Exercise 2

Compilation 0368:3133

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 **??** 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 **??**.

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 Running the skeleton

You can run the skeleton by running the following command (from the directory of the execrcise):

- make

The makefile generates the relevant files using jflex/cup, compiles everything, runs, and generates an image of the resulting syntax tree.

7 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. The exercise skeleton can be found (along with the other files...) in the course repository using the following link:
`https://github.com/davidtr1037/COMPILATION_TAU_FOR_STUDENTS/tree/master/FOLDER_3_SOURCE_CODE/EX2`

Execution parameters PARSER receives 2 input file names:

InputPicassoProgram.txt
OutputStatus.txt

8 Additional Notes

Note that in this exercise, the token for integers (INT) excludes '+' and '-', so you might need to update your lexer according to that. Now, if the input for the lexer is *-2*, then it should return 2 tokens: MINUS INT.

Program	::=	dec^+
dec	::=	varDec | funcDec | classDec | arrayDec
varDec	::=	ID ID [ASSIGN exp] ';'
	::=	ID ID ASSIGN newExp ';'
funcDec	::=	ID ID $'('$ [ID ID [',' ID ID]*] $')'$ $'\{'$ stmt [stmt]* $'\}'$
classDec	::=	CLASS ID [EXTENDS ID] $'\{'$ cField [cField]* $'\}'$
arrayDec	::=	ARRAY ID = ID $'['$ $']'$
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 ID | NEW ID $'['$ 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 9 for the highest.