НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Розрахункова-графічна робота

з дисципліни «Основи проектування трансляторів»

Тема: «РОЗРОБКА СИНТАКСИЧНОГО АНАЛІЗАТОРА»

Виконав: студент III курсу

ФПМ групи КВ-81

Ядуха Б.В.

Викладач: Марченко О. І.

Київ 2021

Мета розрахунково-графічної роботи

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

Варіант 19

1. <signal-program> --> <program>

2. <program> --> PROGRAM <procedure-identifier> ;
<block>.

3. <block> --> <variable-declarations> BEGIN
<statements-list> END

4. <variable-declarations> --> VAR <declarations-list> |
<empty>

5. <declarations-list> --> <declaration>
<declarations-list> |
<empty>

6. <declaration> --> <variable-identifier>:<attribute> ;

7. <attribute> --> INTEGER |
FLOAT

8. <statements-list> --> <statement> <statementslist> |
<empty>

9. <statement> --> <condition-statement> ENDIF ;

10. <condition-statement> --> <incompletecondition-statement><alternative-part>

11. <incomplete-condition-statement> --> IF

<conditional-expression> THEN
<statements-list>

12. <alternative-part> --> ELSE <statements-list> |
<empty>

13. <conditional-expression> --> <expression> = <expression>

14. <expression> --> <variable-identifier> |
<unsigned-integer>

15. <variable-identifier> --> <identifier>

16. <procedure-identifier> --> <identifier>

17. <identifier> --> <letter><string>

18. <string> --> <letter><string> |
<digit><string> |
<empty>

19. <unsigned-integer> --> <digit><digits-string>

20. <digits-string> --> <digit><digits-string> |
<empty>

21. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

22. <letter> --> A | B | C | D | ... | Z

Таблиця роботи АМК

	Адреса операції	Код операції	AT (Адреса True)	AF (Адреса False)
1	<signal-program>	<program>	2	F
2	<program>	PROGRAM	3	F
3		<procedure-identifier>	4	F
4		;	5	F
5		<block>	6	F
6		.	T	F
7	<block>	<variable-declarations>	8	F
8		BEGIN	9	F
9		<statements-list>	10	F
10		END	T	F
11	<variable-declarations>	VAR	if (lexem == VAR) goto 12 else return T	Якщо знизу повернули F, то повертаємо F
12		<declarations-list>	13	F
13	<declarations-list>	<declaration>	if(lexem==<identifier>) goto 15 else return T	Якщо знизу повернули F, то повертаємо F
14		<declarations-list>	13	F
15	<declaration>	<variable-identifier>	16	F
16		:	17	F
17		<attribute>	18	F
18		;	T	F
19	<attribute>	INTEGER	if (lexem == INTEGER) return T else goto 20	Якщо знизу повернули F, то повертаємо F
20		FLOAT	T	F
21	<statements-list>	<statement>	if(lexem==IF) goto 23 else return T	F
22		<statements-list>	T	F
23	<statement>	<condition-statement>	24	F
24		ENDIF	25	F
25		;	T	F
26	<condition-statement>	<incomplete-condition-statement>	27	F
27		<alternative-part>	T	F
28	<incomplete-condition-statement>	IF	29	F

29		<conditional-expression>	30	F
30		THEN	31	F
31		<statements-list>	T	F
32	<alternative-part>	ELSE	if (lexem == ELSE) goto 33 else return true	Якщо знизу повернули F, то повертаємо F
33		<statements-list>	T	F
34	<conditional-expression>	<expression>	35	F
35		=	36	F
36		<expression>	T	F
37	<expression>	<variable-identifier>	if (lexem == <variable-identifier>) return T else goto 38	Якщо знизу повернули F, то повертаємо F
38		<unsigned-integer>	T	F
39	<variable-identifier>	<identifier>	T	F
40	<procedure-identifier>	<identifier>	T	F

Source.cpp

```
#include "lexer.h"
#include "parser.h"
#include "parser_tree.h"

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char** args)
{
	fill_symbol_categories();
	for (int i = 1; i < argc; i++)
	{
		string path(args[i]);
		Lexer l;
		l.Parse(path + "/input.sig");
		ofstream of(path + "/generated.txt");
		l.Print(of);
		l.ShowErrors(of);
		of << "\n";
		Parser p;
		p.Parse();
		p.Print(of);
		of.close();
	}
```

