

# Exercise 2

Compilation 0368:3133

Due 15/11/2017

## 1 Introduction

We continue our journey of building a compiler for the invented object oriented language RioMare. Remember that the entire specification of RioMare appears inside the relevant folder of the course website. In order to make this document self contained, all the information needed to complete the second exercise is brought here again.

## 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 RioMare 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 RioMare. 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 RioMare Syntax

Table 1 specifies the context free grammar of RioMare. 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 RioMare program.

Program	::=	dec <sup>+</sup>
dec	::=	funcDec   varDec   classDec
varDec	::=	ID ID [ ASSIGN exp ] ';'
funcDec	::=	ID ID '(' [ ID ID [ ',' ID ID ]* ] ')' '{' stmt [ stmt ]* '}'
classDec	::=	CLASS ID [ EXTENDS ID ] '{' cField [ cField ]* '}'
exp	::=	'(' exp ')'
	::=	exp BINOP exp
	::=	[ var '.' ] ID '(' [ exp [ ',' exp ]* ] ')'
	::=	INT   NIL   STRING   NEW ID   var
var	::=	ID
	::=	var '.' ID
	::=	var '[' exp ']'
stmt	::=	varDec
	::=	var ASSIGN exp ';'
	::=	RETURN [ exp ] ';'
	::=	IF '(' exp ')' '{' stmt [ stmt ]* '}'
	::=	WHILE '(' exp ')' '{' stmt [ stmt ]* '}'
	::=	[ var '.' ] ID '(' [ exp [ ',' exp ]* ] ') ';'
cField	::=	varDec   funcDec

Table 1: Context free grammar for the RioMare programming language.

## 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 code for this exercise resides (as usual) in subdirectory EX2 of the course repository. You need to add the relevant derivation rules and AST constructors. COMPILATION/EX2 should contain a makefile building your source files to a runnable jar file called PARSER (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.

Precedence	Operator	Description	Associativity
1	<code>:=</code>	assign	right
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>-&gt;</code>	field access	left

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

**Execution parameters**    `PARSER` receives 2 input file names:

`InputRioMareProgram.txt`

`OutputStatus.txt`