Report - Phase 1: Lexical Analysis

Yongun Seong 2017-19937

For the first assignment, I implemented a scanner for SnuPL/2, building upon a simple scanner for a simpler SnuPL/-1 language.

SnuPL/-1, while similar to SnuPL/2, lacks a few important syntatic elements. Significantly, it has no keywords, identifiers, and string or character literals. SnuPL/2 also includes a few additional operators and syntatic elements requiring changes in the scanning function, but the majority of the increase in complexity lies in the above mentioned features.

My process while implementing the scanner can be roughly divided into three phases. First, I examined the grammar in the specification and modified the declaration of various token types in the source code. Here, I added elements into the `EToken` enum such as `tComma` and `tStringConst`, and I removed `tDigit` and `tLetter`, as they were no longer necessary in SnuPL/2. I also ensured the other lists, `ETokenName` and `ETokenStr`, were updated. Finally, I added the language keywords the `Keywords` array.

In the second phase, I updated the code to handle the simpler syntax elements. This included every single token type except the character literal, string literal, number, and identifier. I first updated the code to skip over any comments by skipping any code followed by `//` until the end of the line, using a well-placed `goto` to re-start scanning on the next line. The other syntatic elements were very simple to implement, requiring peeking at most one character.

In the final phase, I implemented the remaining token types. The `tNumber` token, representing a numeric decimal literal, was the simplest, and I only had to consume the numeric literals until there were no more. `tIdent` is similar, with the added constraint that the first character cannot be numeric. I also piggy-backed the `t[Keyword` implementation on `tIdent` as every keyword was a valid identifier. Once a `tIdent` was completely scanned, I checked whether it was a keyword, returning a `t[Keyword` instead if true.

For `tCharConst`, I used the existing `GetCharacter` method, which reads and parses a single (potentially escaped) character. This method handles escapes for both the `tCharConst` and `tStringConst` case, so I only had to check whether it was a immediately closed character (`''`) and consume the closing quote. The `tStringConst` case was quite similar, but it was run in a loop, checking whether the following character was a `"`.