}}

parser_tree.h

```
#pragma once
#include "tables.h"
#include <vector>

using namespace std;
```

```
struct ParserTreeNode {
	TockenTableField value;
	vector<ParserTreeNode*> children;
};

ParserTreeNode* AddNode(ParserTreeNode* tree, const TockenTableField& value);
void DeleteTree(ParserTreeNode* tree);
ofstream& operator<<(ofstream& os, ParserTreeNode* tree);

void DeleteSubTree(ParserTreeNode* tree);
void PrintTree(ofstream& os, ParserTreeNode* tree, int depth_count);
```

parser_tre.cpp

```
#include "parser_tree.h"

using namespace std;

ParserTreeNode* AddNode(ParserTreeNode* tree, const TockenTableField& value)
{
	ParserTreeNode* new_node = new ParserTreeNode;
	new_node->value = value;
	if (tree == nullptr)
	{
		tree = new_node;
	}
	else
	{
		tree->children.push_back(new_node);
	}
	return new_node;
}

void DeleteTree(ParserTreeNode* tree)
{
	if (tree != nullptr)
	{
		DeleteSubTree(tree);
	}
}


void DeleteSubTree(ParserTreeNode* tree)
{

	for (ParserTreeNode* elem : tree->children)
	{
		DeleteSubTree(elem);
	}
	delete tree;

}

ofstream& operator<<(ofstream& os, ParserTreeNode* tree)
{
	if (tree != nullptr)
	{
		PrintTree(os, tree, 0);
	}
	else
	{
		os << "Empty tree!\n";
```

```
	}
	return os;
}

void PrintTree(ofstream& os, ParserTreeNode* tree, int depth_count)
{
	for (int i = 0; i < depth_count * 2; i++)
	{
		os << ".";
	}
	if (tree->value.id == -1)
	{
		os << tree->value.value << "\n";
	}
	else
	{
		os << tree->value.id << " " << tree->value.value << "\n";
	}
	for (ParserTreeNode* elem : tree->children)
	{
		PrintTree(os, elem, depth_count + 1);
	}
}
```

parser.h

```
#pragma once
#include "lexer.h"
#include "parser_tree.h"
#include "tables.h"
#include <vector>
#include <string>
#include <fstream>

using namespace std;

enum class ParseStages {
	ProgramFirstPart,
	ProgramSeconPart,
	ProgramThirdPart,
	BlockFirstPart,
	BlockSecondPart,
	BlockThirdPart,
	VariableDeclaration,
	DeclarationsList,
	DeclarationFirstPart,
	DeclarationSecondPart,
	DeclarationThirdPart,
	StatementsList,
	Attribute,
	ProcedureIdentifier,
	VariableIdentifier,
	Identifier,
	StatementFirstPart,
	StatementSecondPart,
	ConditionStatementFirstPart,
	ConditionStatementSecondPart,
	IncompleteConditionStatementFirstPart,
	IncompleteConditionStatementSecondPart,
	AlternativePart,
	ConditionalExpressionFirstPart,
	ConditionalExpressionSecondPart,
	Expression,
```

```
	UnsignedInteger,
	Return,
	End
};

class Parser {
public:
	void Parse();
	void Print(ofstream& os);
private:
	void make_error(int pos, string expected);
	ParserTreeNode* tree;
	string error = "";
};
```

