string literal to a variable explicitly typed as i32 demonstrates a semantic error that can be detected without running the program. This is a good example because it shows how Rust's type system catches type mismatches during compilation, preventing potential runtime errors. The code is lexically and syntactically valid but violates Rust's semantic rules. Static semantic errors are important in typed languages like Rust because they help ensure type safety at compile time, reducing the likelihood of runtime errors.

d) Dynamic Semantic Error – Error_d.rs
The dynamic semantic error example shows an array bounds violation that can be detected at runtime. While Rust prevents several runtime errors, array index out of bounds errors can still occur. This is a good example because it demonstrates a logical error that cannot be caught at

compile time but is detected by the runtime checks of the compiler. The code is valid according to all static checks (lexical, syntactic, and semantic), but fails during execution. Even in a language with strong static checks like Rust, some conditions can only be validated when the program is running, making these errors an important in language design and implementation.

e) Uncatchable Error – Error_e.rs
The uncatchable error example demonstrates a violation of Rust's type system by attempting to simultaneously assign a single variable to two different types. This is a good example of an error that the compiler cannot reliably detect or generate code to catch because it attempts to break the assumptions of the language itself. The example tries to force a variable to be both an integer and a string type at the same time through pointer manipulation, which violates Rust's type system guarantees. While Rust's type system normally ensures that each variable has only one type, using "unsafe" in this way attempts to circumvent these rules, creating a situation where neither compile-time nor runtime checks can reliably detect the problem. This type of error represents a violation of the language's core design principles rather than just a typical programming error. The use of "unsafe" code in this example doesn't just bypass normal safety checks – it attempts to create a situation that contradicts the basic premises of how Rust's type system works.