Assignment 3

CSCI 3136: Principles of Programming Languages

Due Mar 5, 2018

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

Banner ID: _____ Name: _____

The next three programming assignments together implement an interpreter for a FORTH-like programming language called TOY. The description can be found at http//www.cs.dal.ca/ nzeh/Teaching/3136/Assignments/toylang.html. This document describes the way TOY works and also includes, towards the end of the document, a specification of the lexical and syntactic structure of a valid TOY program.

In this assignment, you are to implement a scanner for this language. Thus, you'll have to consult the description of all valid tokens of the language and implement a scanner that accepts exactly these tokens.

1 Programming Languages You May Use

To keep things manageable for the TAs, you must choose one of three languages to implement your scanner: Python, Java or C/C++. Since the parser and semantic analyzer in the next two programming assignments each build on the scanner, I will make sample implementations available so you can complete the next assignment even if you failed the current one. Note, however, that I will only make a Python implementation available. Thus, if you choose a language other than Python for this assignment and fail to complete it, you may be forced to switch to Python in the next assignment.

2 Requirements

Implement your scanner in a file `scanner.py`, `Scanner.java` or `scanner.(c|cpp)`. You may split your code into multiple files, but the main file must be `scanner.(py|java|c|cpp)`. This file should expect one command line argument, the name of a TOY program file. If the program is composed of lexically correct tokens (but may or may not be syntactically correct), the scanner should print the recognized tokens and the text of each token to the screen. For example, for the program

```
fun fibonacci // n -- F_n

  // TOY supports local functions
  fun fib // n-i F_{i-1} F_i -- F_n
    rotl dup 0 =
    [ drop swap drop ]
    [ 1 - rotr dup rotl + fib ] ??
  .

  0 1 fib
.
```

the token stream should be

```
fun(fun) id(fibonacci) fun(fun) id(fib) id(rotl) id(dup) int(0) id(=) lambdabegin([)
id(drop) id(swap) id(drop) lambdaend(]) lambdabegin([) int(1) id(-) id(rotr) id(dup)
id(rotl) id(+) id(fib) lambdaend(]) id(??) end(.) int(0) int(1) id(fib) end(.)
```

If the input contains a word (part of the input delimited by spaces) that is not a valid token, the scanner should report a lexical error including the first invalid token and its position in the input.

While the above is the required output of the program, make sure that, internally, the scanner reports tokens as integer IDs (represented as an `enum` type in C/C++ or using appropriate "constants" in Python). This is essential so the scanner can efficiently compare tokens with expected token values without performing string comparisons. For example, we could define token values as

```
typedef enum {
  FUN = 1,
  ID = 2,
  INT = 3,
  ...
} Token;
```

Then the scanner should produce the sequence of integer values `1 2 1 2 2 2 3 ...` for the program above, along with the associated strings of the different tokens. In order to meet the above output requirements, you should transform these integers into the human-readable form above before printing them to stdout.

You are free to implement the scanner as a separate pass that completely scans the input before passing the resulting token stream to the parser or as an on-demand scanner that reports the next available token to the parser whenever the parser asks for it.

3 Submission Instructions

Put your scanner implementation into a single zip-file and email this file to Arash Kayhani (arash.kayhani@dal.ca).