parser.cpp

```
#include "parser.h"
#include <stack>

using namespace std;

void Parser::Parse()
{
	DeleteTree(tree);
	int len = tocken_table.size();
	if (len == 0)
	{
		return;
	}
	int current_tocken_position = 0;
	ParseStages current_stage = ParseStages::ProgramFirstPart;
	ParserTreeNode* current_node;
	stack<ParserTreeNode*> return_nodes;
	stack<ParseStages> next_stages;
	bool is_end = false;
	TockenTableField tmp = { tocken_table[0].column, tocken_table[0].line, -1,
"<signal-program>" };
	tree = AddNode(tree, tmp);
	tmp = { tocken_table[len - 1].column, tocken_table[len - 1].line, -1, "end" };
	current_node = AddNode(tree, tmp);
	tmp.value = "end";
	tocken_table.push_back(tmp);
	while (!is_end)
	{

		switch (current_stage)
		{
		case ParseStages::ProgramFirstPart:
			if (tocken_table[current_tocken_position].id == 401)
			{
				AddNode(current_node,
tocken_table[current_tocken_position]);
				current_tocken_position++;
				next_stages.push(ParseStages::ProgramSeconPart);
				return_nodes.push(current_node);
				current_stage = ParseStages::ProcedureIdentifier;

			}
			else
			{
				make_error(current_tocken_position, "'PROGRAM'");
				is_end = true;
```

```
                        }

                        break;
                case ParseStages::ProgramSeconPart:
                        if (tocken_table[current_tocken_position].id == ';')
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                next_stages.push(ParseStages::ProgramThirdPart);
                                current_stage = ParseStages::BlockFirstPart;
                                return_nodes.push(current_node);
                        }
                        else
                        {
                                make_error(current_tocken_position, "';'");
                                is_end = true;
                        }
                        break;
                case ParseStages::ProgramThirdPart:
                        if (tocken_table[current_tocken_position].id == '.')
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::End;
                                //is_end = true;
                        }
                        else
                        {
                                make_error(current_tocken_position, "'.'");
                                is_end = true;
                        }
                        break;
                case ParseStages::ProcedureIdentifier:
                        tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<procedure-identifier>" };
                        current_node = AddNode(current_node, tmp);
                        current_stage = ParseStages::Identifier;
                        break;
case ParseStages::VariableIdentifier:
                        tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<variable-identifier>" };
                        current_node = AddNode(current_node, tmp);
                        current_stage = ParseStages::Identifier;
                        break;
                case ParseStages::Identifier:
                        if (tocken_table[current_tocken_position].id > 1000)
                        {
                                tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<identifier>" };
                                current_node = AddNode(current_node, tmp);
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::Return;
                        }
                        else
                        {
                                make_error(current_tocken_position, "<identifier>");
                                is_end = true;
                        }
```

```
                        break;
                case ParseStages::BlockFirstPart:
                        tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<block>" };
                        current_node = AddNode(current_node, tmp);
                        return_nodes.push(current_node);
                        next_stages.push(ParseStages::BlockSecondPart);
                        current_stage = ParseStages::VariableDeclaration;
                        break;
                case ParseStages::BlockSecondPart:
                        if (tocken_table[current_tocken_position].id == 402)
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                return_nodes.push(current_node);
                                next_stages.push(ParseStages::BlockThirdPart);
                                current_stage = ParseStages::StatementsList;
                        }
                        else
                        {
                                is_end = true;
                                make_error(current_tocken_position, "'BEGIN'");
                        }
                        break;
                case ParseStages::BlockThirdPart:
                        if (tocken_table[current_tocken_position].id == 403)
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::Return;
                        }
                        else
                        {
                                is_end = true;
                                make_error(current_tocken_position, "'END'");
                        }
                        break;
                case ParseStages::VariableDeclaration:
                        tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<variable-declaration>" };
                        current_node = AddNode(current_node, tmp);
                        if (tocken_table[current_tocken_position].id == 404)
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::DeclarationsList;
                        }
        else
                        {
                                tmp.value = "<empty>";
                                AddNode(current_node, tmp);
                                current_stage = ParseStages::Return;
                        }
                        break;
                case ParseStages::DeclarationsList:
                        tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<declarations-list>" };
