## Overview

The goal of this project is to build a scanner for a version of the Core language, a pretend language we will be discussing in class.
For this project you are given the following:
“Project 1.pdf" - This handout. Make sure you read it completely and handle all requirements in your implementation. You are encouraged to post any questions on Piazza.
“ScannerOutline.pdf" - These are some lectures slides which outline my recommended approach towards implementing the scanner.
“Main.java", “Core.java", “Scanner.java" - I have outlined the project in this files and give you some of the code you will need. Make no changes to to “Core.java" or “Main.java".
. You may create additional files to contain any additional classes or methods you want to create.
“tester.sh" - This is a script I wrote to help you test your project. It is very similar to the script that will be used to grade your project, so if your project works correctly with this script you are probably doing well. The only guarantee I give is that this script will work on stdlinux.
. Folder “Correct" - This contains some correct inputs and their expected outputs. The “tester.sh" script will test your code against these examples.
. Folder “Error" - This contains some inputs that should generate error messages. The “tester.sh" script will test your code against these examples.
The following are some constraints on your implementation:
. Do not use scanner generators (e.g. lex, ex, ilex, jex, ect) or parser generators (e.g. yacc, CUP, ect)
. Use only the standard libraries of Java.
Your submission should compile and run in the standard linux environment the CSE department provides (stdlinux). I will leave it up to you to decide what IDE you will use or if you will develope your code locally or remotely, but as a final step before submitting your code please make sure it works on stdlinux. Use the subscribe command - make sure you are subscribed to JDK-CURRENT. The graders will not spend any time fixing your code - if it does not compile on stdlinux, your project will not be graded and you will get a 0.
1



## Your Scanner

You are responsible for writing a scanner, which will take as input a text file and output a stream of “tokens" from the Core.java enumeration. You scanner must implement the following methods:
. Scanner: These functions open the file, find the first token, and release memory when we are done scanning.
. currentToken: This function should return the token the scanner is currently on, without consuming that token.

. nextToken: This function should advance the scanner to the next token in the stream (the next token becomes the current token).
. getId: If the current token is ID, then this function should return the string value of the identifier. If the current token is not ID, behavior is undefined.
. getConst: If the current token is CONST, then this function should return the value of the constant. If the current token is not CONST, behavior is undefined.
All of these functions will be necessary for the parser you will write in the second project. You are free to create additional functions.

## Input

The input to the scanner will come from a single ASCII text file. The name of this file will be given as a command line argument to the main function.
The scanner should process the sequence of ASCII characters in this file and should produce the appropriate sequence of tokens. There are two options for how your scanner can
operate:
(1) the scanner can read the entire character stream from the file, tokenize it, stores all the tokens in some list or array and calls to currentToken and nextToken simply walk through the list
or
(2) the scanner reads from the file only enough characters to construct the first token, and then later reads from the file on demand as the currentToken or nextToken functions are called.
Real world scanners typically work as described in (2). In your implementation, you can implement (1) or (2), whichever you prefer.
Once your scanner has scanned the entire file, it should return the EOS token (End Of Stream).
2



## Invalid Input

Your scanner should recognize and reject invalid input with a meaningful error message. The scanner should make sure that the input stream of characters represents a valid sequence of tokens. For example, characters such as ` ' and '%' are not part of a valid sequence of tokens. If your scanner encounters a problem, it should print a meaningful error message to standard out (please use the format "ERROR: Something meaningful here") and return the ERROR token so the main program halts.

## The Language

The Core language consists of 4 kinds of strings, which you will need to tokenize:
. Keywords:
procedure begin is end if else in integer return
do new not and or out array then while
. Identifiers:
Begins with a letter (uppercase or lowercase) followed by zero or more letters/digits.

Refer to this regular expression once we cover regular expressions:
(aj : : : jzjAj : : : jZ)(aj : : : jzjAj : : : jZj0j1j : : : j9)*
. Constants:
Integers from 0 to 100003 (inclusive)
. Symbols:
+ - * / := = < : ; . , ( ) [ ]
Your scanner walk through the input character stream, recognize strings from the language, and return the appropriate token from the enumeration in “Core.java" or “Core.py". If there is any situation in which it is unclear to you which token should be returned, please ask for clarification on Piazza.
Write your scanner with these rules in mind:
1. The language is case sensitive, and the keywords take precedence over the identifiers. For example, “begin" should produce the token BEGIN (not ID), but “bEgIn" should produce the token ID.
2. Strings in the language may or may not be separated by whitespaces. For example the character stream may contain the string “x=10" or the string “x = 10", and both of these should generate the token sequence ID EQUAL CONST.
3. Always take the greedy approach. For example, the string “whilewhile" should produce an ID token instead of two WHILE tokens, string “123" should produce a single CONST token, and string “:=" should produce ASSIGN.
3
4. Keyword/identifier strings end with either whitespace or a non-digit/letter character. For example:
(a) the string “while (" and the string “while(" should both result in the WHILE and LPAREN tokens.
(b) the string “while 12" should result in the WHILE and CONST tokens, but the string “while12" should result in the ID token.
5. Constant strings end with any non-digit character. For example:
(a) the string “120while" or “120 while" should result in the CONST and WHILE tokens.
(b) When applying the greedy rule to constants, you may assume a continuous sequence of digit characters is meant to be a single constant, then report an error if it does not form a valid constant. For example, for strings like “007" or “123456789" your scanner should report an error rather than trying to break them up into valid constants.
6. Symbols may or may not be separated from other strings by whitespace. For example:
(a) String “++while<= =12=" should result in the token sequence ADD ADD WHILE LESS EQUAL EQUAL CONST EQUAL.
Let me know if you think of any situations not covered here.

## Testing Your Project

I have provided some test cases. For each correct test case there are two files (for example 4.code and 4.expected). On stdlinux you can redirect the output of the main program to a file, then use the difi command to see is there is any difierence between your output and

the
expected output. For an example of how to do this you can take a look at the script file "tester.sh".