МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО”

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА «РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА»

Виконав: Ковкін В.В.
Студент групи КВ-22

Перевірив(ла):_____

Київ-2025

Мета розрахунково-графічної роботи

Метою розрахунково-графічної роботи «Розробка синтаксисного аналізатора» є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки синтаксичних аналізаторів (парсерів).

Постановка задачі

1. Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL згідно граматики за варіантом.
2. Програма має забезпечувати наступне:

- читання рядка лексем та таблиць, згенерованих лексичним аналізатором, який було розроблено в лабораторній роботі «Розробка лексичного аналізатора»;
- синтаксичний аналіз (розбір) програми, поданої рядком лексем (алгоритм синтаксичного аналізатора вибирається за варіантом);
- побудову дерева розбору;
- формування таблиць ідентифікаторів та різних констант з повною інформацією, необхідною для генерування коду;
- формування лістингу вхідної програми з повідомленнями про

Варіант №10

Граматика за варіантом:

1. <signal-program> --> <program>
2. <program> --> PROGRAM <procedure-identifier> ;

<block> ;

3. <block> --> <declarations> BEGIN <statements-list> END
4. <statements-list> --> <empty>
5. <declarations> --> <procedure-declarations>
6. <procedure-declarations> --> <procedure> <procedure-declarations> |

<empty>

7. <procedure> --> PROCEDURE <procedure-identifier><parameters-list> ;
8. <parameters-list> --> (<declarations-list>) |

<empty>

9. <declarations-list> --> <declaration> <declarations-list> |

<empty>

10. <declaration> --><variable-identifier><identifiers-list>:<attribute><attributes-list> ;
11. <identifiers-list> --> , <variable-identifier>

<identifiers-list> |

<empty>

12. <attributes-list> --> <attribute> <attributes-list> |

<empty>

13. <attribute> --> SIGNAL |

COMPLEX |

INTEGER |

FLOAT | BLOCKFLOAT | EXT

14. <variable-identifier> --> <identifier>
15. <procedure-identifier> --> <identifier>
16. <identifier> --> <letter><string>
17. <string> --> <letter><string> |

<digit><string> |

<empty>

18. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
19. <letter> --> A | B | C | D | ... | Z