                        current_node = AddNode(current_node, tmp);
                        if (tocken_table[current_tocken_position].id > 1000)
```

```
                        {
                                return_nodes.push(current_node);
                                next_stages.push(ParseStages::DeclarationsList);
                                current_stage = ParseStages::DeclarationFirstPart;
                        }
                        else
                        {
                                tmp.value = "<empty>";
                                AddNode(current_node, tmp);
                                current_stage = ParseStages::Return;
                        }
                        break;
                case ParseStages::DeclarationFirstPart:
                        next_stages.push(ParseStages::DeclarationSecondPart);
                        return_nodes.push(current_node);
                        tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<declaration>" };
                        current_node = AddNode(current_node, tmp);
                        tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<variable-identifier>" };
                        current_node = AddNode(current_node, tmp);
                        current_stage = ParseStages::Identifier;
                        break;
                case ParseStages::DeclarationSecondPart:
                        if (tocken_table[current_tocken_position].id == ':')
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                return_nodes.push(current_node);
                                next_stages.push(ParseStages::DeclarationThirdPart);
                                current_stage = ParseStages::Attribute;
                        }
                        else
                        {
                                make_error(current_tocken_position, "':'");
                                is_end = true;
                        }
                        break;
                case ParseStages::DeclarationThirdPart:
                        if (tocken_table[current_tocken_position].id == ';')
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::Return;
                        }
                        else
                        {
                                make_error(current_tocken_position, "';'");
                                is_end = true;
                        }
                        break;
                case ParseStages::Attribute:
                        if (tocken_table[current_tocken_position].id == 405 ||
tocken_table[current_tocken_position].id == 406)
                        {
                                tmp = { tocken_table[current_tocken_position].column,
tocken_table[current_tocken_position].line, -1, "<attribute>" };
                                current_node = AddNode(current_node, tmp);
                                AddNode(current_node,
tocken_table[current_tocken_position]);
```

```
                        current_tocken_position++;
                        current_stage = ParseStages::Return;
                }
                else
                {

                        make_error(current_tocken_position, "'INTEGER' or
'FLOAT'");
                        is_end = true;
                }
                break;
        case ParseStages::StatementsList:
                tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<statements-list>" };
                current_node = AddNode(current_node, tmp);
                if (tocken_table[current_tocken_position].id == 408)
                {
                        return_nodes.push(current_node);
                        next_stages.push(ParseStages::StatementsList);
                        current_stage = ParseStages::StatementFirstPart;
                }
                else
                {
                        tmp.value = "<empty>";
                        AddNode(current_node, tmp);
                        current_stage = ParseStages::Return;
                }
                break;
        case ParseStages::StatementFirstPart:
                tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<statement>" };
                current_node = AddNode(current_node, tmp);
                return_nodes.push(current_node);
                next_stages.push(ParseStages::StatementSecondPart);
                current_stage = ParseStages::ConditionStatementFirstPart;
                break;
        case ParseStages::StatementSecondPart:
                if (tocken_table[current_tocken_position].id == 407)
                {
                        AddNode(current_node,
tocken_table[current_tocken_position]);
                        current_tocken_position++;
                        if (tocken_table[current_tocken_position].id == ';')
                        {
                                AddNode(current_node,
tocken_table[current_tocken_position]);
                                current_tocken_position++;
                                current_stage = ParseStages::Return;
                        }
                        else
                        {
                                make_error(current_tocken_position, "';'");
                                is_end = true;
                        }
                }
                else
                {
                        make_error(current_tocken_position, "'ENDIF'");
                        is_end = true;
                }
                break;
        case ParseStages::ConditionStatementFirstPart:
```

```
                    tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<condition-statement>" };
                    current_node = AddNode(current_node, tmp);
                    return_nodes.push(current_node);
                    next_stages.push(ParseStages::ConditionStatementSecondPart);
                    current_stage =
ParseStages::IncompleteConditionStatementFirstPart;
                    break;
             case ParseStages::ConditionStatementSecondPart:
