Exercise 1

Due 3/11/2021, 23:59

1 Introduction

During the semester we will implement a compiler to an invented object oriented language called TBD . In order to make this document self contained, all the information needed to complete the first exercise is brought here.

2 Programming Assignment

The first exercise implements a lexical scanner based on the open source tool JFlex. The input for the scanner is a single text file containing a TBD program, and the output is a (single) text file containing a tokenized representation of the input. The course repository contains a simple skeleton program, and you are encouraged to work your way up from there.

3 Lexical Considerations

Identifiers may contain letters and digits, and must start with a letter.

Keywords in Table 1 can *not* be used as identifiers.

class	nil	array	while
extends	return	new	if

Table 1: Reserved keywords of TBD .

White spaces consist of spaces, tabs and newlines characters. They may appear between any tokens. Keywords and identifiers must be separated by a white space, or a token that is neither a keyword nor an identifier.

letters	digits	white spaces	line terminators
( ) [ ] { }	? !	+ − ∗ /	. (dot) ; (semicolon)

Table 2: Characters that may appear inside comments in TBD .

Comments in TBD are similar to those used in the C programming language, but may only contain characters from Table 2. A comment beginning with the characters // indicates that the remainder of the line is a comment. In addition, a comment can be a sequence of characters that begins with /*, followed by any characters from Table 2, up to the first occurrence of the end sequence */. Comments that contain characters that do not appear in Table 2 are lexical errors. Unclosed comments are lexical errors too.

Integer literals may start with an optional negation sign −, followed by a sequence of digits. Integers should *not* have leading zeroes, and when they do, it is a lexical error. In addition, −0 is a lexical error. Though integers are stored as 32 bits in memory, they are artificially limited in TBD to have 16-bits signed values between -2^{15} and $2^{15} - 1$. Integers out of this range are lexical errors.

Strings are sequences of (zero or more) letters between double quotes. Strings that contain non letter characters are lexical errors. Unclosed strings are lexical errors too.

4 Input

The input for this exercise is a single text file, the input TBD program.

5 Output

The output is a single text file that contains a tokenized representation of the input program. Each token should appear in a separate line, together with the line number it appeared on, and the character position inside that line. The list of token names appears in Table 3, and will only be used in this first exercise. Later phases of the compiler will make no use of these token names. Three types of tokens are associated with corresponding values: integers, identifiers and strings. The printing format for these tokens can be easily deduced from the examples in Table 4. Whenever the input program contains a lexical error, the output file should contain a *single* word only: ERROR.

Token Name	Description		Token Name	Description
LPAREN	(		ASSIGN	:=
RPAREN	)		EQ	=
LBRACK	[		LT	<
RBRACK	]		GT	>
LBRACE	{		ARRAY	
RBRACE	}		CLASS	
NIL			EXTENDS	
PLUS	+		RETURN	
MINUS	−		WHILE	
TIMES	∗		IF	
DIVIDE	/		NEW	
COMMA	,		INT(*value*)	*value* is an integer
DOT	.		STRING(*value*)	*value* is a string
SEMICOLON	;		ID(*value*)	*value* is an identifier
ELLIPSIS	...			

Table 3: Token names for the first exercise. Note that three types of tokens are associated with corresponding values: integers, identifiers and strings. The rest of the tokens encountered only contain their name.

6 Submission Guidelines

Open an account on GitHub. Then, visit the academic discount page to enable the free creation of private repositories. One team member should create a new *private* repository called *compilation*, and then invite other team members, the course grader (omarmahamid) and myself (davidtr1037) as collaborators. Please put inside the uppermost folder (*compilation*) the following text files:

- *ids.txt:* the ID's of all team members (one ID per line).
- *users.txt:* the GitHub user names of all team members (one user name per line).
- *names.txt:* the *hebrew* names of all team members (one name per line).

In addition, *compilation* should contain a sub folder called *ex1* where your code will reside. *compilation/ex1* should contain a makefile building your source files to a runnable jar file called LEXER (note the lack of the .jar suffix). Feel free to use the makefile supplied in the course repository, or write a new one if you want to. Before you submit, make sure that your exercise compiles and runs on the school server: *nova.cs.tau.ac.il*. This is the formal running environment of the course. You are encouraged to use the makefile provided by the exercise skeleton, which can be found (along with the other files...) in the course repository using the following link:
`https://github.com/davidtr1037/compilation-tau/tree/master/src/ex1`

Printed Lines Examples	Description
LPAREN[7,8]	left parenthesis is encountered in line 7, character position 8
INT(74)[3,8]	integer 74 is encountered in line 3, character position 8
STRING(“Dan”)[2,5]	string “Dan” is encountered in line 2, character position 5
ID(numPts)[1,6]	identifier numPts is encountered in line 1, character position 6

Table 4: Printing format for the first exercise. Each line in the output text file should contain the token name, the line number they appeared on, and the character position inside that line. Note that three types of tokens are associated with corresponding values: integers, identifiers and strings. These values should appear inside the parentheses as shown.

Execution parameters LEXER receives 2 file names:

- Input.txt
- OutputTokenizedProgram.txt