

Національний технічний університет України

«Київський Політехнічний Інститут»

Факультет прикладної математики

Кафедра системного програмування та спеціалізованих комп’ютерних систем

**Лабораторна робота № 3**

**З дисципліни «Інженерія програмного забезпечення. Основи проектування трансляторів»**

**На тему: «Розробка генератора коду»**

**Виконав:**

**Студент групи КВ-31**

**Черниш Андрій**

**Номер у списку групи : 23**

**Київ 2016**

**Постановка завдання**

Розробити програму генератора коду для підмножини мови програмування SIGNAL.

Програма має забезпечувати:

* читання дерева розбору та таблиць, створених синтаксичним аналізатором, який було розроблено в розрахунково-графічній роботі;
* виявлення семантичних помилок;
* генерацію коду та/або побудову внутрішніх таблиць для генерації коду.

**Входом** ГК мають бути:

* дерево розбору;
* таблиці ідентифікаторів та констант з повною інформацією, необхідною для генерації коду;
* вхідна програма на підмножині мови програмування SIGNAL згідно з варіантом (необхідна для формування лістингу програми).

**Виходом** ГК мають бути:

* асемблерний код згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;
* внутрішні таблиці генератора коду (якщо потрібні);

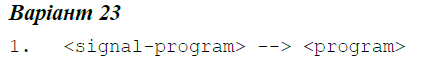
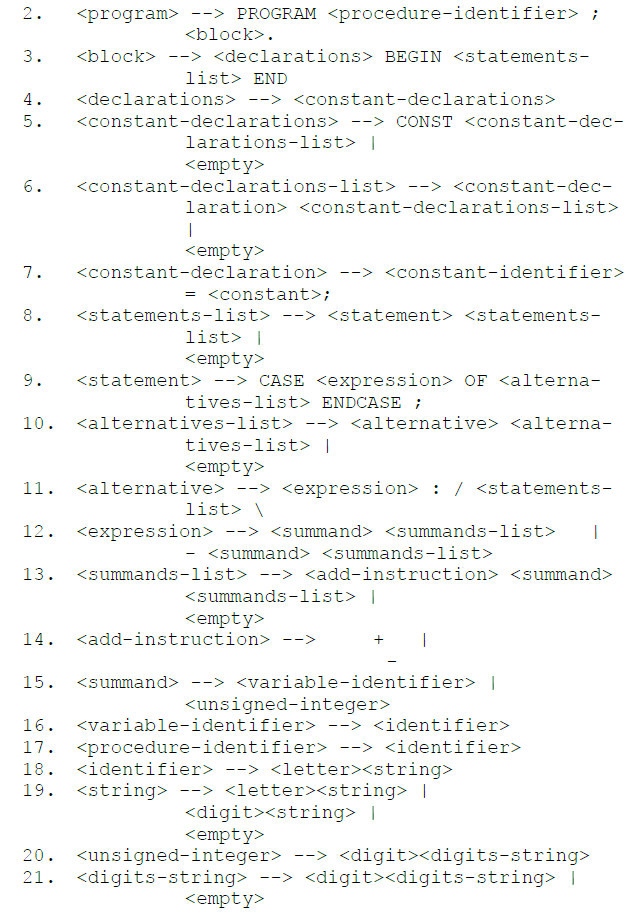
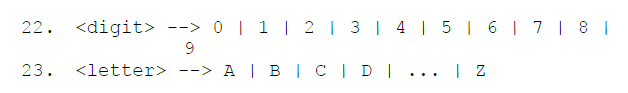
**Зкомпонувати повний компілятор, що складається з розроблених раніше лексичного та синтаксичного аналізаторів і генератора коду, який забезпечує наступне:**

* генерацію коду та/або побудову внутрішніх таблиць для генерації коду;
* формування лістингу вхідної програми з повідомленнями про лексичні, синтаксичні та семантичні помилки.