                    current_stage = ParseStages::AlternativePart;
                    break;
             case ParseStages::IncompleteConditionStatementFirstPart:
                    tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<incomplete-condition-statement>" };
                    current_node = AddNode(current_node, tmp);
                    if (tocken_table[current_tocken_position].id == 408)
                    {
                           AddNode(current_node,
tocken_table[current_tocken_position]);
                           current_tocken_position++;

      next_stages.push(ParseStages::IncompleteConditionStatementSecondPart);
                           return_nodes.push(current_node);
                           current_stage =
ParseStages::ConditionalExpressionFirstPart;

                    }
                    else
                    {
                           make_error(current_tocken_position, "'IF'");
                           is_end = true;
                    }
                    break;

             case ParseStages::IncompleteConditionStatementSecondPart:
                    if (tocken_table[current_tocken_position].id == 409)
                    {
                           AddNode(current_node,
tocken_table[current_tocken_position]);
                           current_tocken_position++;
                           current_stage = ParseStages::StatementsList;
                    }
                    else
                    {
                           make_error(current_tocken_position, "'THEN'");
                           is_end = true;
                    }
                    break;

             case ParseStages::AlternativePart:
                    tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<alternative-part>" };
                    current_node = AddNode(current_node, tmp);
                    if (tocken_table[current_tocken_position].id == 410)
                    {
                           AddNode(current_node,
tocken_table[current_tocken_position]);
                           current_tocken_position++;
                           current_stage = ParseStages::StatementsList;
                    }
                    else
                    {
```

```
                        tmp.value = "<empty>";
                        AddNode(current_node, tmp);
                        current_stage = ParseStages::Return;
                }
                break;

        case ParseStages::ConditionalExpressionFirstPart:
                tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<conditional-expression>" };
                current_node = AddNode(current_node, tmp);
                return_nodes.push(current_node);
                next_stages.push(ParseStages::ConditionalExpressionSecondPart);
                current_stage = ParseStages::Expression;
                break;

        case ParseStages::ConditionalExpressionSecondPart:
                if (tocken_table[current_tocken_position].id == '=')
                {
                        AddNode(current_node,
tocken_table[current_tocken_position]);
                        current_tocken_position++;
                        current_stage = ParseStages::Expression;
                }
                else
                {
                        make_error(current_tocken_position, "=");
                        is_end = true;
                }
                break;

        case ParseStages::Expression:
                tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<expretion>" };
                current_node = AddNode(current_node, tmp);
                if (tocken_table[current_tocken_position].id > 1000)
                {
                        current_stage = ParseStages::VariableIdentifier;
                }
                else
                {
                        if (tocken_table[current_tocken_position].id > 500)
                        {
                                current_stage = ParseStages::UnsignedInteger;
                        }
                        else
                        {
                                make_error(current_tocken_position, "<identifier>
or <unsigned-integer>");
                                is_end = true;
                        }
                }
                break;

        case ParseStages::UnsignedInteger:
                tmp = { tocken_table[current_tocken_position].column ,
tocken_table[current_tocken_position].line, -1, "<unsigned-integer>" };
                current_node = AddNode(current_node, tmp);
                if (tocken_table[current_tocken_position].id > 500)
                {
                        AddNode(current_node,
tocken_table[current_tocken_position]);
                        current_tocken_position++;
```

```
                        current_stage = ParseStages::Return;
                }
                else
                {
                        make_error(current_tocken_position, "<unsigned-integer>");
                        is_end = true;
                }
                break;
        case ParseStages::Return:
                current_stage = next_stages.top();
                current_node = return_nodes.top();
                next_stages.pop();
                return_nodes.pop();
                break;
        case ParseStages::End:
                if (tocken_table[current_tocken_position].id != -1)
                {
                        make_error(current_tocken_position, "end of file");
                }
                is_end = true;
                break;
        }
    }
}

void Parser::Print(ofstream& os)
{
    os << tree;
    os << error;
}

void Parser::make_error(int pos, string expected)
{
    error = "Parser: Error(line " + to_string(tocken_table[pos].line) +
        ", column " + to_string(tocken_table[pos].column) +
        "):";
    if (tocken_table[pos].id != -1)
    {
        error += "Found '" + tocken_table[pos].value +
            "' Expected " + expected;
    }
    else
    {
        error += "File end"
            " Expected " + expected;
    }

}
```

