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:

- "3341 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.
- "main.c", "core.h", "scanner.h", "scanner.c" - I have outlines the project in this files, and give you some of the code you will need. Make no changes to to "core.h" file. You can make changes to "main.c" as long as those changes do not change break my tester.sh script.
- 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, flex, jlex, jflex, ect) or parser generators (e.g. yacc, CUP, ect)
- Use only the standard libraries of C. This is the reference I like to use: `https://en.cppreference.com/w/c/header`

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 GCC-10.1.0**. 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**.

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.h enumeration. You scanner must implement the following functions:

- scanner_open and scanner_close: 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.

To make things easier for you, you may assume no token is made of more than 20 characters. Also, I suggest using calloc to allocate memory, instead of malloc.

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).

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 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:**
  and begin do else end if in integer
  is new not or out procedure record 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:
  (a|...|z|A|...|Z)(a|...|z|A|...|Z|0|1|...|9)*
- **Constants:**
  Integers from 0 to 1009 (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.

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.

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 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 create and post additional test cases on piazza.

Project Submission

On or before 11:59 pm January 27th, you should submit to the Carmen dropbox for Project 1 a single zip file containing the following:

- All your .java or .py files.
- 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 January 28th 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 errors 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.