Код програми

Parser.cpp

```
#include "parser.h"

Parser::Parser(const LexemString &Lexem_String, const Tables &tables)
    : lexems(Lexem_String), tables(tables), i(0)
{

}

Tree *Parser::generate_tree(){
    SCN();
    Tree *result_tree = new Tree(signal_program());
    return result_tree;
}

void Parser::SCN(){
    if(i <  static_cast<int>(lexems.get().size())){
        TS = lexems.get()[i++];
    }
}

Node* Parser::signal_program(){
    Node *signal_program = new Node("<signal-program>");
    signal_program->add_child(program());
    return signal_program;
}

Node* Parser::program(){
    Node *program = new Node("<program>");

    if(TS.code != tables.Keywords.get("PROGRAM")){
        errorLogger.logError("Parser",  TS.nline,  TS.ncol,  "Program  should  start  with
'PROGRAM' keyword");
        program->add_child(new Node("<error>"));
    } else {
        program->add_child(new Node(TS.Lexem));
    }
    SCN();
    program->add_child(procedure_identifier());
```

```
    if(program->get_children().size() == 0){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected program identifier");
        program->add_child(new Node("<error>"));
    }
    SCN();
    if(TS.code != tables.Delimiters.get(";")){
        errorLogger.logError("Parser",  TS.nline,  TS.ncol,  "Expected  ';'  after  program
identifier");
        program->add_child(new Node("<error>"));
        return program;
    } else {
        program->add_child(new Node(TS.Lexem));
    }
    SCN();
    program->add_child(block());

    SCN();
    if(TS.code != tables.Delimiters.get(";")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected ';' after the block");
        program->add_child(new Node("<error>"));
    } else {
        program->add_child(new Node(TS.Lexem));
    }
    return program;
}

Node *Parser::identifier(){
    Node* identifier = new Node("<identifier>");
    if(TS.code != tables.Identifiers.get(TS.Lexem)){
        return nullptr;
    }

    identifier->add_child(new Node(TS.Lexem));
    return identifier;
}

Node *Parser::block(){
    Node *block = new Node("<block>");
    block->add_child(declarations());
    if(TS.code != tables.Keywords.get("BEGIN")){
```

```
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected 'BEGIN' keyword");
        block->add_child(new Node("<error>"));
        return block;
    } else {
        block->add_child(new Node(TS.Lexem));
    }
    SCN();
    block->add_child(statements_list());
    if(TS.code != tables.Keywords.get("END")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected 'END' keyword");
        block->add_child(new Node("<error>"));
    } else {
        block->add_child(new Node(TS.Lexem));
    }
    return block;
}

Node *Parser::declarations(){
    Node* declarations = new Node("<declarations>");
    declarations->add_child(procedure_declarations());
    return declarations;
}

Node *Parser::procedure_declarations() {
    Node *procedure_declarations_node = new Node ("<procedure_declarations>");
    if(TS.code == tables.Keywords.get("BEGIN")){
        procedure_declarations_node->add_child(empty());
        return procedure_declarations_node;
    }
    procedure_declarations_node->add_child(procedure());
    while(TS.code == tables.Delimiters.get(";")){
        SCN();
        if(TS.code != tables.Keywords.get("BEGIN")){
            procedure_declarations_node->add_child(procedure());
        }
    }
    return procedure_declarations_node;
}

Node *Parser::procedure(){
```

```
    Node *procedure_node = new Node ("<procedure>");
    if(TS.code != tables.Keywords.get("PROCEDURE")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected 'PROCEDURE' keyword");
        procedure_node->add_child(new Node("<error>"));
        return procedure_node;
    } else {
        procedure_node->add_child(new Node(TS.Lexem));
    }
    SCN();
    procedure_node->add_child(procedure_identifier());
    SCN();
    procedure_node->add_child(parameters_list());
    if(TS.code != tables.Delimiters.get(";")){
        errorLogger.logError("Parser",  TS.nline,  TS.ncol,  "Expected  ending  ';'  after
procedure declaration");
        procedure_node->add_child(new Node("<error>"));
    } else {
        procedure_node->add_child(new Node(";"));
    }
    return procedure_node;
}

Node *Parser::parameters_list(){
    Node *parameters_list_node = new Node ("<parameters-list>");
    if(TS.code != tables.Delimiters.get("(")){
        parameters_list_node->add_child(empty());
        return parameters_list_node;
    } else {
        parameters_list_node->add_child(new Node(TS.Lexem));
    }
    SCN();
    parameters_list_node->add_child(declarations_list());
    if(TS.code != tables.Delimiters.get(")")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected '(' after parameters
list");
        parameters_list_node->add_child(new Node("<error>"));
    } else {
        parameters_list_node->add_child(new Node(TS.Lexem));
    }
    SCN();
```

```
    return parameters_list_node;
}

Node *Parser::declarations_list(){
    Node *declaration_list_node = new Node ("<declarations-list>");
    if(TS.code == tables.Delimiters.get(")")){
        declaration_list_node->add_child(empty());
        return declaration_list_node;
    }
    declaration_list_node->add_child(declaration());
    while(TS.code == tables.Delimiters.get(";")){
        SCN();
        if(TS.code != tables.Delimiters.get(")")){
            declaration_list_node->add_child(declaration());
        }
    }
    return declaration_list_node;
}

Node *Parser::declaration(){
    Node *declaration_node = new Node ("<declaration>");
    declaration_node->add_child(variable_identifier());
    SCN();
    if(TS.code == tables.Delimiters.get(",")){
        declaration_node->add_child(identifiers_list());
    }
    if(TS.code != tables.Delimiters.get(":")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected ':' between variables
identifiers and attributes");
        declaration_node->add_child(new Node("<error>"));
        return declaration_node;
    } else {
        declaration_node->add_child(new Node(TS.Lexem));
    }
    SCN();
    declaration_node->add_child(attribute());
    SCN();
    if(TS.code == tables.Delimiters.get(",")){
        declaration_node->add_child(attributes_list());
    }
```

```
    if(TS.code != tables.Delimiters.get(";")){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Expected ';' in the end of
declaration");
        declaration_node->add_child(new Node("<error>"));
        return declaration_node;
    } else {
        declaration_node->add_child(new Node (TS.Lexem));
    }
    return declaration_node;
}

Node *Parser::identifiers_list(){
    Node *identifiers_list = new Node ("<identifiers-list>");
    while(TS.code == tables.Delimiters.get(",")){
        identifiers_list->add_child(new Node(TS.Lexem));
        SCN();
        identifiers_list->add_child(variable_identifier());
        SCN();
    }
    return identifiers_list;
}

Node *Parser::variable_identifier(){
    Node* variable_identifier_node = new Node ("<variable-identifier>");
    Node *identifier_node = identifier();
    if(identifier_node == nullptr){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Variable identifier is explected
to be an identifier");
        variable_identifier_node->add_child(new Node("<error>"));
        return variable_identifier_node;
    } else {
        variable_identifier_node->add_child(identifier_node);
    }
    return variable_identifier_node;
}

Node *Parser::attributes_list(){
    Node *attributes_list = new Node ("<attributes-list>");
    while(TS.code == tables.Delimiters.get(",")){
        attributes_list->add_child(new Node(TS.Lexem));
```

```
        SCN();
        attributes_list->add_child(attribute());
        SCN();
    }
    return attributes_list;
}

Node *Parser::attribute(){
    Node* attribute_node = new Node ("<attribute>");
    Node *identifier_node = identifier();
    if(identifier_node == nullptr || TS.code > 1006){
        errorLogger.logError("Parser", TS.nline, TS.ncol, "Attribute name is explected to
be type");
        attribute_node->add_child(new Node("<error>"));
        return attribute_node;
    } else {
        attribute_node->add_child(identifier_node);
    }
    return attribute_node;
}

Node *Parser::procedure_identifier(){
    Node *procedure_identifier = new Node ("<procedure-identifier>");
    Node *identifier_node = identifier();
    if(identifier_node == nullptr){
        errorLogger.logError("Parser",  TS.nline,  TS.ncol,  "Procedure  identifier  is
explected to be an identifier");
        procedure_identifier->add_child(new Node("<error>"));
    } else {
        procedure_identifier->add_child(identifier_node);
    }
    return procedure_identifier;
}

Node *Parser::statements_list(){
    Node *statements_list_node = new Node("<statements-list>");
    statements_list_node->add_child(empty());
    return statements_list_node;
}
```

```
Node *Parser::empty(){
    Node *empty = new Node("<empty>");
    return empty;
}
```

