Department of Computer Science
Technical University of Cluj-Napoca

Limbaje formale şi translatoare

Laboratory activity

Name: Vijoli Ioana, Zagan Bogdan
Group: 30232
Email: ioanavijoli@yahoo.com

Contents

1 Introducere **3**
1.1 Problema aleasa 3

2 Descriere teoretica **4**

3 Decizii de implementare **5**

4 Rulare **6**

5 GitHub **8**

Chapter 1

Introducere

1.1 Problema aleasa

In cadrul acestui assignment am ales să realizăm un interpretor al unui subset de instrucţiuni din limbajul de programare Java. Pentru a rezolva această cerinţă, am folosit Lex şi Yacc. După implementarea interpretorului, am implementat de asemenea un arbore de parsare, în două fişiere separate, tree.h şi tree.c.

Chapter 2

Descriere teoretica

Cele mai importante două fișiere din acest proiect sunt: un fișier .lex și un fișier .yacc (sau .y). Acestea sunt utilizate pentru a construi un analizor sintactic și un analizor lexical pentru un limbajul de programare Java.

Fișierul .lex conține reguli de analiză lexicală scrise într-un limbaj numit Lex. Aceste reguli sunt utilizate pentru a recunoaște și a separa diverse cuvinte cheie, identificatori, constante și simboluri din codul nostru sursă pe care îl vom introduce ca input. Fiecare regulă este formată dintr-un șablon și o acțiune asociată. Când un șablon se potrivește cu un token în codul sursă, se execută acțiunea corespunzătoare. De exemplu, regulile pot recunoaște cuvinte cheie precum "class", "if", "while" sau constante precum numere întregi sau identificatori. Acțiunile pot crea obiecte de tipul "tree-node", care va reprezenta arborele nostru de parsare. Prin intermediul variabilei lineNumber vom contoriza linia curentă care ne va ajuta ulterior să afişăm arborele.

Fișierul .yacc conține reguli de analiză sintactică scrise în limbajul numit Yacc. Aceste reguli sunt utilizate pentru a defini structura sintactică a limbajului de programare. Aceste reguli sunt organizate într-un set de producții care specifică cum sunt combinate și organizate diferitele elemente ale limbajului. Fiecare producție are o parte stângă și o parte dreaptă, care descrie cum sunt combinate elementele și ce rezultă.

În cazul nostru, producțiile definesc structura unei clase, a unei declarații de variabilă sau a unei declarații de metodă. Fiecare producție are asociată o acțiune care se execută atunci când producția este recunoscută. Aceste acțiuni creează arborele de parsare, care va fi ulterior afişat folosind metoda pretty print din fisierul tree.c.

În ansamblu, aceste două fișiere formează un sistem de analiză lexicală și sintactică pentru a parsa și a construi un arbore sintactic al codului sursă al limbajului de programare specificat.

Chapter 3

Decizii de implementare

Deoarece sintaxa limbajului Java este una extrem de complexă, am ales ca proiectul nostru să poată recunoaşte un număr limitat de instrucţiuni şi cuvinte cheie, pentru a demostra cum funcţionează un astfel de interpretor. În implementare ne-am folosit de modelul de interpretor C care exista pe moodle, dar şi de resursele pe care le aveam de la laborator.

Astfel, am ales să recunoaştem instrucţiunile de if, respecti if, else şi while, atât în cazurile în care acestea apar simple, cât şi dacă sunt imbricate. Totodată vom recunoaşte diverse semne matematice de bază, variabilele, constantele şi bineînţeles clasele, modificatorii de acces de tipul private sau public, specifici limbajului Java. Vom recunoaşte de asemenea şi diferite tipuri de date precum String, boolean şi Integer, chiar dacă în limbajul Java sunt mult mai multe şi nu le vom putea trata pe toate.

Pentru a reprezenta eficient structura sintactică a codului sursă, am decis să utilizăm un arbore de parsare, în care fiecare nod poate avea mai mulţi copii. Fiecare nod al arborelui reprezintă o producție din gramatica limbajului de programare. Această structură permite reprezentarea ierarhică a codului sursă și permite extinderea facilă a gramaticii și a analizei sintactice.

