CSE 3341 Project 1 - CORE Scanner

Overview

The goal of this project is to build a Scanner for a version of the Core language discussed in class. This handout defines a variation of this language; make sure you implement a scanner for the version described in this handout, not the version described in the lecture notes. **You must use one of these languages for the project implementation: C, C++, Java, or Python**. Please note that using C will restrict you implementation in the second project to a non-Object Oriented approach. This does not mean it is harder to implement the project in C, just that you will have less options.

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)
- Do not use functionality from external libraries for complex text processing. However, you can use all functionality from stdlib.h/string.h or java.lang.String.

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. The graders will not spend any time porting your code - **if it does not work on stdlinux, they will not grade it**.

You scanner will implement a variation of the CORE grammar, which is given on the last page. **Please note that the grammar 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.

The semantics of this modified language should be obvious; if some aspects of the language are not, please bring it up on Piazza or in class for discussion. You need to write a scanner for the exact language described. Every lexically valid input for this language should be tokenized correctly, and every lexically invalid input for this language should be rejected with a meaningful error message.

Starting Files

You should start with the following files from Carmen:

- C: main.c, Scanner.h and Core.h
- C++: main.cpp, Scanner.h and Core.h
- Java: main.java, Scanner.java and Core.java
- Python: main.py, Scanner.py and Core.py

The "Core" files contain an enumeration you should use to represent your tokens. The "main" file demonstrates how the scanner should be interacted with and is how you should test your implementation. The "Scanner" file should contain the implementation of your Scanner, and contains a dummy class/functions so the provided files can be compiled and ran (the "main" file will contain a comment stating how to compile and run the starting files). Do not edit the Core or main files.

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: this function takes as input the name of the input file, and initializes the scanner. If you are using an object oriented language, this should be your class constructor.
- currentToken(): this function should return the token the scanner is currently on, without consuming that token.
- nextToken(): this function should return the token the scanner is currently on, and 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 add additional functions to aid with error checking.

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 arguments 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. You should use the standard libraries for I/O in your chosen language to interact with this file. As discussed in class, getting the tokens is typically done on demand: the parser asks the scanner for the current token, or moves to the next token. There are two options for creating the token stream: (1) upon initialization, the scanner reads the entire program from the file, tokenizes it, and stores all tokens in some list or array, 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 compilers and interpreters use (2); in your implementation, you can implement (1) or (2), whichever you prefer.

For example, if this is the input file

```
program
int                                          x
, y, z; begin input x, y;
z:= x; x :=y; y:=
z;              output x; output y ; end
```

running the main function should produce the following output:

```
PROGRAM
INT
ID[x]
COMMA
ID[y]
COMMA
ID[z]
SEMICOLON
BEGIN
INPUT
ID[x]
COMMA
ID[y]
SEMICOLON
ID[z]
ASSIGN
ID[x]
SEMICOLON
ID[x]
ASSIGN
ID[y]
SEMICOLON
ID[y]
ASSIGN
ID[z]
SEMICOLON
OUTPUT
ID[x]
SEMICOLON
OUTPUT
ID[y]
SEMICOLON
END
```

Lexical Analysis

When implementing the scanner, do not use external libraries or tools that allow complex text processing. The purpose of these restrictions is to give you hands-on experience with the implementation details of lexical analysis, using only built-in functionality from mainstream languages. Some suggestions for building the scanner are included at the end of the lecture notes on grammars. If you need clarification for some scanning issues, ask a question in class or on piazza.

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 EOS token so the main program halts.

Testing Your Project

On the course web page I will post some test cases. For each test case (e.g. t4) there are two files (e.g. t4.code and t4.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.

The test cases are very 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 January 27th, you should submit the following:

- Your scanner (just the source code).
- An ASCII text file named README.txt that contains:
  - Your name on top
  - Whether or not you want your project to be considered for the extra credit
  - The names of all files you are submitting and a brief description stating what each file contains
  - Instructions on how to compile and run the project (please be exact here)
  - Any special features or comments on your project
  - Any known bugs in your scanner

Submit your project as a single zipped file to the Carmen dropbox for Project 1.

If the time stamp on your submission is 12:00 am on January 27th 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.

If the graders have problems compiling or executing your program, they will email you; you must respond within 48 hours to resolve the problem. Please check often your xyz.123@osu.edu account after submitting the project in case the graders need to get in touch with you.

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.

Extra Credit

I will be selecting 4 projects (one for each language) to serve as a basis for Project 2, so that any students who are not confident in their scanner will have a correct one to use in Project 2. This selection will be based on correctness, code style, and quality of documentation/comments. If you want your project to be considered, please make sure to mention this in your README file.

The projects selected will all receive 5 bonus 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.

<prog> ::= **program** <decl-seq> **begin** <stmt-seq> **end**

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= <decl-id> | <decl-func>

<decl-id> ::= **int** <id-list> **;**

<decl-func> ::= <id**> (** <id-list> **) begin** <stmt-seq> **endfunc ;**

<id-list> ::= <id**>** | <id**> ,** <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out>

<assign> ::= <id**> :=** <expr> **;**

<in> ::= **input** <id-list**> ;**

<out> ::= **output** <expr> **;**

<if> ::= **if** <cond> **then** <stmt-seq> **endif ;**

| **if** <cond> **then** <stmt-seq> **else** <stmt-seq> **endif ;**

<loop> ::= **while** <cond> **begin** <stmt-seq> **endwhile ;**

<cond> ::= <cmpr> | **! (** <cond> **)**

| <cmpr> **or** <cond>

<cmpr> ::= <expr> **=** <expr> | <expr> **<** <expr>

| <expr> **<=** <expr>

<expr> ::= <term> | <term> **+** <expr> | <term> **–** <expr>

<term> ::= <factor> | <factor> ***** <term>

<factor> ::= <const> | <id> | **(** <expr> **)**

<const> ::= **0** | **1** | **2** | … | **1023**

<id> ::= <letter> | <id><letter> | <id><digit>

<letter> ::= **a** | **b** | … | **z** | **A** | **B** | **…** | **Z**

<digit> ::= **1** | **2** | … | **9**