Parser.h

```
#include "LexemString.h"
#include "tree.h"
#include <vector>
#include "Table.h"
#include "Error.h"

class Parser{
private:
    Lexem TS;
    const LexemString& lexems;
    const Tables &tables;
    int i;
public:
    Parser(const LexemString &Lexem_String, const Tables &tables);
    void SCN();
    Tree *generate_tree();
    Node *signal_program();
    Node *program();
    Node *procedure_identifier();
    Node *block();
    Node *declarations();
    Node *statements_list();
    Node *empty();
    Node *procedure_declarations();
    Node *procedure();
    Node *parameters_list();
    Node *declarations_list();
    Node *declaration();
    Node *variable_identifier();
    Node *identifiers_list();
    Node *attribute();
    Node *attributes_list();
    Node *identifier();
};
```

Tree.cpp

```
#include "tree.h"
#include <iostream>

Tree::Tree(Node *root_node)
:root(root_node)
{}

std::string Node::get_value() const {
    return value;
}

void Node::add_child(Node *newNode){
    if(newNode != nullptr){
        children.push_back(newNode);
    }
}

Node::Node(std::string new_value){
    value = new_value;
}

void Node::set_value(std::string new_value){
    value = new_value;
}

const std::vector<Node *> &Node::get_children() const {
    return children;
}

void output_tree_recursive(Node *root, std::string prefix = "", bool isLast = false){
    std::cout << prefix;

    std::cout << (!isLast ? "├──" : "└──" );
    prefix += isLast ? "    " : "│   ";

    std::cout << root->get_value() << std::endl;
    std::vector<Node *> children = root->get_children();
    int children_N = static_cast<int>(children.size());
    for(int i = 0; i < children_N; i++){
```

```
        output_tree_recursive(children[i], prefix, i >= children_N);
    }
}

void Tree::output_tree(){
    output_tree_recursive(root);
}
```

Tree.h

```
#include <vector>
#include <string>

class Node{
   std::vector<Node*> children;
   std::string value;
   public:
   Node(std::string new_value);
   void add_child(Node *newNode);
   void set_value(std::string new_value);
   std::string get_value() const;
   const std::vector<Node*> &get_children() const;
};

class Tree{
   Node* root;
   public:
   Tree(Node *root_node);

   void output_tree();
};
```

main.cpp

```
#include <iostream>
#include <fstream>
#include <error.h>
#include "Lexer.h"
#include "Table.h"
#include "Attributes.h"
#include "LexemString.h"
#include "parser.h"
#include "Error.h"
```

```
int main(int argc, char** argv){

    if(argc != 2){
        std::cerr << "Compilation error: Path of file is required!";
        return 1;
    }

    std::string Test = argv[1];
    Tables LexTable;
    Attributes attributes;
    Lexer lexer(LexTable, attributes);
    lexer.scan(Test);
    LexemString scan_result = lexer.GetScanResult();
    scan_result.Output();
    LexTable.OutputAllTables();
    Parser parser(scan_result, LexTable);
    parser.generate_tree()->output_tree();
    errorLogger.printErrors();


}
```

Тести

1. truetest1.txt містить граматично правильний код з усіма можливими варіантами оголошення функцій. А саме, PROC1 має оголошені procedure-declarations, PROC2 не має оголошених procedure-declarations, PROC3 має пустий declarations-list, PROC4 має декілька declaration всередині declaration-list

```
1  PROGRAM TEST1 ;
2
3  PROCEDURE PROC1 (
4      VAR1, VAR2, VAR3: BLOCKFLOAT, SIGNAL, INTEGER;
5  );
6  PROCEDURE PROC2;
7  PROCEDURE PROC3 (
8  );
9
10 PROCEDURE PROC4 (
11     VAR6, VAR7, VAR8: BLOCKFLOAT, SIGNAL, INTEGER;
12     VAR11, VAR12: SIGNAL, EXT;
13 );
14
15 BEGIN
16 END ;
```

Сформований рядок лексем:

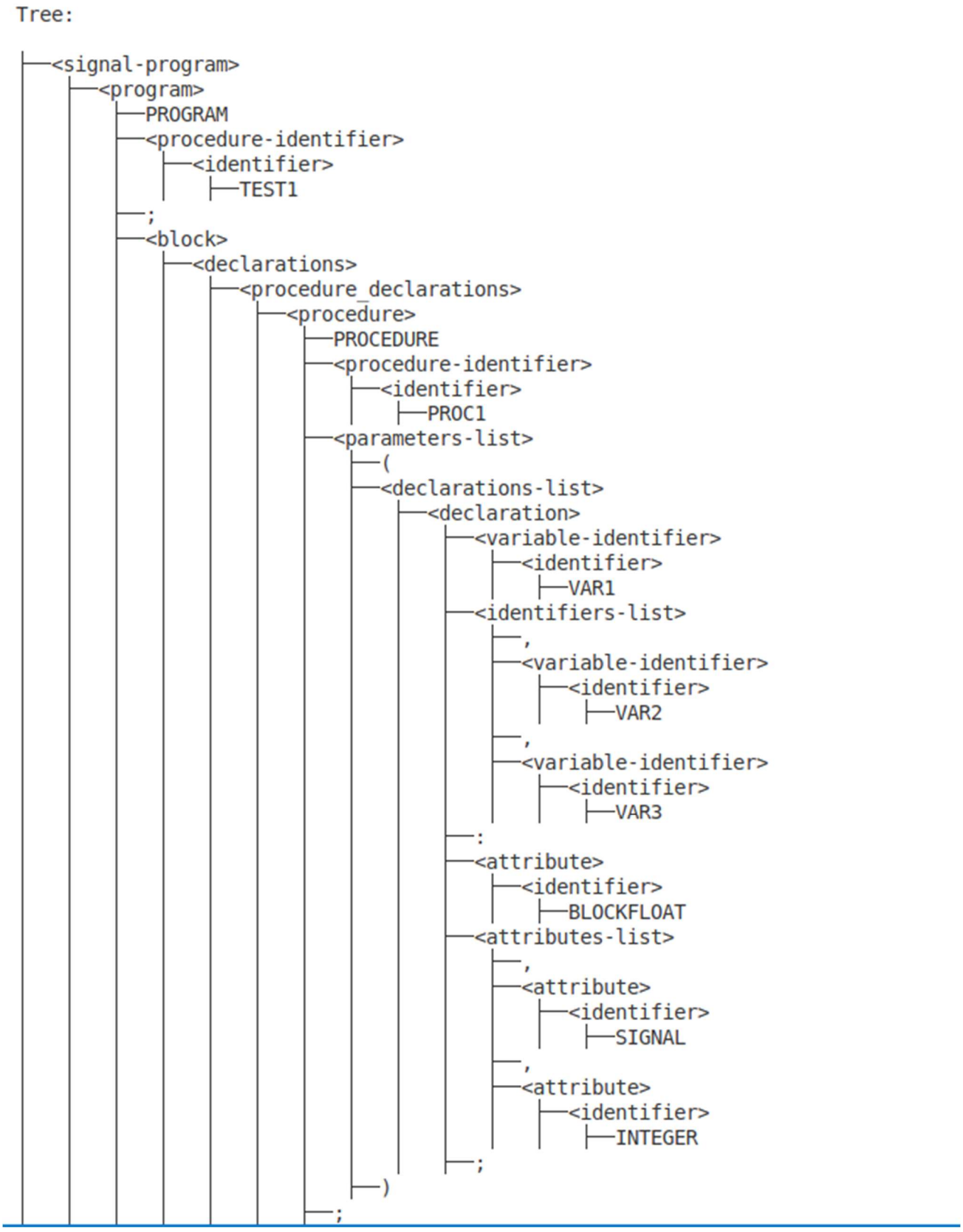
```
vlad@vlad-VirtualBox:~/kpi/opt/Parser$ ./parser trueTest1.txt

Lexem String:
      Line    Column      Code     Lexem
         1         2       401   PROGRAM
         1        10      1007     TEST1
         1        16         0         ;
         3         1       404 PROCEDURE
         3        11      1008     PROC1
         3        17         3         (
         4         5      1009      VAR1
         4         9         1         ,
         4        11      1010      VAR2
         4        15         1         ,
         4        17      1011      VAR3
         4        21         2         :
         4        23      1005BLOCKFLOAT
         4        33         1         ,
         4        35      1001    SIGNAL
         4        41         1         ,
         4        43      1003   INTEGER
         4        50         0         ;
         5         1         4         )
         5         2         0         ;
         6         1       404 PROCEDURE
         6        11      1012     PROC2
         6        16         0         ;
         7         1       404 PROCEDURE
         7        11      1013     PROC3
         7        17         3         (
         8         1         4         )
         8         2         0         ;
        10         1       404 PROCEDURE
        10        11      1014     PROC4
        10        17         3         (
        11         5      1015      VAR6
        11         9         1         ,
        11        11      1016      VAR7
        11        15         1         ,
        11        17      1017      VAR8
        11        21         2         :
        11        23      1005BLOCKFLOAT
        11        33         1         ,
        11        35      1001    SIGNAL
        11        41         1         ,
        11        43      1003   INTEGER
        11        50         0         ;
        12         5      1018     VAR11
        12        10         1         ,
```