**Входом** компілятора має бути програма на підмножині мови програмування SIGNAL згідно з варіантом;

**Виходом** компілятора мають бути:

* асемблерний код згенерований для вхідної програми та/або внутрішні таблиці для генерації коду;



Код програми

**CodeGenerator.java**

**package** com.itcherry.translators;  
  
**import** javax.swing.tree.DefaultMutableTreeNode;  
**import** java.io.\*;  
**import** java.util.LinkedList;  
**import** java.util.Stack;  
  
*/\*\*  
 \* Class, which describe algorithm of code generating of signal program  
 \* We recursively go down on a tree and make assembler text  
 \*/***public class** CodeGenerator {  
 **private** PrintWriter **out**; *// To write asm code to the file* **private** StringBuffer **asmTextString**; *// To write asm code to the frame* **private** String **bf**; *// Buffer for variable* **private** String **zn**; *// Buffer for arithmetical operation* **private** String **reg**; *// Buffer for different register* **private boolean alternativeListFlag** = **false**; *//boolean flag for CASE* **private int labelsCounter** = 0;  
 **private** Stack<LinkedList<Integer>> **labelsStack**;  
 **private** LinkedList<Integer> **labelsArray**;  
 **private** Stack<Integer> **endcaseLabelsStack**;  
 **private int endcaseLabelsCounter** = 1;  
  
 **private** DefaultMutableTreeNode **node**;  
  
 */\*\*  
 \*  
 \** ***@param node*** *- root of the tree  
 \*/* **public** CodeGenerator(DefaultMutableTreeNode node) {  
 **this**.**node** = node;  
 **labelsStack** = **new** Stack<>();  
 **endcaseLabelsStack** = **new** Stack<>();  
 **asmTextString** = **new** StringBuffer();  
 **labelsCounter** = 0;  
 }  
  
 */\*\*  
 \* Simple function for main  
 \*/* **public void** generate(){  
 spr(**node**.getNextNode());  
 }  
  
 */\*\*  
 \* Writes string to the file ant to the frame  
 \** ***@param str*** *- string, which will write to a file and to a frame  
 \*/* **private void** write(String str){  
 **out**.append(str);  
 **asmTextString**.append(str);  
 }  
  
 */\*\*  
 \* Basic procedure, which we call recursively  
 \* It generates a .asm code  
 \** ***@param node*** *- root of the tree  
 \*/* **private void** spr(DefaultMutableTreeNode node){  
 DefaultMutableTreeNode tempNode = node;  
 String temp = (String) node.getUserObject();  