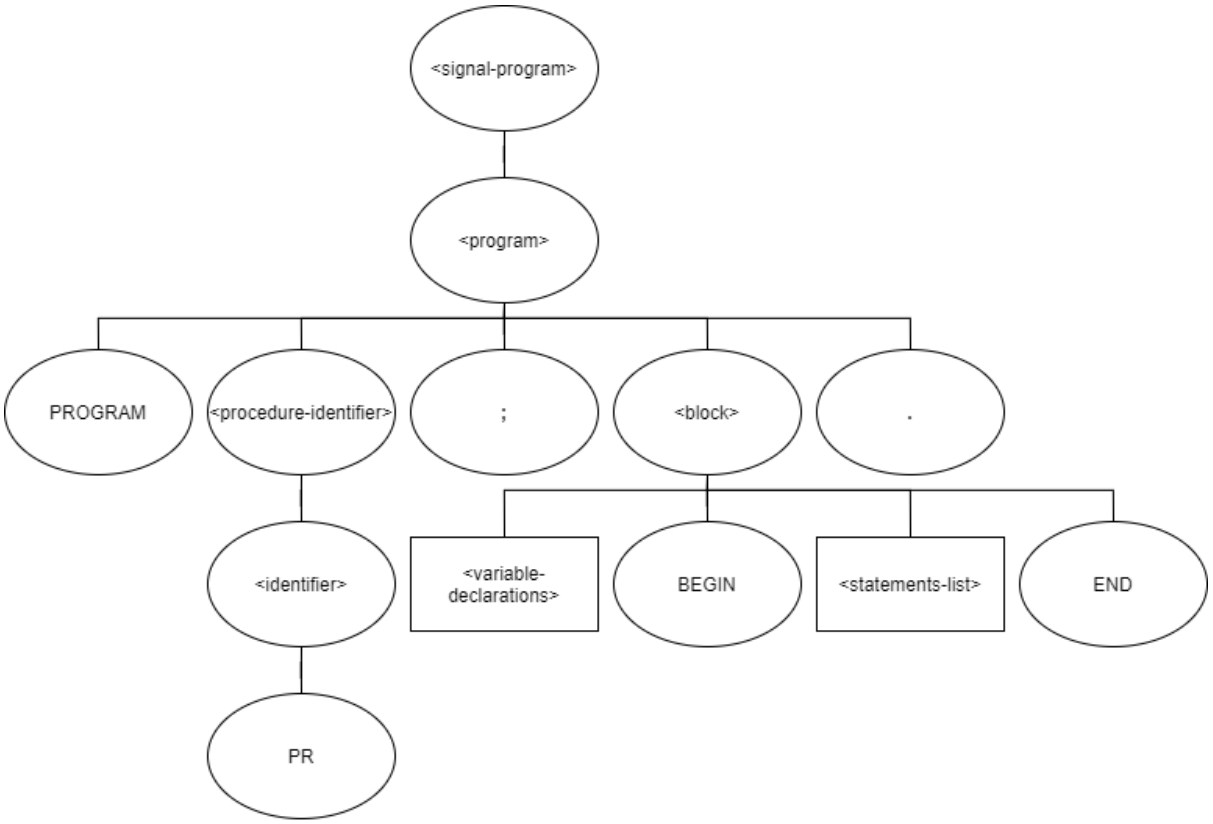
tables.h

```
#pragma once
#include <map>
#include <set>
#include <vector>
#include <string>
#include <fstream>

using namespace std;

struct TockenTableField
```

```
{
	int column, line, id;
	string value;
};
extern map<string, int> key_word_table;
extern map<string, int> identifier_table;
extern map<string, int> constants_table;
extern map<string, int> control_task_table;
extern vector<TockenTableField> tocken_table;
```