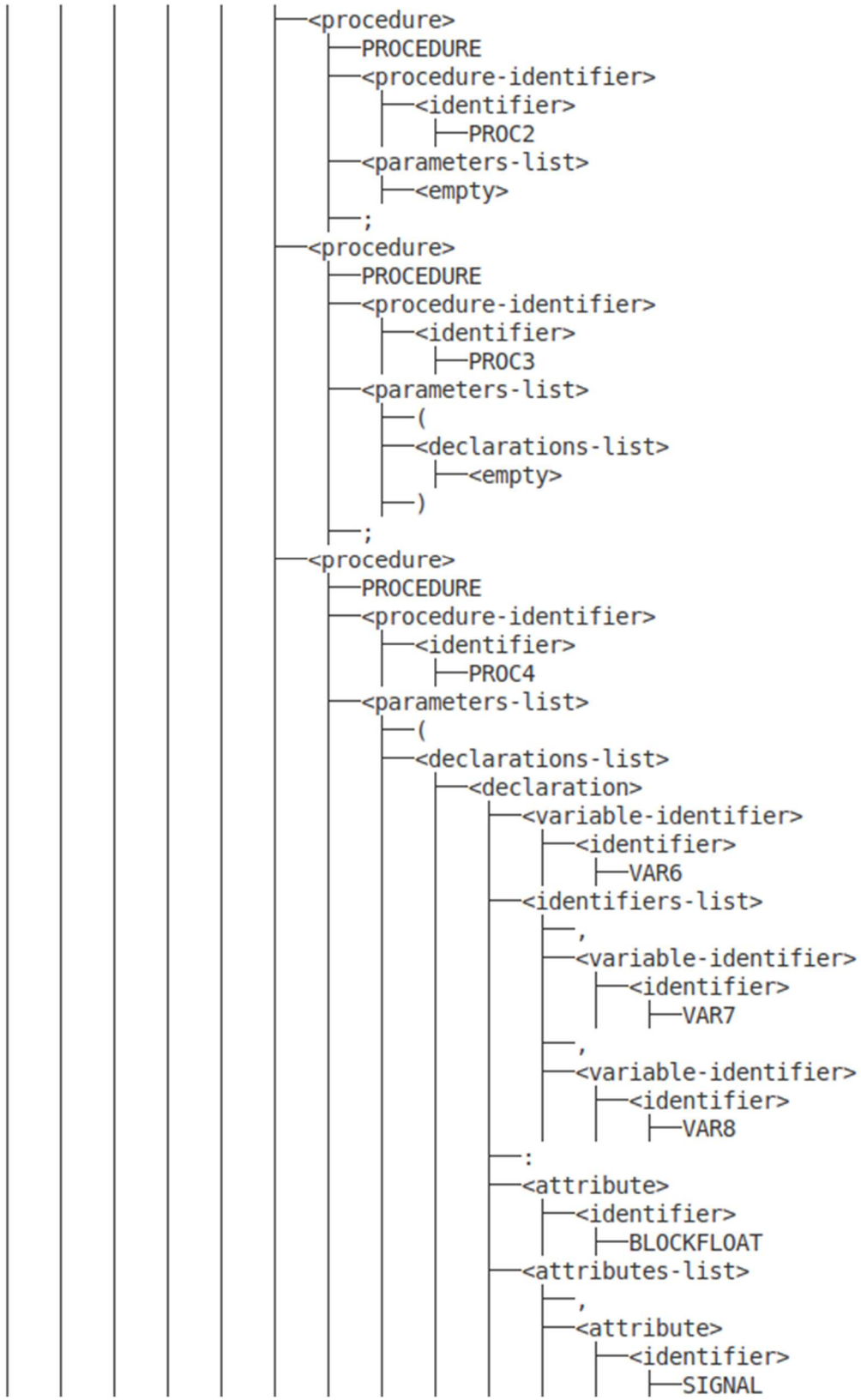
```
      12        12      1019     VAR12
      12        17         2         :
      12        19      1001    SIGNAL
      12        25         1         ,
      12        27      1006       EXT
      12        30         0         ;
      13         1         4         )
      13         2         0         ;
      15         1       402     BEGIN
      16         1       403       END
      16         5         0         ;
```

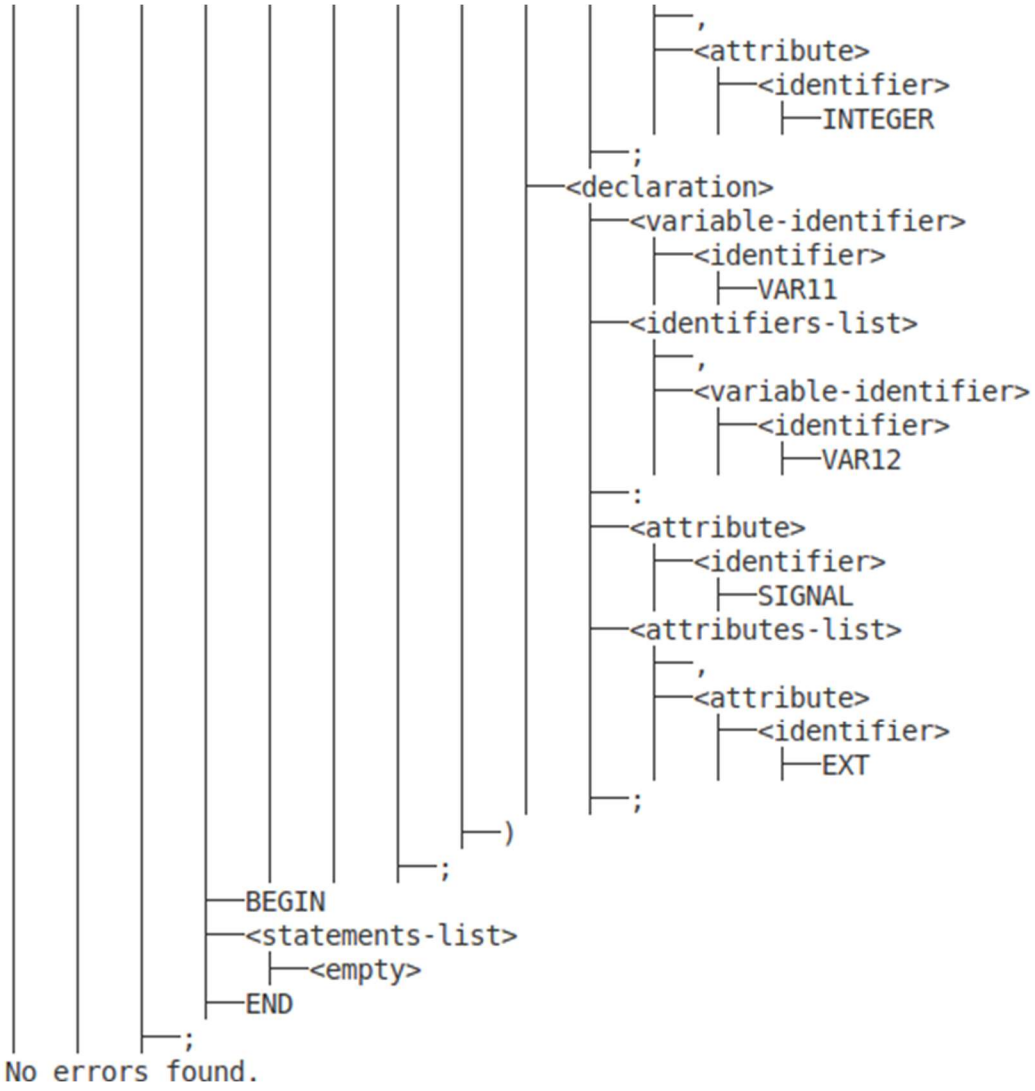
Інформаційні таблиці:

```
Informational tables:

Keywords:
Identifier      Code
   PROGRAM       401
     BEGIN       402
       END       403
 PROCEDURE       404

Identifiers:
Identifier      Code
    SIGNAL      1001
   COMPLEX      1002
   INTEGER      1003
     FLOAT      1004
BLOCKFLOAT      1005
       EXT      1006
     TEST1      1007
     PROC1      1008
      VAR1      1009
      VAR2      1010
      VAR3      1011
     PROC2      1012
     PROC3      1013
     PROC4      1014
      VAR6      1015
      VAR7      1016
      VAR8      1017
     VAR11      1018
     VAR12      1019

Delimiters:
 Delimiter      Code
         ;         0
         ,         1
         :         2
         (         3
         )         4
```