 **int** ruleNumber = Integer.*parseInt*(temp.substring(temp.indexOf(**'>'**) + 1, temp.length()).trim());  
 **switch** (ruleNumber) {  
 **case** 1: *//<signal-program> --> <program>* spr(tempNode.getNextNode());  
 **break**;  
 **case** 2: *//<program> --> PROGRAM <procedure-identifier> ; <block> .* tempNode = tempNode.getNextNode().getNextNode();  
 spr(tempNode);  
 spr(tempNode.getNextSibling().getNextSibling());  
 write(**"END START\n"**);  
 **out**.close();  
 **break**;  
 **case** 17: *// <procedure-identifier> --> <identifier>* **try** {  
 File file = **new** File(**"src/res/"** + tempNode.getNextNode().getUserObject());  
 **if**(file.exists()) file.delete();  
 FileWriter fw = **new** FileWriter(file, **true**);  
 BufferedWriter bw = **new** BufferedWriter(fw);  
 **out** = **new** PrintWriter(bw);  
 }**catch** (IOException e){  
 System.***out***.println(**"IO Exception in .asm file creating!!!"**);  
 }  
 **break**;  
 **case** 3: *// <block> --> <declarations> BEGIN <statements-list> END* tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 write(**"CODE SEGMENT\n\nSTART:\n"**);  
 spr(tempNode.getNextSibling().getNextSibling());  
 write(**"CODE ENDS\n"**);  
 **break**;  
 **case** 4: *// <declarations> --> <constant-declarations>* spr(tempNode.getNextNode());  
 **break**;  
 **case** 5: *// <constant-declarations> --> CONST <constant-declarations-list> | <empty>* **if**(tempNode.getChildCount() == 0) **return**;  
 tempNode = tempNode.getNextNode();  
 write(**"DATA SEGMENT\n"**);  
 spr(tempNode.getNextSibling());  
 write(**"DATA ENDS\n"**);  
 **break**;  
 **case** 6: *//<constant-declarations-list> --> <constant-declaration><constant-declarations-list> | <empty>* **if**(tempNode.getChildCount() == 0) **return**;  
 tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 spr(tempNode.getNextSibling());  
 **break**;  
 **case** 7: *// <constant-declaration> --> <constant-identifier> = <constant>* tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 write(**"\t "** + **bf** + **"\tdw\t"**);  
 spr(tempNode.getNextSibling().getNextSibling());  
 write(**bf** + **"\n"**);  
 **break**;  
 **case** 8: *// <statements-list> --> <statement><statements-list> | <empty>* **if**(tempNode.getChildCount() == 0) {  
 write(**"\t\tNOP\n"**);  
 **return**;  
 }  
 tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 spr(tempNode.getNextSibling());  
 **break**;  
 **case** 9: *//<statement> --> CASE <expression> OF <alternatives-list> ENDCASE ;* tempNode = tempNode.getNextNode().getNextSibling();  
 **labelsArray** = **new** LinkedList<>();  
 **endcaseLabelsStack**.push(**endcaseLabelsCounter**++);  
  
