

Exercise 2

Compilation 0368-3133

Due 19/12/2024, 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 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**, where the start symbol is program. Note that regular expression syntax is used to define the grammar. Since CUP does not support this syntax, you will need to modify the given grammar to only use standard context free grammar notation. 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.)

program	→	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 stmt (stmt)* RBRACE
classDec	→	CLASS ID (EXTENDS ID)? LBRACE cField (cField)* RBRACE
arrayTypedef	→	ARRAY ID EQ type LBRACK RBRACK SEMICOLON
exp	→	var LPAREN exp RPAREN exp BINOP exp (var DOT)? ID LPAREN (exp (COMMA exp)*)? RPAREN (MINUS)? INT NIL STRING
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 stmt (stmt)* RBRACE WHILE LPAREN exp RPAREN LBRACE stmt (stmt)* RBRACE (var DOT)? ID LPAREN (exp (COMMA exp)*)? RPAREN SEMICOLON
newExp	→	NEW type NEW type LBRACK exp RBRACK
cField	→	varDec funcDec
BINOP	→	PLUS MINUS TIMES DIVIDE LT GT EQ

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

When there is a lexical error:

ERROR

6 Submission Guidelines

The skeleton for this exercise can be found here. You need to add the relevant derivation rules to the CUP file and AST constructors in your project repository. 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)

Precedence	Operator	Description	Associativity
1	<code>:=</code>	assign	
2	<code>=</code>	equals	left
3	<code><, ></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: Operators of **L** along with their associativity and precedence. 1 stands for the lowest precedence, and 8 for the highest.

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* (classes for the AST nodes)

To run the skeleton, you might need to install the following packages (this is only required for generating an image of the AST):

```
$ 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 definitions of the first exercise.
- 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.