Exercise 1

Compilation 0368-3133

Due 28/11/2024, 23:59

1 Introduction

During the semester we will implement a compiler to an object oriented language called **L**. 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 **L** 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	int	void
extends	return	new	if	string	

Table 1: Reserved keywords of **L**.

White spaces consist of spaces, tabs and newlines characters. They may appear between any tokens.

Comments in **L** are similar to those used in the C programming language, but may only contain characters from Table 2.
Type-1 Comment:
A sequence of characters that begins with // followed by any character from Table 2 up to the first occurrence of a newline character. Note, for example, that //a$ is not considered as a *Type-1* comment.
Type-2 Comment:
A sequence of characters that begins with /* followed by any characters from Table 2, up to the first occurrence of the end sequence */. An unclosed *Type-2* comment (that is, a missing */) is a lexical error. For example, the input /*a is a lexical error, but a*/ is valid.

Comments (of both types) that contain characters that do not appear in Table 2 are lexical errors.

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

Table 2: Characters that may appear inside comments in **L**.

Integers in **L** are sequences of digits. Integers should *not* have leading zeroes, and when they do, it is a lexical error. Though integers are stored as 32 bits in memory, they are artificially limited in **L** to have 16-bits signed values between 0 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 **L** 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. 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	array
RBRACE	}		CLASS	class
NIL	nil		EXTENDS	extends
PLUS	+		RETURN	return
MINUS	−		WHILE	while
TIMES	∗		IF	if
DIVIDE	/		NEW	new
COMMA	,		INT(*value*)	*value* is an integer
DOT	.		STRING(*value*)	*value* is a string
SEMICOLON	;		ID(*value*)	*value* is an identifier
TYPE_INT	*int*		TYPE_STRING	string
TYPE_VOID	*void*			

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.

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.

6 Submission Guidelines

6.1 Project Repository

Create an account on GitHub.
Then, visit this page to enable 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 (galvien) and myself (noa-schiller) 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 subfolder called *ex1* where your code will reside. *compilation/ex1* should contain a makefile (at *compilation/ex1/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.

6.2 School Server

The official running environment of the course is a Docker image that runs in a Docker container on the school server: *nova.cs.tau.ac.il*. To access it:

1. Connect to nova.cs.tau.ac.il:
   `ssh <username>@nova.cs.tau.ac.il`
2. Define containers location (it can be changed at .login file so it will be auto-defined every ssh):
   `setenv UDOCKER_DIR /vol/csrepo/`
3. Start the container:
   `udocker run --bindhome compiler_course`

Note that JFlex is already installed there and can be used with the `jflex` command.
Before submitting, ensure your exercise compiles and runs inside the Docker container.

6.3 Command-line usage

LEXER receives 2 parameters (file paths):

- *input* (input file path)
- *output* (output file path containing the expected output)

6.4 Skeleton

You are encouraged to use the makefile provided by the exercise skeleton, which can be found in the course repository using the following link:
`https://github.com/noa-schiller/compilation-tau/tree/main/skeletons/ex1`.
Some other files of interest in the provided skeleton:

- *jflex/LEX_FILE.lex* (LEX configuration file)
- *src/TokenNames.java* (enum with token types)
- *src/Main.java*
- The directories *input* and *expected_output* contain input tests and their corresponding expected results

To use the skeleton, run the following command (in the *ex1* directory):
`$ make`
This performs the following steps:

- Generates *src/Lexer.java*
- Compiles the modules into *LEXER*
- Runs *LEXER* on *input/Input.txt*

6.5 Environment requirements

- Ubuntu 14.04 / 16.04
- Java 8

6.6 GitHub related material

- `https://guides.github.com/activities/hello-world/`
- `https://docs.github.com/en/get-started`