 **reg** = **"AX"**;  
 spr(tempNode);  
  
 **reg** = **"BX"**;  
 **alternativeListFlag** = **false**;  
 spr(tempNode.getNextSibling().getNextSibling());  
  
 **labelsStack**.push(**labelsArray**);  
 **alternativeListFlag** = **true**;  
 spr(tempNode.getNextSibling().getNextSibling());  
 write(**"?E"** + (**endcaseLabelsStack**.pop()) + **":\tNOP\n"**);  
 **labelsArray** = **labelsStack**.pop();  
 **if**(**labelsArray**.isEmpty() && !**labelsStack**.isEmpty()) **labelsArray** = **labelsStack**.peek();  
 **break**;  
 **case** 10:*// <alternatives-list> --> <alternative><alternatives-list> | <empty>* **if**(tempNode.getChildCount() == 0) **return**;  
 tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 **if**(**alternativeListFlag** && tempNode.getNextSibling().getChildCount()!=0)  
 write(**"\t\tJMP ?E"** + **endcaseLabelsStack**.peek() + **"\n"**);  
 spr(tempNode.getNextSibling());  
 **break**;  
 **case** 11: *// <alternative>--> <expression>:/<statements-list>\* tempNode = tempNode.getNextNode();  
 **if**(!**alternativeListFlag**) {  
 spr(tempNode);  
 write(**"\t\tCMP AX,BX\n\t\tJE ?L"** + (++**labelsCounter**) + **"\n"**);  
 **labelsArray**.add(**labelsCounter**);  
 }**else**{  
 write(**"?L"** + **labelsArray**.pop() + **":\n"**);  
 spr(tempNode.getNextSibling().getNextSibling().getNextSibling());  
 }  
 **break**;  
 **case** 12: *//<expression> --> <summand><summands-list> | - <summand><summands-list>* tempNode = tempNode.getNextNode();  
 **if**(tempNode.getUserObject().equals(**"-"**)){  
 spr(tempNode.getNextSibling());  
 write(**"\t\tMOV "** + **reg** + **","** + **bf** + **"\n\t\tNOT "** + **reg** + **"\n\t\tINC "** + **reg** + **"\n"**);  
 spr(tempNode.getNextSibling().getNextSibling());  
 }**else**{  
 spr(tempNode);  
 write(**"\t\tMOV "** + **reg** + **","** + **bf** + **"\n"**);  
 spr(tempNode.getNextSibling());  
 }  
 **break**;  
 **case** 13: *// <summands-list> --> <add-instruction><summand><summands-list> | <empty>* **if**(tempNode.getChildCount() == 0) **return**;  
 tempNode = tempNode.getNextNode();  
 spr(tempNode);  
 spr(tempNode.getNextSibling());  
 **if**(**zn**.equals(**"+"**)) write(**"\t\tADD "** + **reg** + **","** + **bf** + **"\n"**);  
 **else** write(**"\t\tSUB "** + **reg** + **","** + **bf** + **"\n"**);  
 spr(tempNode.getNextSibling().getNextSibling());  
 **break**;  
 **case** 14: *// <add-instruction> --> + | -* **zn** = (String) tempNode.getNextNode().getUserObject();  
 **break**;  
 **case** 15: *// <summand> --> <variable-identifier> | <unsigned-integer>* spr(tempNode.getNextNode());  
 **break**;  
 **case** 16: *// <variable-identifier> --> <identifier>* **case** 18: *// <constant-identfier> --> <identifier>* **case** 20: *// <unsigned-integer> --> <digit><digits-string>* **bf** = (String) tempNode.getNextNode().getUserObject();  
 **break**;  
 }  
 }  
  
