1 Introduction

We often think of compilers in terms of their capability to translate source code in one language into target code of anything. Typically the context for this to move from a higher-level language into assembly language, where the assembler interprets (is this true?) the assembly language as machine code. Yet when compilers are taught in the classroom setting, the implementation of their components is generally taught such that source and target language are the same. For the student, this may beg the question "what are compilers really about?"

The process we know as compilation analyzes a source, represented as a stream of characters, at the lexical level, the syntactic level, and the semantic level, places the resulting program representation in a data structure and then *does something*. That *something* could be translating the code from one language into another. It could be rewriting the code to make it more efficient. It could even just return the same code, but with different formatting.

This tutorial will focus on the lexical analysis and parsing.

2 Lexical Analysis

A *lexeme* is the basic and abstract unit of analysis in a program. The concrete analog is a *token*. Tokens are separated by delimiters, typically whitespace. We do not consider tokens that are used for blocking, such as curly braces in Java, or for application[1] , such as the parentheses in Lisp.

[1] *application* here means something very specific - it is the resolution of binding variables to become bound variables. For example, in the Lisp code `(defunsquare(x)(*xx))`, x is the binding variable. We call this example an abstraction. When we call `(square5)`, we bind 5 to x. This is called an application. Note that when we resolve this expression to 25, we have an atom. Thus, the abstraction is the *expression* corresponding to the function call, not its resolution.