Exercise 2

Due 8/12/2022, before 23:59

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:
When the input program has correct syntax:
OK

When there is a syntax error:
ERROR(*location*)
(where *location* is the line number of the *first* error that was encountered.)

When there is a lexical error:
ERROR

6 Submission Guidelines

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

- ex2/Makefile

This makefile will 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 intereset 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 run the skeleton, you might need to install the following packages: `$ sudo apt-get install graphviz eog`

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)

7 Notes

- The terminals in the grammar are defined according to the token dfinitions of the first exercise. Note the additional token **TYPE_VOID**.
- The skeleton creates an image file of the AST only for debugging, this should not be included in the final submission.
- The directories *ex2/input* and *ex2/expected_output* contain input tests and their corresponding expected results.

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

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

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 2: Binary operators of **L** along with their associativity and precedence. 1 stands for the lowest precedence, and 8 for the highest.