Побудоване дерево:




<procedure>
PROCEDURE
<procedure-identifier>
<identifier>
PROC2
<parameters-list>
<empty>
;
<procedure>
PROCEDURE
<procedure-identifier>
<identifier>
PROC3
<parameters-list>
(
<declarations-list>
<empty>
)
;
<procedure>
PROCEDURE
<procedure-identifier>
<identifier>
PROC4
<parameters-list>
(
<declarations-list>
<declaration>
<variable-identifier>
<identifier>
VAR6
<identifiers-list>
,
<variable-identifier>
<identifier>
VAR7
,
<variable-identifier>
<identifier>
VAR8
:
<attribute>
<identifier>
BLOCKFLOAT
<attributes-list>
,
<attribute>
<identifier>
SIGNAL



2. testTrue2.txt має код без оголошень жодної процедури (тобто <procedure_declarations> = <empty>):

trueTest2.txt

```
1   PROGRAM TEST1 ;
2
3   BEGIN
4   END ;
5
```

Сформований рядок лексем:

```
vlad@vlad-VirtualBox:~/kpi/opt/Parser$ ./parser trueTest2.txt

 Lexem String:
       Line    Column      Code     Lexem
          1         2       401   PROGRAM
          1        10      1007     TEST1
          1        16         0         ;
          3         1       402     BEGIN
          4         1       403       END
          4         5         0         ;

 Informational tables:
```

Інформаційні таблиці:

```
 Informational tables:

 Keywords:
 Identifier      Code
    PROGRAM       401
      BEGIN       402
        END       403
  PROCEDURE       404

 Identifiers:
 Identifier      Code
     SIGNAL      1001
    COMPLEX      1002
    INTEGER      1003
      FLOAT      1004
 BLOCKFLOAT      1005
        EXT      1006
      TEST1      1007

 Delimiters:
  Delimiter      Code
          ;         0
          ,         1
          :         2
          (         3
          )         4
```

Дерево розбору:

```
Tree:

├──<signal-program>
│   ├──<program>
│   │   ├──PROGRAM
│   │   ├──<procedure-identifier>
│   │   │   ├──<identifier>
│   │   │   │   ├──TEST1
│   │   ├──;
│   │   ├──<block>
│   │   │   ├──<declarations>
│   │   │   │   ├──<procedure_declarations>
│   │   │   │   │   ├──<empty>
│   │   │   ├──BEGIN
│   │   │   ├──<statements-list>
│   │   │   │   ├──<empty>
│   │   │   ├──END
│   │   ├──;
No errors found.
```

3. falseTest1.txt -- немає ‘;’ після ім’я програми:

falseTest1.txt

```
1   PROGRAM TEST1
2
3   BEGIN
4   END ;
5
6
```

Рядок Лексем:

```
Lexem String:
      Line    Column      Code     Lexem
         1         2       401   PROGRAM
         1        10      1007     TEST1
         3         1       402     BEGIN
         4         1       403       END
         4         5         0         ;
```

Інформаційні таблиці:

```
Informational tables:

Keywords:
Identifier      Code
   PROGRAM       401
     BEGIN       402
       END       403
 PROCEDURE       404

Identifiers:
Identifier      Code
    SIGNAL      1001
   COMPLEX      1002
   INTEGER      1003
     FLOAT      1004
BLOCKFLOAT      1005
       EXT      1006
     TEST1      1007

Delimiters:
 Delimiter      Code
         ;         0
         ,         1
         :         2
         (         3
         )         4
```

Дерево розбору:

```
Tree:
├──<signal-program>
│   ├──<program>
│   │   ├──PROGRAM
│   │   ├──<procedure-identifier>
│   │   │   ├──<identifier>
│   │   │   │   ├──TEST1
│   │   ├──<error>
Parser: Error (Line 3, Column 1): Expected ';' after program identifier
```

Робота синтаксичного аналізатора завершилася коректно з виведеням повідомлення про допущену помилку. В дереві розбору на місці

очікуваного символу вставлений елемент <error>.