 */\*\*  
 \** ***@return*** *result of the code generator  
 \*/* **public** StringBuffer getAsmTextString() {  
 **return asmTextString**;  
 }  
  
}

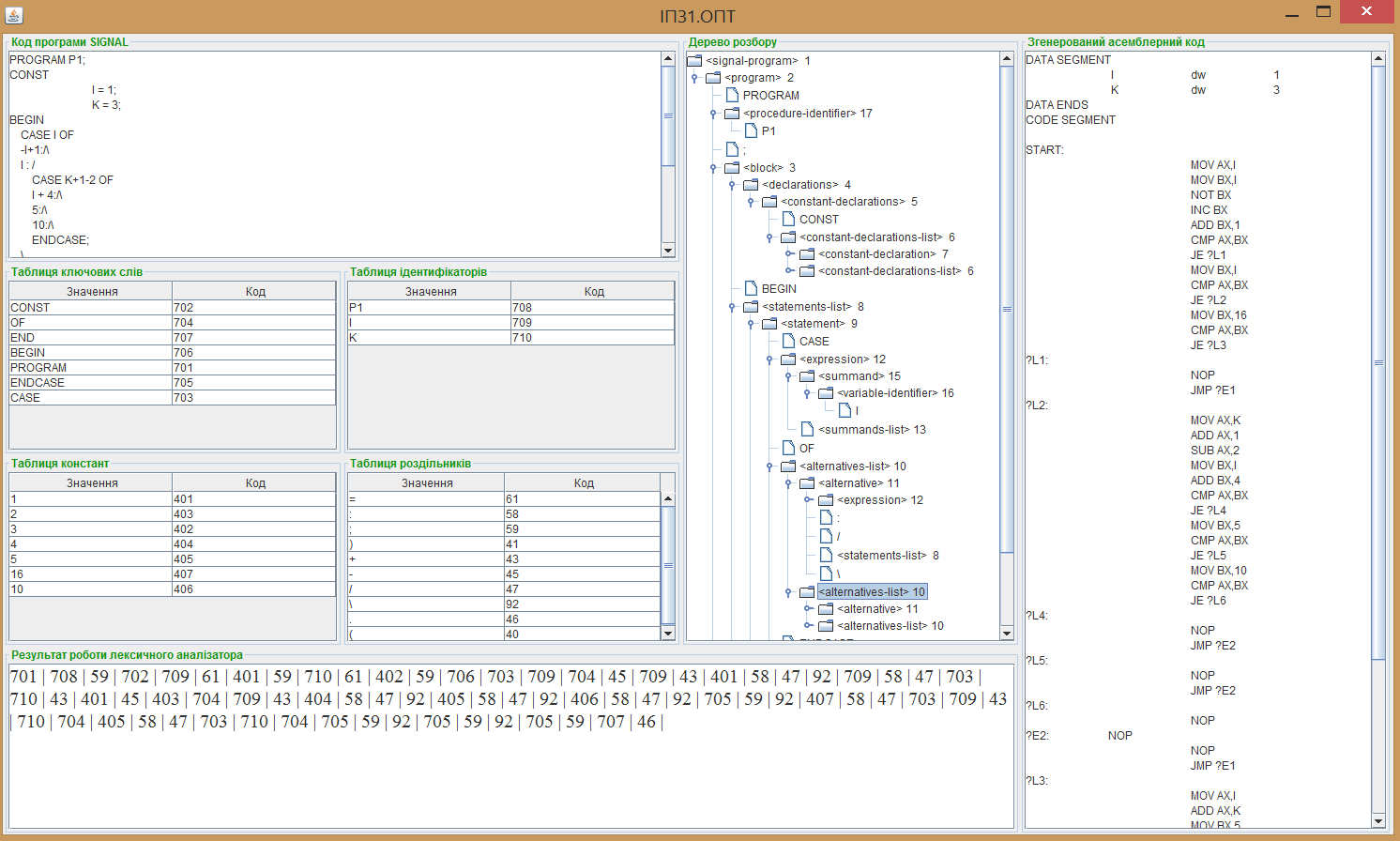
**Main.java**

**package** com.itcherry.translators;  
  
**import** javax.swing.tree.DefaultTreeModel;  
**import** java.io.IOException;  
  
**public class** Main {  
  
 **public static void** main(String[] args) **throws** IOException {  
 LexicalAnalyser lexicalAnalyser = **new** LexicalAnalyser();  
 lexicalAnalyser.analyse();  
  
 SyntaxAnalyser syntaxAnalyser = **new** SyntaxAnalyser(lexicalAnalyser);  
 DefaultTreeModel tree = syntaxAnalyser.analyse();  
  
 CodeGenerator codeGenerator = **new** CodeGenerator(syntaxAnalyser.getResult());  
 codeGenerator.generate();  
  
 GUI frame = lexicalAnalyser.getFrame();  
 frame.initTable(lexicalAnalyser.**keywords**,  
 lexicalAnalyser.**identificators**,  
 lexicalAnalyser.**constants**,  
 lexicalAnalyser.getResult(),  
 lexicalAnalyser.getSignalText(),  
 tree,  
 codeGenerator.getAsmTextString());  
 frame.drawFrame();