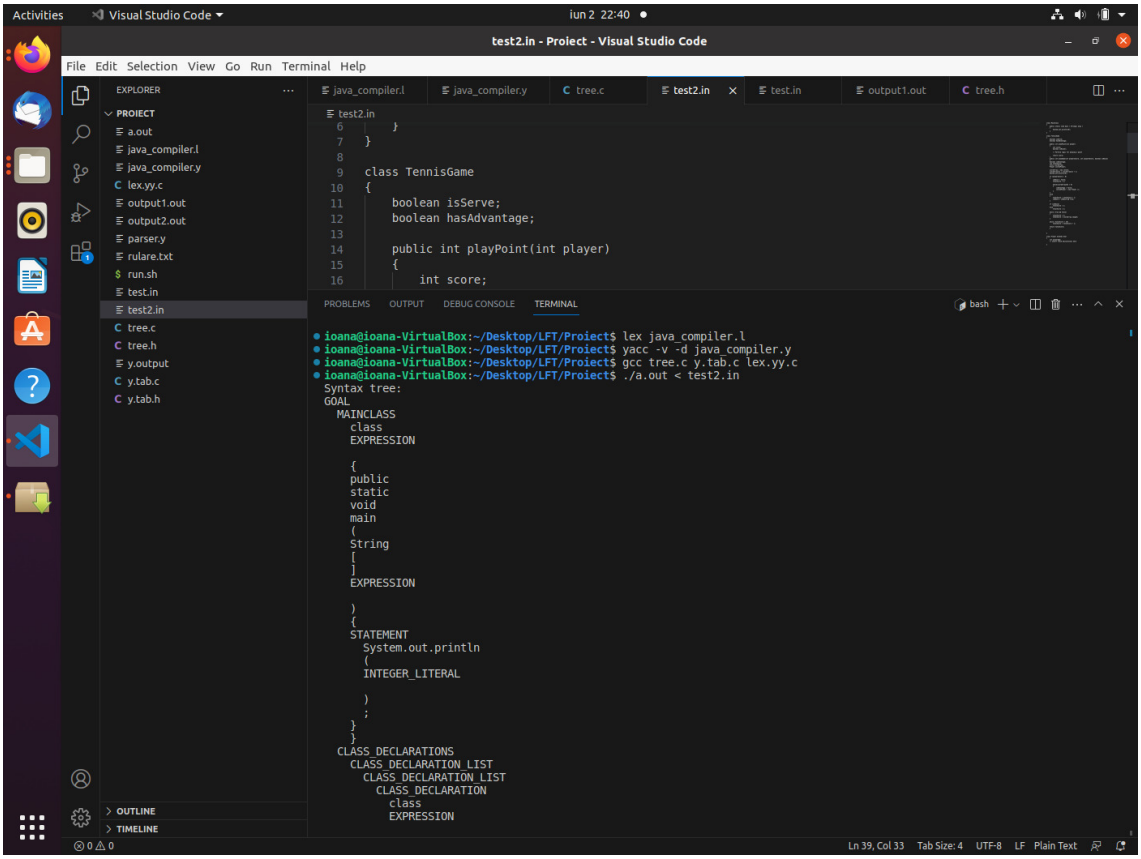
Pentru rezolvarea conflictelor de Shift-Reduce care au apărut pe parcursul implementării, a fost nevoie să specificăm tipul de asociativitate dorit pentru anumiţi tokeni. Astfel, am ales ca acoladele, semnele de mai mic, mai mare, egal să aiba tipul de asociativitate left, pe când operatorul de negaţie ! să fie asociativ dreapta.

