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 summarizes the context free grammar of RioMare. You will need to feed this grammar to CUP, and make sure there are no shift-reduce conflicts.

To create a graph visualization of the AST, please install graphviz and run

```
$ dot -Tjpeg -o ./AST_Graph.jpeg ./AST_Graph.txt
```

from `EX5/LINUX_GCC_MAKE`

Program	::=	dec^+
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.

4 Input

The input for this exercise is a single text file, the input RioMare program.

5 Output

The output is a single text file that contains a tokenized representation of the input program. Each token should appear in a separate line, together with the line number it appeared on, and the character position inside that line. The list of token names appears in Table 777, and will only be used in this first exercise. Later phases of the compiler will make no use of these token names.

Three types of tokens are associated with corresponding values: integers, identifiers and strings. The printing format for these tokens can be easily deduced from the examples in Table 5.

int	string	void	while
class	nil	new	if
PrintInt	extends	return	end
PrintString			

Table 2: Reserved keywords of RioMare.

Precedence	Operator	Description	Associativity
1	:=	assign	right
2	=	equals	left
3	$<, \leq, >, \geq$		left
4	$+, -$		left
5	$*, /$		left
6	[	array indexing	
7	(	function call	
8	->	field access	left

Table 3: Binary operators of RioMare along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.

6 Submission Guidelines

The code for this exercise resides as usual in subdirectory EX5 of the course GitHub. Currently, the grammar in RioMare.y contains a shift/reduce conflict, so you should start by fixing this. Next, you need to add the relevant derivation rules and AST constructors for classes. Last, you should implement the missing parts of the RioMare semantic analyzer. The semantic analyzer resides in the file semant.c, and this is where most of your changes will occur. Please submit your exercise in your GitHub repository under COMPILATION/EX5, and have a makefile there to build a runnable program called compiler. Make sure that compiler is created in the same level as the makefile: inside EX5. To avoid the pollution of EX5, please remove all *.o files once the target is built. The next paragraph describes the execution of compiler.

Execution parameters compiler receives 2 input file names:

InputRioMareProgram.txt
OutputStatus.txt

Token Name	Description		Token Name	Description
LPAREN	(		ASSIGN	:=
RPAREN	)		EQ	=
LBRACK	[		LT	<
RBRACK	]		GT	>
LBRACE	{		CLASS	
RBRACE	}		EXTENDS	
PLUS	+		RETURN	
MINUS	−		WHILE	
TIMES	∗		IF	
DIVIDE	/			
COMMA	,		INT(*value*)	*value* is an integer
DOT	.		STRING(*value*)	*value* is a string
SEMICOLON	;		ID(*value*)	*value* is an identifier

Table 4: Token names and printing format for the first exercise. Each line in the output text file should contain a single token. Note that three types of tokens are associated with corresponding values: integers, identifiers and strings. The rest of the tokens encountered should only contain their name, the line number they appeared on, and the character position inside that line.

Printed Lines Examples	Description
INT(74)[3,8]	integer 74 is encountered in line 3, character position 8
STRING(“Dan”)[2,5]	string “Dan” is encountered in line 2, character position 5
ID(numPts)[1,6]	identifier numPts is encountered in line 1, character position 6

Table 5: Token names and printing format for the first exercise. Each line in the output text file should contain a single token. Note that three types of tokens are associated with corresponding values: integers, identifiers and strings. The rest of the tokens encountered should only contain their name, the line number they appeared on, and the character position inside that line.