**Контрольні приклади**

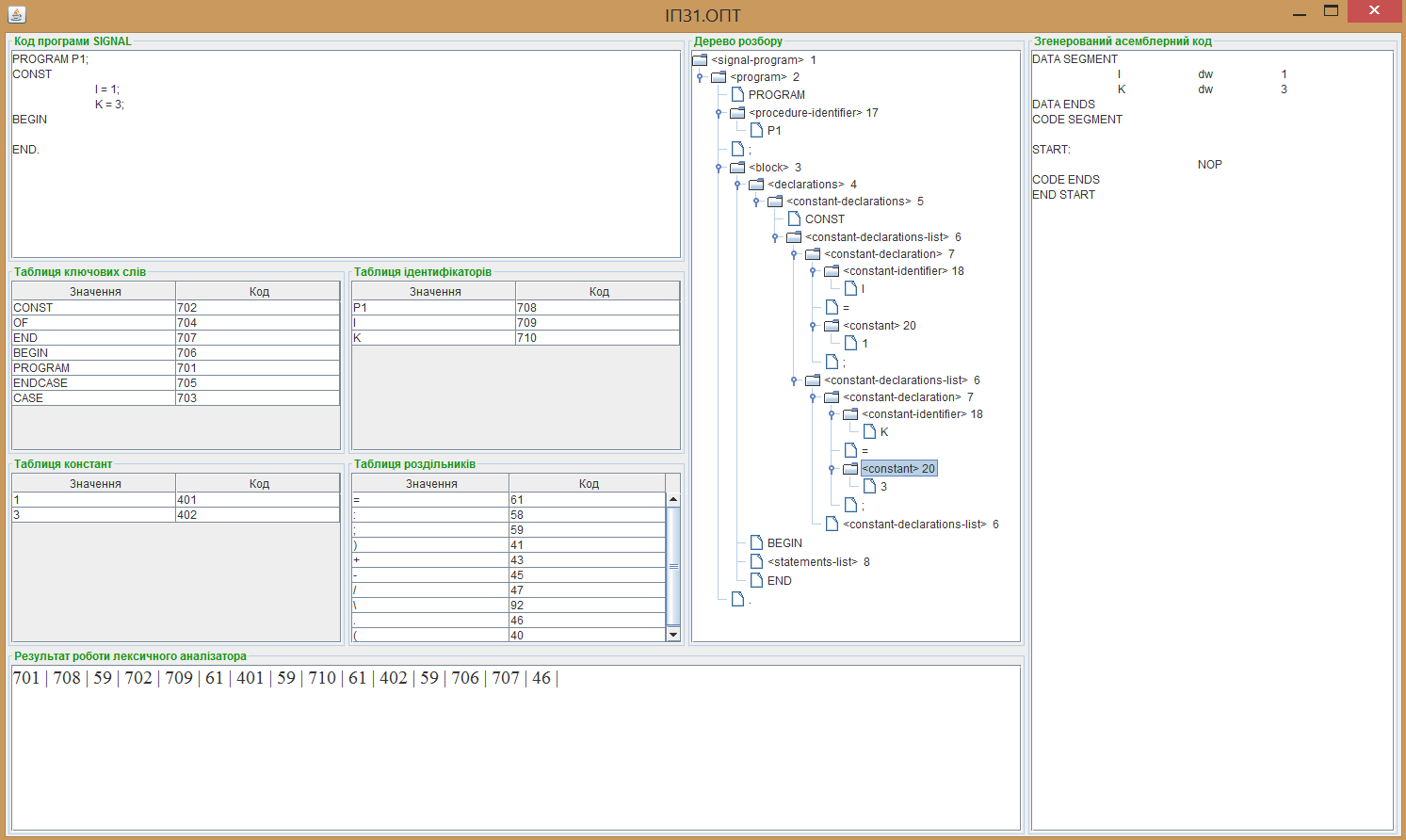
**Приклад 1 :**

PROGRAM P1;  
CONST  
 I = 1;  
 K = 3;  
BEGIN  
 CASE I OF  
 -I+1:/\  
 I : /  
 CASE K+1-2 OF  
 I + 4:/\  
 5:/\  
 10:/\  
 ENDCASE;  
 \  
 16 :/  
 CASE I+K OF  
 5:/  
 CASE K OF  
 ENDCASE;  
 \  
 ENDCASE;  
 \  
 ENDCASE;  
END.



**Приклад 2 :**

PROGRAM P1;  
CONST  
 I = 1;  
 K = 3;  
BEGIN  
  
END.