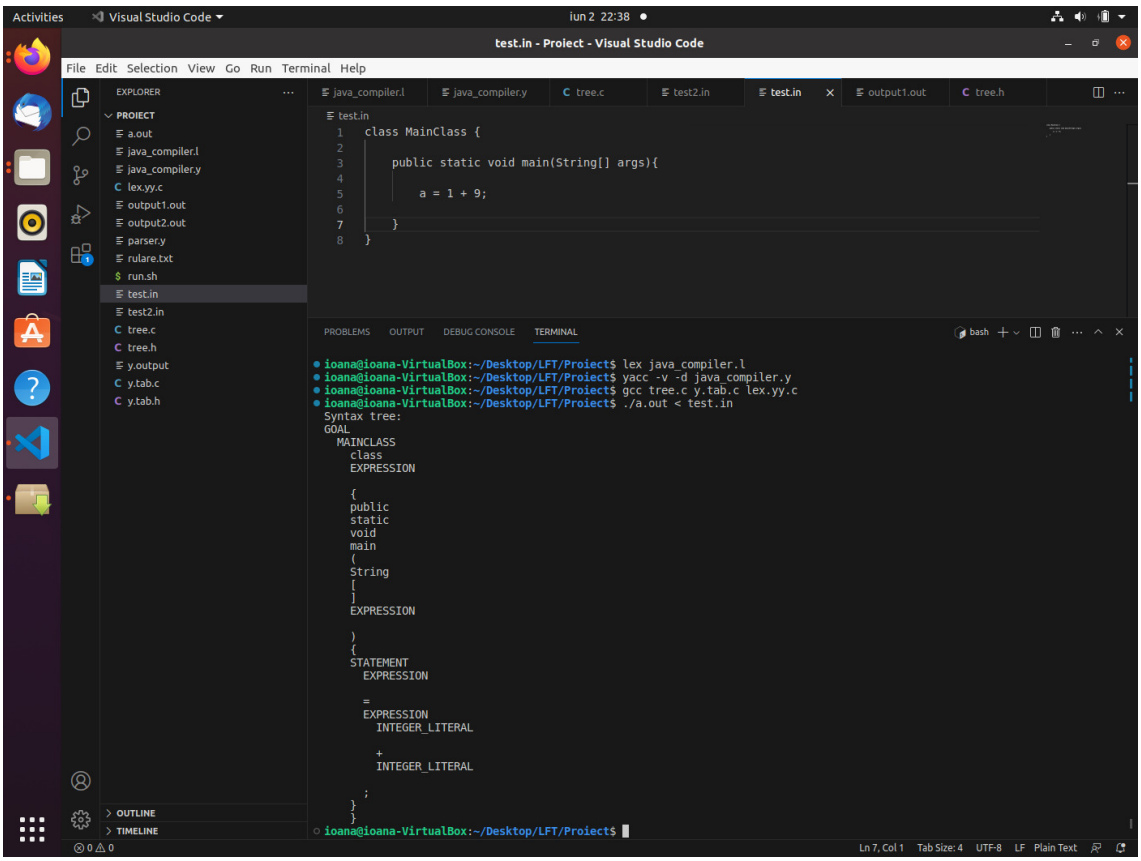
Chapter 4

Rulare

Pentru rulare se folosesc comenzile din urmatoarele 2 imagini. Pentru test am ales 2 fisiere, care contin 2 coduri de Java.

Fişierul test.in conţine cod cu o sintaxă mai simplă, pe când textul din test2.in are o sintaxă mai complicată, deci şi un arbore de parsare de adâncime mai mare.




Activities
Visual Studio Code
iun 2 22:38
test.in - Proiect - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
PROIECT
a.out
java_compiler.l
java_compiler.y
lex.yy.c
output1.out
output2.out
parser.y
rulare.txt
run.sh
test.in
test2.in
tree.c
tree.h
y.output
y.tab.c
y.tab.h
OUTLINE
TIMELINE
java_compiler.l
java_compiler.y
tree.c
test2.in
test.in
output1.out
tree.h
test.in
1 class MainClass {
2
3 public static void main(String[] args){
4
5 a = 1 + 9;
6
7 }
8 }
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
bash
ioana@ioana-VirtualBox:~/Desktop/LFT/Proiect$ lex java_compiler.l
ioana@ioana-VirtualBox:~/Desktop/LFT/Proiect$ yacc -v -d java_compiler.y
ioana@ioana-VirtualBox:~/Desktop/LFT/Proiect$ gcc tree.c y.tab.c lex.yy.c
ioana@ioana-VirtualBox:~/Desktop/LFT/Proiect$ ./a.out < test.in
Syntax tree:
GOAL
MAINCLASS
class
EXPRESSION
{
public
static
void
main
(
String
[
]
EXPRESSION
)
{
STATEMENT
EXPRESSION
=
EXPRESSION
INTEGER_LITERAL
+
INTEGER_LITERAL
;
}
}
ioana@ioana-VirtualBox:~/Desktop/LFT/Proiect$
0 0
Ln 7, Col 1
Tab Size: 4
UTF-8
LF
Plain Text

Chapter 5

GitHub

https://github.com/ioanavijoli/Interpretor-Java