CSE 3341 Project 1 - Core Scanner

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 files:

- "3341 Project 1.pdf" - This handout. Make sure you read it completely and handle all requirements in your implementation. Please post any questions on Piazza.
- "main.java", "main.py", "Core.java", "Core.py" - Your scanner will interact with these files. Please make no changes to these files.
- "Scanner.java", "Scanner.py" - This is where your scanner implementation will go. You may also create additional files.
- "tester.sh" - This is a script I wrote to help you test your project. It is very similar to the script the graders will use to grade your project, so if your project works correctly with the script you are probably doing well.
- Folder "Correct" - This contains correct inputs and their expected outputs.
- Folder "Error" - This contains some inputs that should generate error messages.

If you are using java, delete the .py files; if you are using python, delete the .java files.

The following are some constraints on your implementation:

- Do not use scanner generators (e.g. lex, flex, jlex, jflex, ect) or parser generators (e.g. yacc, CUP, ect)
- Use only the standard libraries of Java or Python.

Your submission should compile and run in the standard environment on stdlinux. If you work in some other environment, it is your responsibility to port your code to stdlinux and make sure it works there. **Use the subscribe command - make sure you are subscribed to JDK-CURRENT or PYTHON-3.7**. The graders will not spend any time porting your code - **if it does not work on stdlinux, it will not be graded and you will get a 0**.

The Language

We define regular expressions R_1, R_2, R_3, R_4 here:

- R_2 **defines keywords:**
  program | begin | end | new | int | define | endfunc | class | extends | endclass
  | if | then | else | while | endwhile | endif | or | input | output

- R_2 **defines identifiers:**
  (a|...|z|A|...|Z)(a|...|z|A|...|Z|0|1|...|9)*
- R_3 **defines constants:**
  0|1|2|...|1023
- R_4 **defines everything else:**
  ; | (|) | , | = | ! | == | < | <= | + | - | *

Your scanner walk through the input character stream, recognize strings from the language $L(R_1) \cup L(R_2) \cup L(R_3) \cup L(R_4)$, and return the appropriate token from the enumeration in "Core.h". If there is any situation in which it is unclear to you which token should be returned, please ask for clarification on Piazza.

Please note that the language is case sensitive. A keyword takes precedence over an id; for example, "begin" should produce the token BEGIN, but "bEgIn" should produce the token ID.

Please also note that strings in the language may or may not be separated by whitespaces in the character stream. I don't want to overwhelm you by giving you the complete context-free grammar for the language right now. Instead write your scanner with these rules in mind:

1. 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 "===1" should produce EQUAL then ASSIGN then CONST.
2. 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.
3. 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.
4. Special character strings 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 LESSEQUAL ASSIGN CONST ASSIGN.

Let me know if you think of any situations not covered here.

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 language. You scanner must implement the following functions:

- Scanner: the constructor takes as input the name of the input file and finds the first token.
- 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.
- getID: if the current token is ID, then this function should return the string value of the ID.
- getCONST: if the current token is CONST, then this function should return the value of the CONST.

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 and passed to your Scanner function. Your Scanner function will need to open and interact with the file.

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) your 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) upon initialization, 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.

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 allowed in the input stream. If your scanner encounters a problem, it should print a meaningful errormessage (please use the format "ERROR: Something meaningful here") and return the EOF token so the main program halts.

Testing Your Project

I have provided some test cases. For each correct test case (e.g. 04) there are two files (e.g. 04.code and 04.expected). On stdlinux you can redirect the output of the main program to a file, then use the diff command to see is there is any difference 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".

The test cases are weak. You should do additional testing with your own test cases. Feel free to share test cases with each other through piazza.

Project Submission

On or before 11:59 pm Sept 11th, you should submit to the Carmen dropbox for Project 1 a single zip file containing the following:

- Your scanner (just the source code).
- An ASCII text file named README.txt that contains:
  - Your name on top
  - The names of all files you are submitting and a brief description stating what each file contains
  - Any special features or comments on your project
  - Any known bugs in your scanner

If the time stamp on your submission is 12:00 am on Sept 12th or later, you will receive a 10% reduction per day, for up to three days. If your submission is more than 3 days late, it will not be accepted and you will receive zero points for this project. If you resubmit your project, only the latest submission will be considered.

Grading

The project is worth 100 points. Correct functioning of the scanner is worth 65 points. The handling of error conditions is worth 20 points. The implementation style and documentation are worth 15 points.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own; all the design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with severe consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see http://oaa.osu.edu/coamresources.html). If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.