4. falseTest2.txt – не оголошена директива BEGIN

falseTest2.txt

```
1   PROGRAM TEST1;
2
3
4   END ;
5
6
```

Рядок лексем:

```
vlad@vlad-VirtualBox:~/kpi/opt/Parser$ ./parser falseTest2.txt

Lexem String:
      Line    Column      Code     Lexem
         1         2       401   PROGRAM
         1        10      1007     TEST1
         1        15         0         ;
         4         1       403       END
         4         5         0         ;
```

Інформаційні таблиці:

```
Informational tables:

Keywords:
Identifier      Code
   PROGRAM       401
     BEGIN       402
       END       403
 PROCEDURE       404

Identifiers:
Identifier      Code
    SIGNAL      1001
   COMPLEX      1002
   INTEGER      1003
     FLOAT      1004
BLOCKFLOAT      1005
       EXT      1006
     TEST1      1007

Delimiters:
 Delimiter      Code
         ;         0
         ,         1
         :         2
         (         3
         )         4
```

Дерево розбору:

```
Tree:

├──<signal-program>
│    ├──<program>
│    │    ├──PROGRAM
│    │    ├──<procedure-identifier>
│    │    │    ├──<identifier>
│    │    │    │    ├──TEST1
│    │    ├──;
│    │    ├──<block>
│    │    │    ├──<declarations>
│    │    │    │    ├──<procedure_declarations>
│    │    │    │    │    ├──<procedure>
│    │    │    │    │    │    ├──<error>
│    │    │    ├──<error>
│    │    ├──;
Parser: Error (Line 4, Column 1): Expected 'PROCEDURE' keyword
Parser: Error (Line 4, Column 1): Expected 'BEGIN' keyword
```

Повідомлення відповідає помилці.

5. falseTest3.txt – не оголошена директива END

falseTest3.txt

```
1   PROGRAM TEST1 ;
2
3   PROCEDURE PROC1 (
4   );
5
6   BEGIN
7
```

Рядок лексем:

Lexem String:

Line	Column	Code	Lexem
1	2	401	PROGRAM
1	10	1007	TEST1
1	16	0	;
3	1	404	PROCEDURE
3	11	1008	PROC1
3	17	3	(
4	1	4	)
4	2	0	;
6	1	402	BEGIN

Інформаційні таблиці:

Keywords:

Identifier	Code
PROGRAM	401
BEGIN	402
END	403
PROCEDURE	404

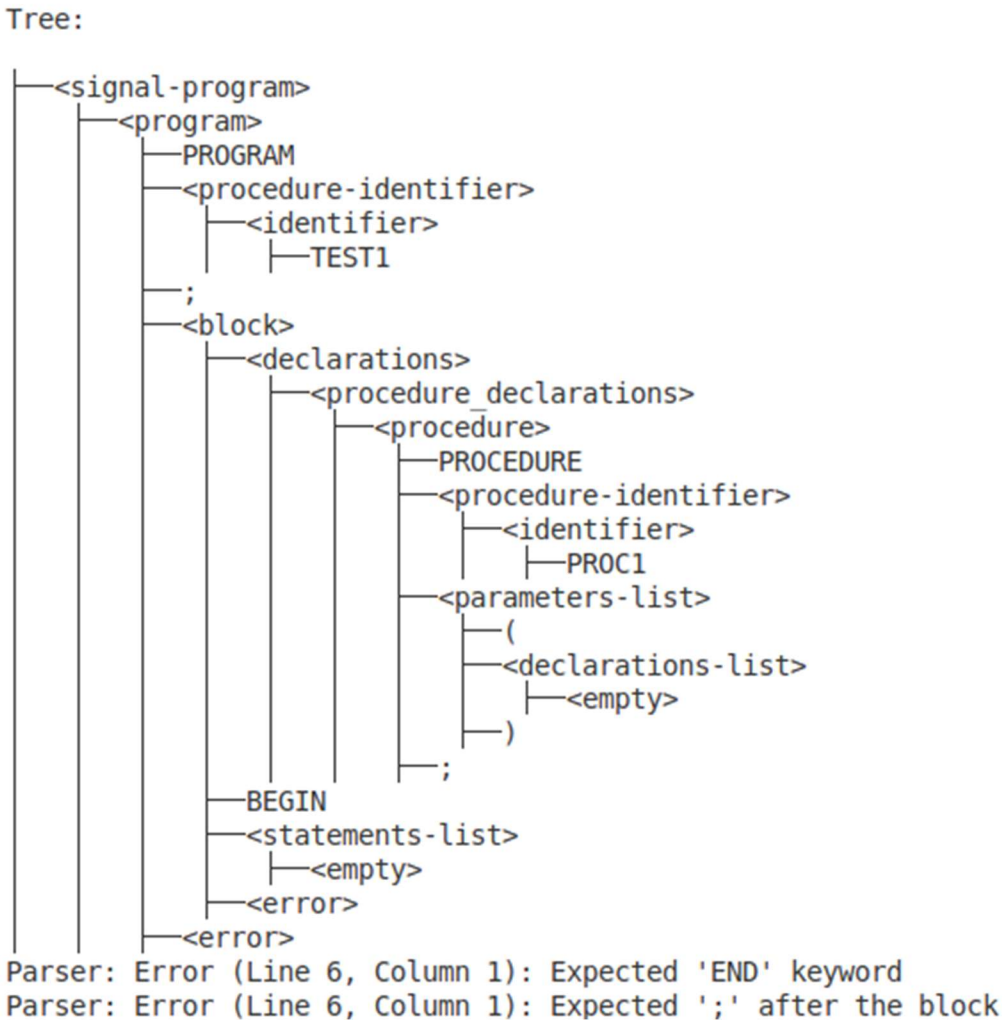
Identifiers:

Identifier	Code
SIGNAL	1001
COMPLEX	1002
INTEGER	1003
FLOAT	1004
BLOCKFLOAT	1005
EXT	1006
TEST1	1007
PROC1	1008

Delimiters:

Delimiter	Code
;	0
,	1
:	2
(	3
)	4

Дерево розбору:



Повідомлення про відсутність END

6. falseTest4 – ситуація коли оператор : не має операндів і присутня зайвий роздільник “;” після декларацій:

```
1   PROGRAM TEST1 ;
2
3   PROCEDURE PROC1 (
4       :
5   );
6
7   ;
8   BEGIN
9   END ;
10
11
```

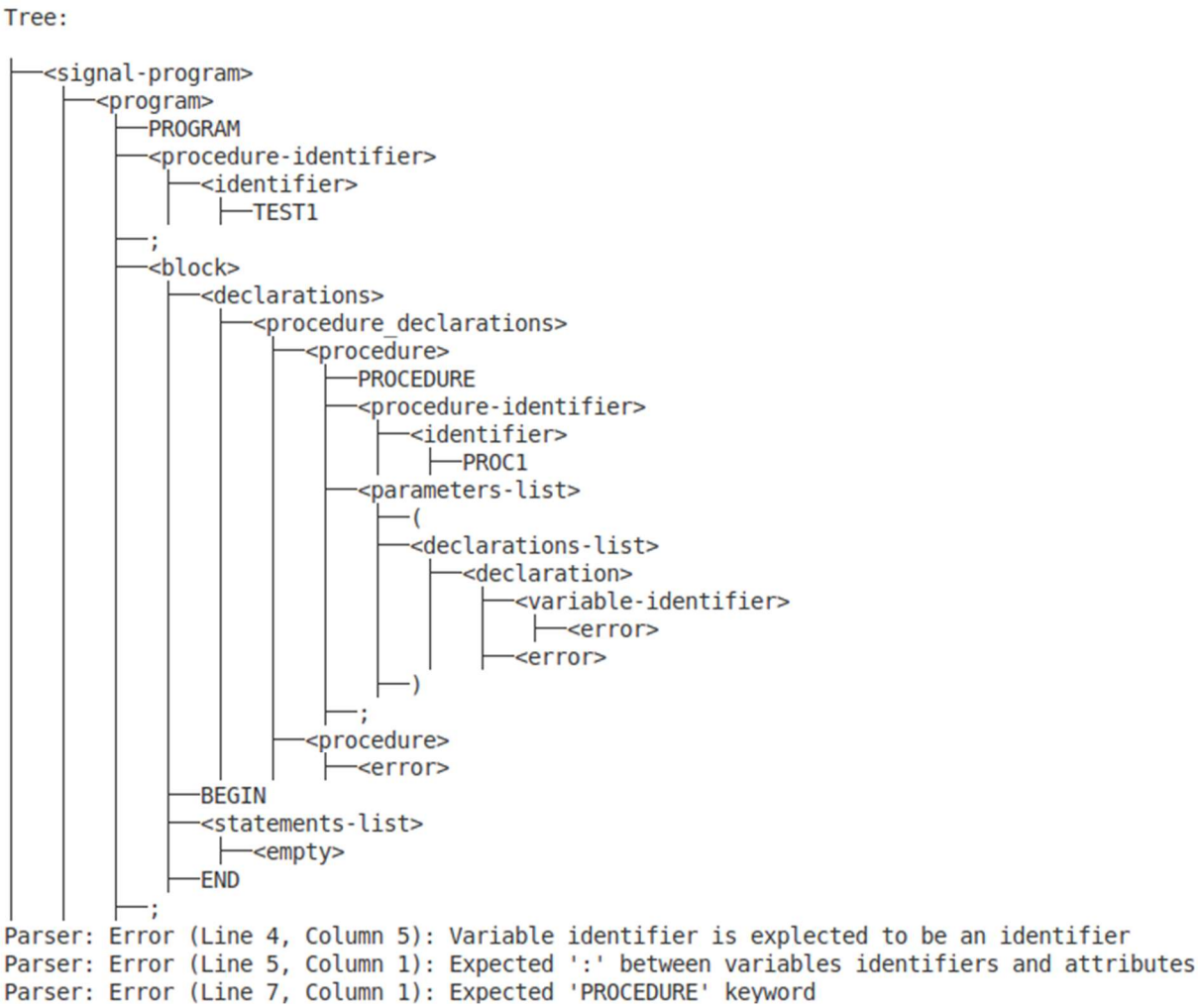
Рядок лексем:

```
Lexem String:
      Line    Column      Code     Lexem
         1         2       401   PROGRAM
         1        10      1007     TEST1
         1        16         0         ;
         3         1       404 PROCEDURE
         3        11      1008     PROC1
         3        17         3         (
         4         5         2         :
         5         1         4         )
         5         2         0         ;
         7         1         0         ;
         8         1       402     BEGIN
         9         1       403       END
         9         5         0         ;
```

Інформаційні таблиці:

```
Informational tables:

Keywords:
Identifier      Code
   PROGRAM       401
     BEGIN       402
       END       403
 PROCEDURE       404

Identifiers:
Identifier      Code
    SIGNAL      1001
   COMPLEX      1002
   INTEGER      1003
     FLOAT      1004
BLOCKFLOAT      1005
       EXT      1006
     TEST1      1007
     PROC1      1008

Delimiters:
 Delimiter      Code
         ;         0
         ,         1
         :         2
         (         3
         )         4
```

Дерево розбору:



В результаті маємо повідомлення про відповідні помилки