Exercise 2 – Syntax Analysis

Compilation 0368-3133

Due 27/11/2025, before 23:59

1 Assignment Overview

In this exercise, you will extend your JFlex scanner from the first exercise with a CUP-based parser for **L**. The parser takes a single text file containing a **L** program and outputs a text file indicating whether the program is syntactically valid. When the input is correct, the parser should also build an abstract syntax tree (AST). The exercise repository provides a simple skeleton parser that indicates whether the input program has correct syntax, and internally builds an AST for a small subset of **L**. As always, you are encouraged to work your way up from there, but feel free to write the whole exercise from scratch if you want to. Note that the AST will not be checked in this exercise. It is needed for later phases (semantic analysis and code generation), and the current exercise is the best time to design and implement it.

2 The L Syntax

Table 2 specifies the BNF grammar of **L**, where the start symbol is program. The terminals in the grammar are defined according to the token definitions of the first exercise. Since CUP only supports context-free grammars, you will need to modify the given grammar to standard CFG notation. Finally, feed the grammar to CUP and ensure that no conflicts occur.

Operator precedence: The precedence and associativity of operators in **L** are listed in Table 1. These rules determine how expressions are parsed in the presence of multiple operators. Make sure to test that your AST construction correctly implements operator precedence and associativity. Do this not only for cases involving arithmetic operators, but also for the rest of the operators listed in the table. For example, `a+b.f()` should be parsed as `a+(b.f())` and not `(a+b).f()`. You can use the AST image generated by the starter code and implement the `printMe` method in the new AST nodes to help check this. Note that this exercise does not check whether your AST correctly represents operator associativity and precedence. However, an incorrect structure may cause issues in later exercises.

Precedence	Operator	Description	Associativity
1	:=	assign	
2	=	equals	left
3	<, >		left
4	+, −		left
5	∗, /		left
6	[	array indexing	
7	(	function call	
8	.	field access	left

Table 1: Operators of **L** along with their associativity and precedence. 1 stands for the lowest precedence, and 8 for the highest.

```
program        ::=  dec { dec }

dec            ::=  varDec | funcDec | classDec | arrayTypedef

type           ::=  TYPE_INT | TYPE_STRING | TYPE_VOID | ID

varDec         ::=  type ID [ ASSIGN exp ] SEMICOLON
                 |  type ID ASSIGN newExp SEMICOLON

funcDec        ::=  type ID LPAREN [ type ID { COMMA type ID } ] RPAREN LBRACE stmtList RBRACE
stmtList       ::=  stmt { stmt }

classDec       ::=  CLASS ID [ EXTENDS ID ] LBRACE cField { cField } RBRACE
cField         ::=  varDec | funcDec

arrayTypedef   ::=  ARRAY ID EQ type LBRACK RBRACK SEMICOLON

exp            ::=  var
                 |  LPAREN exp RPAREN
                 |  exp (PLUS | MINUS | TIMES | DIVIDE | LT | GT | EQ) exp
                 |  callExp
                 |  [ MINUS ] INT | NIL | STRING

newExp         ::=  NEW type
                 |  NEW type LBRACK exp RBRACK

callExp        ::=  [ var DOT ] ID LPAREN [ exp { COMMA exp } ] RPAREN

var            ::=  ID
                 |  var DOT ID
                 |  var LBRACK exp RBRACK

stmt           ::=  varDec
                 |  var ASSIGN exp SEMICOLON
                 |  var ASSIGN newExp SEMICOLON
                 |  RETURN [ exp ] SEMICOLON
                 |  IF LPAREN exp RPAREN LBRACE stmtList RBRACE [ ELSE LBRACE stmtList RBRACE ]
                 |  WHILE LPAREN exp RPAREN LBRACE stmtList RBRACE
                 |  callExp SEMICOLON
```

Table 2: BNF grammar for the **L** programming language.

3 Input and Output

Input: The input for this exercise is a single text file, the input **L** program.

Output: The output is a *single* text file that contains a *single* word:

- When the input program has correct syntax: `OK`
- When there is a lexical error: `ERROR`
- When there is a syntax error: `ERROR(`*location*)
  (where *location* is the line number of the *first* error that was encountered.)

4 Implementation Notes

- Take advantage of CUP's precedence and associativity declarations so you can keep the grammar small without encoding this information explicitly and avoiding, for example, the expression/term/factor rules for binary operations.
- CUP automatically generates the file `TokenNames.java`, which contains the token definitions derived from the terminal declarations in your CUP file. Therefore, unlike in the previous exercise, this file should not be included in your submitted source code. Note that the tokens `EOF` and `error` are generated automatically and do not require explicit terminal declarations.
- There is no single correct way to design the AST structure and define the various AST node classes. Keep in mind that in the next exercises we will traverse the AST and perform different actions based on each node type.
- In the next exercise you will traverse the AST and perform semantic checks. Whenever an error is detected, it must be reported together with the line number in which it occurred. For this reason, your AST nodes should store the line number of the command or construct they represent, and you can already implement this in the current exercise.

5 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 `ex2/` containing all your source code.

Inside the `ex2` folder, include a makefile at `ex2/Makefile`. This makefile must build your source files into the parser, which should be a runnable jar file located at `ex2/PARSER` (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: `PARSER` 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-ex2.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-ex2.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.

6 Starter Code

The exercise skeleton can be found in the repository for this exercise. The skeleton provides a basic parser, which you may modify as needed. Some files of interest in the provided skeleton:

- `cup/CUP_FILE.lex` (CUP configuration file)
- `src/AST/*.java` (classes for the AST nodes)
- The `input` and `expected_output` folders contain input tests and their expected results.

Note that you need to use *your own* LEX configuration file from the previous exercise, as the one in the skeleton provides only a partial implementation.

Building and running the skeleton: From the `ex2` directory, run:

```
$ make
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into `PARSER`

For debugging, run:

```
$ make debug
```

This performs the following additional steps:

- Runs `PARSER` on `input/Input.txt`
- Generates an image of the resulting syntax tree (for debugging only)
- Displays the generated image. This step runs eog in the background, and it may take a few moments to appear.

Note: the steps performed in debug mode should not be performed when running `make` using your own makefile.

Dependencies: To generate and view the AST image outside the school server, you may need to install the following tools :

```
sudo apt-get install graphviz eog
```