

# Exercise 2

Due 11/12/2019, before 23:55

## 1 Introduction

We continue our journey of building a compiler for the invented object oriented language **L**. In order to make this document self contained, all the information needed to complete the second exercise is brought here.

## 2 Programming Assignment

The second exercise implements a CUP based parser on top of your JFlex scanner from the exercise 1. The input for the parser is a single text file containing a **L** program, and the output is a (single) text file indicating whether the input program is syntactically valid or not. In addition to that, whenever the input program has correct syntax, the parser should internally create the abstract syntax tree (AST). Currently, the course repository contains 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 also, that the AST will not be checked in exercise 2. It is needed for later phases (semantic analyzer and code generation) but the best time to design and implement the AST is exercise 2.

## 3 The L Syntax

Table 1 specifies the context free grammar of **L**. You will need to feed this grammar to CUP, and make sure there are no shift-reduce conflicts. The operator precedence is listed in Table 2.

## 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 *single* word. Either OK when the input program has correct syntax, or otherwise ERROR(*location*), where *location* is the line number of the *first* error that was encountered.

## 6 Submission Guidelines

The skeleton for this exercise can be found here. In your project, you need to add the relevant derivation rules and AST constructors. Your project must contain the makefile in the following path:

- ex2/Makefile

This makefile should build the parser (a runnable jar file) in the following path:

- ex2/PARSER

Feel free to reuse the makefile supplied in the skeleton, or write a new one if you want to.

### 6.1 Command-line usage

*PARSER* receives 2 parameters (file paths):

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

### 6.2 Skeleton

You are encouraged to use the makefile provided by the skeleton. Some files of interest in the provided skeleton:

- *jflex/LEX\_FILE.lex* (LEX configuration file)
- *cup/CUP\_FILE.lex* (CUP configuration file)
- *src/Main.java*
- *src/AST/\*.java*

To use the skeleton, run the following command (in the *src/ex2* directory):

```
$ make
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *PARSER*
- Runs *PARSER* on *input/Input.txt*
- Generates an image of the resulting syntax tree (for debugging only)

|          |     |   |
|----------|-----|---|
| Program  | ::= | dec <sup>+</sup>  |
| dec      | ::= | varDec   funcDec   classDec   arrayDec  |
| type     | ::= | TYPE_INT   TYPE_STRING   ID   |
| varDec   | ::= | TYPE ID [ ASSIGN exp ] ';' ;<br>TYPE ID ASSIGN newExp ';' ;   |
| funcDec  | ::= | TYPE ID '(' [ TYPE ID [ ',' TYPE ID ]* ] ')' '{' stmt [ stmt ]* '}'   |
| classDec | ::= | CLASS ID [ EXTENDS ID ] '{' cField [ cField ]* '}'  |
| arrayDec | ::= | ARRAY ID = TYPE '[' ']'   |
| exp      | ::= | var ;<br>'(' exp ')';<br>exp BINOP exp ;<br>[ var '.' ] ID '(' [ exp [ ',' exp ]* ] ')' ;<br>[ '-' ] INT   NIL   STRING   |
| var      | ::= | ID ;<br>var '.' ID ;<br>var '[' exp ']' ;   |
| stmt     | ::= | varDec ;<br>var ASSIGN exp ';' ;<br>var ASSIGN newExp ';' ;<br>RETURN [ exp ] ';' ;<br>IF '(' exp ')' '{' stmt [ stmt ]* '}' ;<br>WHILE '(' exp ')' '{' stmt [ stmt ]* '}' ;<br>[ var '.' ] ID '(' [ exp [ ',' exp ]* ] ')' ';' ; |
| newExp   | ::= | NEW ID   NEW ID '[' exp ']'   |
| cField   | ::= | varDec   funcDec  |
| BINOP    | ::= | +   -   *   /   <   >   =   |
| INT      | ::= | [1 - 9][0 - 9]*   0   |

Table 1: Context free grammar for the **L** programming language.

| Precedence | Operator                | Description    | Associativity |
|------------|-------------------------|----------------|---------------|
| 1          | <code>:=</code>         | assign         |               |
| 2          | <code>=</code>          | equals         | left          |
| 3          | <code>&lt;, &gt;</code> |                | left          |
| 4          | <code>+, -</code>       |                | left          |
| 5          | <code>*, /</code>       |                | left          |
| 6          | <code>[</code>          | array indexing |               |
| 7          | <code>(</code>          | function call  |               |
| 8          | <code>.</code>          | field access   | left          |

Table 2: Binary operators of **L** along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.