Exercise 1 – Lexical Analysis

Compilation 0368-3133

Due 13/11/2025, 23:59

1 Assignment Overview

During the semester we will implement a compiler for an object oriented language called **L**. 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 text file containing a tokenized representation of the input. The exercise repository contains a simple skeleton program, and you are encouraged to work your way up from there.

2 Tokens and Lexical Rules

Token Name	Description		Token Name	Description
LPAREN	(		ASSIGN	:=
RPAREN	)		EQ	=
LBRACK	[		LT	<
RBRACK	]		GT	>
LBRACE	{		ARRAY	array
RBRACE	}		CLASS	class
PLUS	+		RETURN	return
MINUS	−		WHILE	while
TIMES	∗		IF	if
DIVIDE	/		ELSE	else
COMMA	,		NEW	new
DOT	.		EXTENDS	extends
SEMICOLON	;		NIL	nil
TYPE_INT	int		INT(*value*)	*value* is an integer
TYPE_STRING	string		STRING(*value*)	*value* is a string
TYPE_VOID	void		ID(*value*)	*value* is an identifier

Table 1: Tokens in **L**.

Table 1 contains the list of valid tokens in **L** along with their names.

- **Identifiers:** Start with a letter and may contain only letters and digits.
- **Keywords:** Cannot be used as identifiers. Reserved keywords of **L**:

  `class`, `nil`, `array`, `while`, `int`, `void`, `extends`, `return`, `new`, `if`, `else`, `string`

- **Integers:** Sequences of digits. Integers should *not* have leading zeroes, and when they do, it is a **lexical error**. Though integers are stored as 32-bit values in memory, they are artificially limited in **L** to 16-bit signed values between 0 and $2^{15} - 1$. Integers out of this range are **lexical errors**.

- **Strings:** Sequences of (zero or more) letters between double quotes. Strings that contain non-letter characters are **lexical errors**. Unclosed strings are **lexical errors** too.
- **White spaces:** Consist of spaces, tabs and newline characters. They may appear between any tokens.
- **Comments** Similar to those used in the C programming language, but may only contain characters from Figure 2. There are two types of comments in **L**. Comments (of both types) that contain characters that do not appear in Figure 2 are **lexical errors**.
  - *Type-1 Comment*: A sequence of characters that begins with // followed by any character from Figure 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 Figure 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.

letters, digits, white spaces, `(`, `)`, `[`, `]`, {, }, `?`, `!`, `+`, `-`, `*`, `/`, `.`, `;`

Figure 2: Characters that may appear inside comments in **L**. Note: Commas are used only to separate the characters. "White spaces" refers to the definition given above.

3 Input and Output

Input: A single text file containing a **L** program.

Output: A text file containing a tokenized representation of the input program. Each token should appear in a separate line, with: (1) the token name, (2) line number it appeared on, and (3) character position inside the line. Three types of tokens are associated with corresponding values: integers, identifiers and strings. The printing format for all tokens can be deduced from the examples in Table 3. White spaces and comments are ignored and produce no output. Whenever the input program contains a lexical error, the output file should contain a *single* word only: `ERROR`. A lexical error occurs if a sequence of characters cannot be matched to any token according to the lexical analysis algorithm presented in class, or if any of the lexical errors described in this document occur.

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 3: 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. Tokens with associated values (integers, identifiers, and strings) should include the value in parentheses, as shown.

4 Submission Guidelines

Each group should have **only one** student submit the solution via the course Moodle. Submit all your code in a single zip file named `<ID>.zip`, where `<ID>` is the ID of the submitting student. The zip file must have the following structure at the top level:

1. A text file named `ids.txt` containing the IDs of all team members (one ID per line).
2. A folder named `ex1/` containing all your source code.

Inside the `ex1` folder, include a makefile at `ex1/Makefile`. This makefile must build your source files into the lexer, which should be a runnable jar file located at `ex1/LEXER` (note the lack of .jar suffix). You may reuse the makefile provided in the skeleton, or create a new one if you prefer.

Command-line usage: `LEXER` receives 2 parameters (file paths):

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

Self-check script: Before submitting, you *must* run the self-check script provided with this exercise. This script verifies that your source code compiles successfully using your makefile and passes several provided tests. You must execute the self-check on your submission zip file within the school server (`nova.cs.tau.ac.il`) and ensure that your submission passes all checks. Follow these steps:

1. Download `self-check-ex1.zip` from the course Moodle and place it in the same directory as your submission zip file. Make sure no other files are present in that directory.
2. Run the following commands:
   ```
   unzip self-check-ex1.zip
   python self-check.py
   ```

Make sure that your `ids.txt` file follows the required format, and that your makefile does *not* contain hard-coded paths. The self-check script does **not** verify these aspects automatically and you are responsible for ensuring that your code compiles correctly in any directory.

Submissions that fail to compile or do not produce the required runnable will receive a grade of 0, and resubmissions will not be allowed.

5 Starter Code

The exercise skeleton can be found in the repository for this exercise. The skeleton provides a basic lexical scanner, which you may modify as needed. Some 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 `input` and `expected_output` folders contain input tests and their expected results.

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

- Generates `src/Lexer.java` from the LEX file
- Compiles the modules into `LEXER`
- Runs `LEXER` on `input/Input.txt`