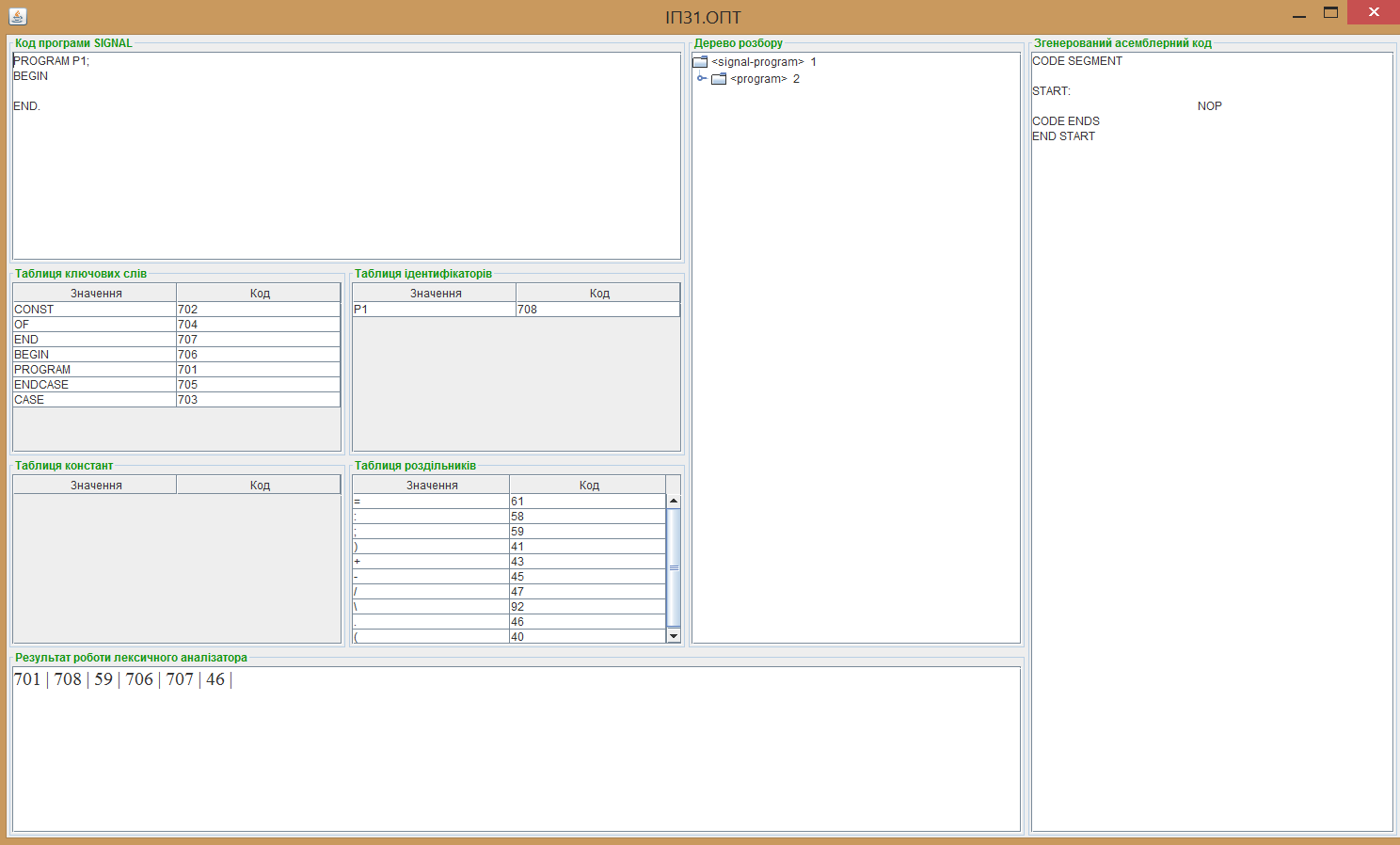


**Приклад 2 :**

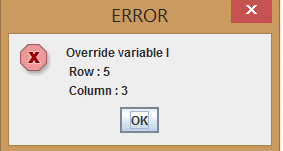
PROGRAM ProG1;

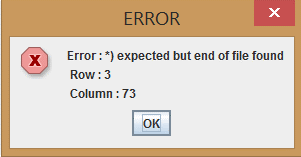
BEGIN

END.

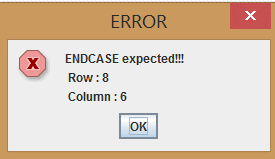


**Приклад 3 :**

PROGRAM P1;  
CONST  
 I = 1;  
 K = 3;  
 I = 4;  
BEGIN  
 CASE K OF  
 3(\*fsdfsdfg\*\*\*)+K: /\  
 ENDCASE;  
END.

**Приклад 4:**

PROGRAM P1;  
CONST  
 (\*I = 1;  
 K = 3;  
BEGIN  
 CASE K OF  
 3+K: /\  
 ENDCASE;  
END.

**Приклад 5:**

PROGRAM P1;  
CONST  
 I = 1;  
 K = 3;  
BEGIN  
 CASE K OF  
 3+K: /\  
 ;  
END.