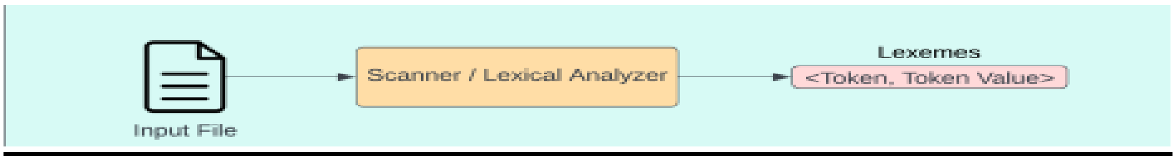
CS 424
Compiler Construction

Assignment 1

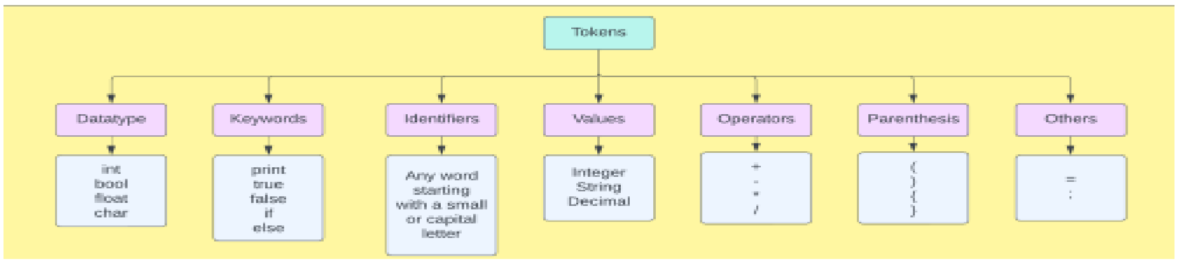
Rafia Faisal

2020406

Scanner Design:



The scanner, which serves as the input for a source file, processes the text we aim to analyze. This text is forwarded to our scanner, also referred to as a lexical analyzer, which interprets the file's content and segments it into lexemes. Each lexeme comprises a token and its associated value. Below, we present visual diagrams of the tokens designed for our scanner, illustrating how the source file's contents are categorized.



Our tokens are divided into seven distinct categories, as described below:

1. **Data Types**: These are reserved keywords that denote the type of a variable being declared, such as int, bool, float, or char.

2. Keywords: These reserved words signify specific actions or values like print, true, false, if, and else.

3. Identifiers: Identifiers serve as names for variables. For instance, in the statement `int age = 3`, `age` is the identifier.

4. Values: Values are the specific data that can be assigned to identifiers. In our system, we recognize three kinds of values: integers, decimals, and strings.

5. Operators: This category includes various operators used for performing mathematical or logical operations in our system.

6. Parentheses: Tokens for parentheses are also included, which are essential for determining the scope of functions or statements.

7. Others: Additionally, we have introduced two more tokens: one for the assignment operator and another for the delimiter, which marks the end of a code segment.

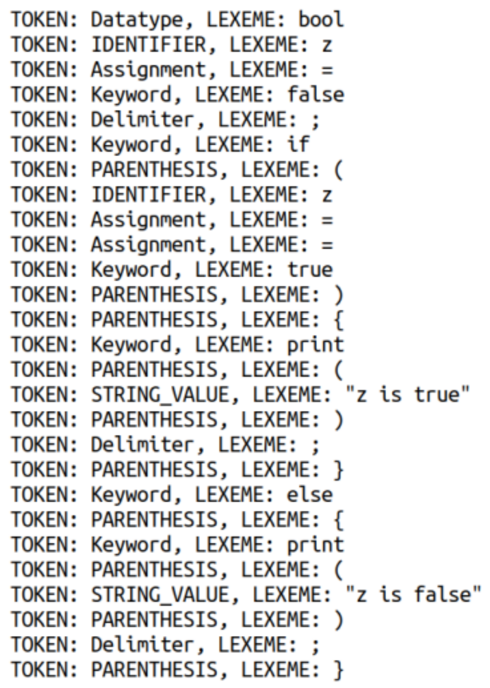
Implementation Details:

For our implementation, we've developed two distinct files: one employs the lex program, and the other does not. In the version without lex, we open the input file in read-only mode using the

`open()` system call from Linux, then read the file's contents and save them into a character array. After reading the entire file, we begin the tokenization of the input stream. We employ simple `if-else` statements to match the input stream with reserved keywords by using the `strcmp()` function for comparison.

In the version that utilizes the lex program, we start by creating a lex file with a `.l` extension. This file contains all our token rules, which are defined using regular expressions in the file's middle section. We then perform lexical analysis on the input stream using the `yylex()` function, which compares the input against the tokens defined by our regular expressions. Below, we provide a visual diagram illustrating this process.

Test Cases:



```
TOKEN: Datatype, LEXEME: bool
TOKEN: IDENTIFIER, LEXEME: z
TOKEN: Assignment, LEXEME: =
TOKEN: Keyword, LEXEME: false
TOKEN: Delimiter, LEXEME: ;
TOKEN: Keyword, LEXEME: if
TOKEN: PARENTHESIS, LEXEME: (
TOKEN: IDENTIFIER, LEXEME: z
TOKEN: Assignment, LEXEME: =
TOKEN: Assignment, LEXEME: =
TOKEN: Keyword, LEXEME: true
TOKEN: PARENTHESIS, LEXEME: )
TOKEN: PARENTHESIS, LEXEME: {
TOKEN: Keyword, LEXEME: print
TOKEN: PARENTHESIS, LEXEME: (
TOKEN: STRING_VALUE, LEXEME: "z is true"
TOKEN: PARENTHESIS, LEXEME: )
TOKEN: Delimiter, LEXEME: ;
TOKEN: PARENTHESIS, LEXEME: }
TOKEN: Keyword, LEXEME: else
TOKEN: PARENTHESIS, LEXEME: {
TOKEN: Keyword, LEXEME: print
TOKEN: PARENTHESIS, LEXEME: (
TOKEN: STRING_VALUE, LEXEME: "z is false"
TOKEN: PARENTHESIS, LEXEME: )
TOKEN: Delimiter, LEXEME: ;
TOKEN: PARENTHESIS, LEXEME: }
```