Контрольні приклади

Дерева усіх прикладів мають таку структуру:



Де <variable-declarations> і <statements-list> є піддеревами, тому надалі це дерево не дублюватиметься.

Приклад №1

input.sig:

```
PROGRAM PR;
	VAR V1:INTEGER;
BEGIN
	IF 100 = V2 THEN
	ELSE
	ENDIF;
END.
```

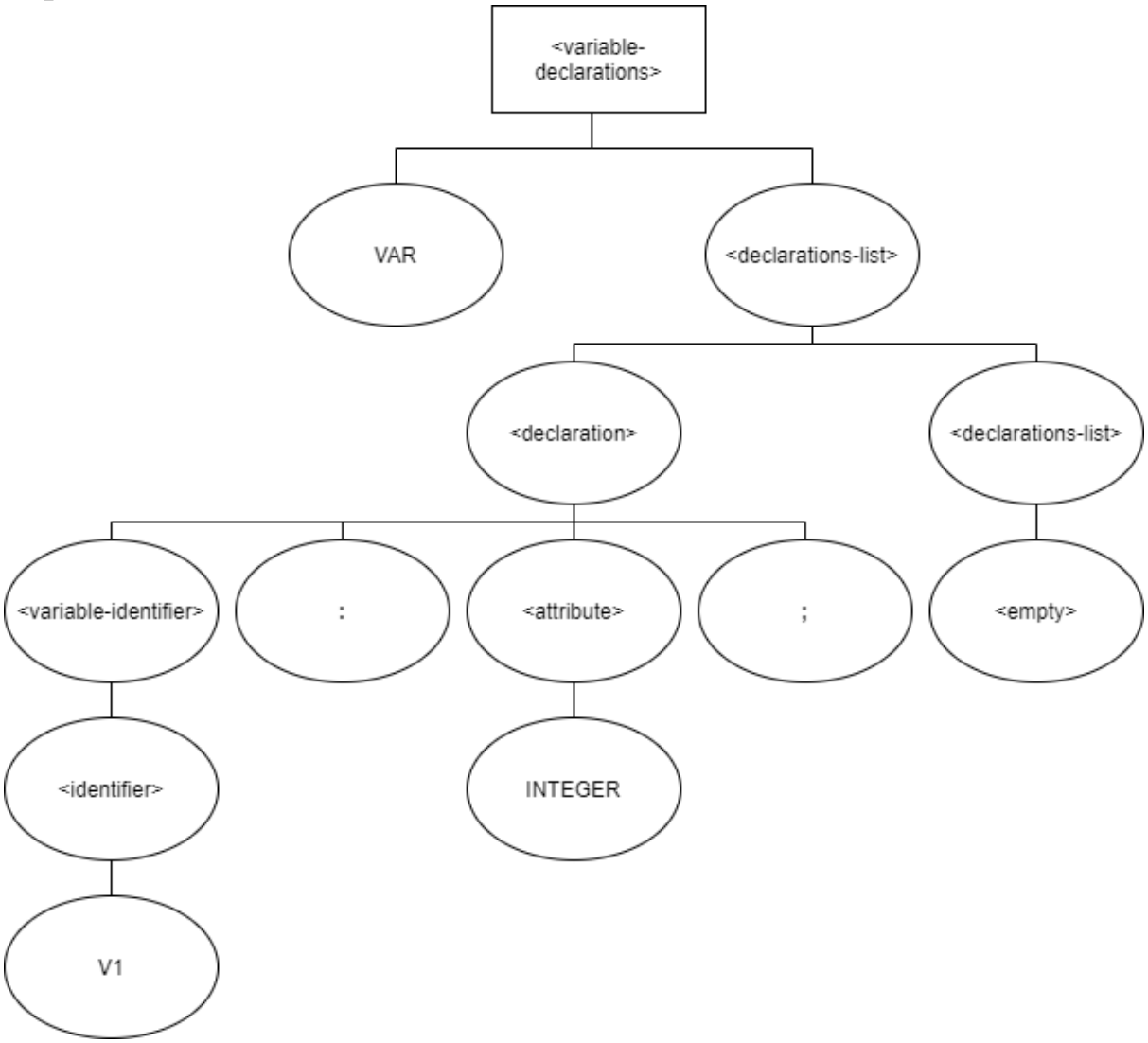
expected.txt:

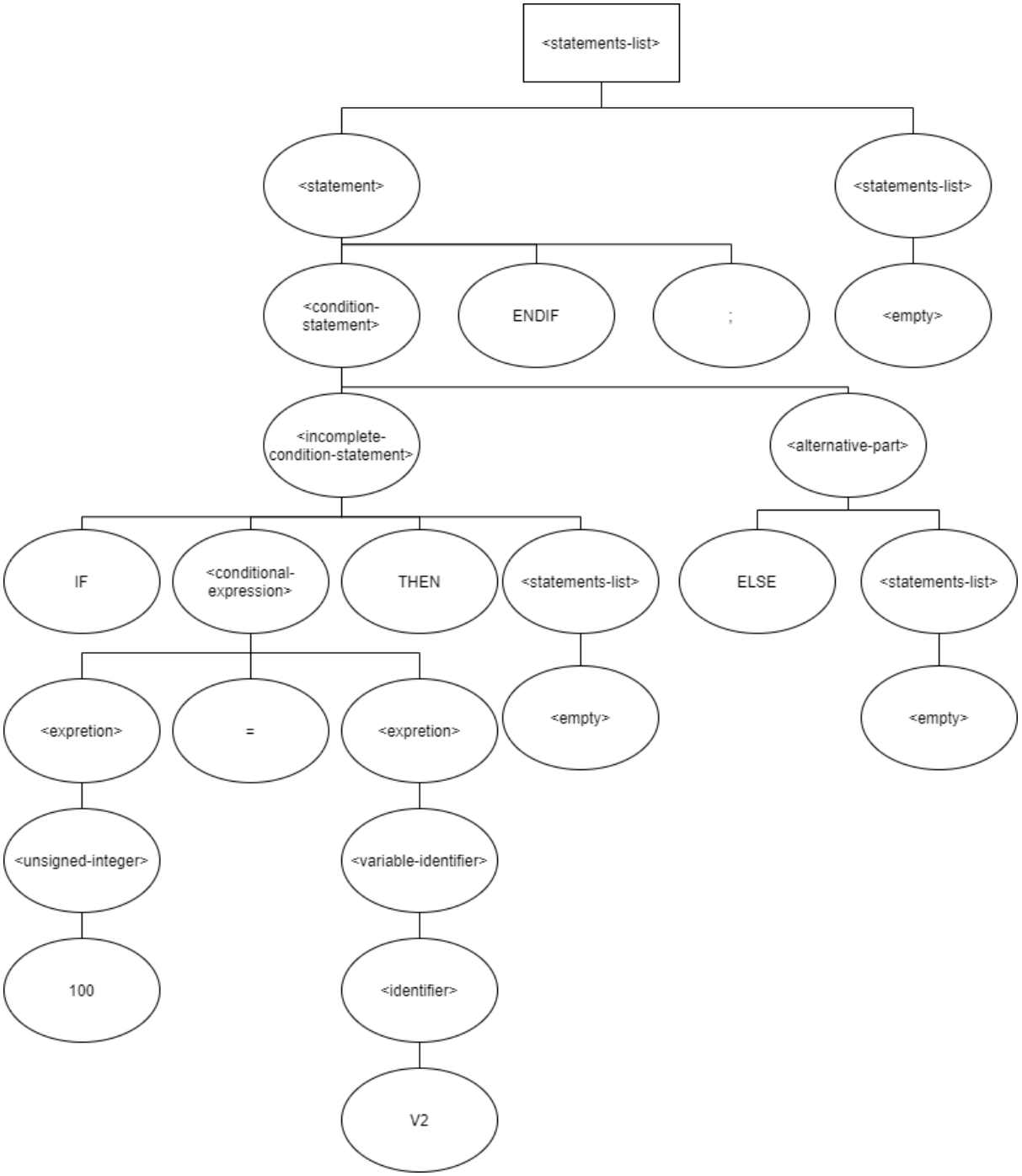
```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
```

```
2 16 59 ;
3 1 402 BEGIN
4 2 408 IF
4 5 501 100
4 9 61 =
4 11 1003 V2
4 14 409 THEN
5 2 410 ELSE
6 2 407 ENDIF
6 7 59 ;
7 1 403 END
7 4 46 .

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
......<variable-declaration>
........404 VAR
........<declarations-list>
..........<declaration>
............<variable-identifier>
..............<identifier>
................1002 V1
..........58 :
..........<attribute>
............405 INTEGER
..........59 ;
..........<declarations-list>
............<empty>
......402 BEGIN
......<statements-list>
........<statement>
```

.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....501 100
.....61 =
.....<expretion>
.....<variable-identifier>
.....<identifier>
.....1003 V2
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....403 END
....46 .

Дерево:


<variable-declarations>
VAR
<declarations-list>
<declaration>
<declarations-list>
<variable-identifier>
:
<attribute>
;
<empty>
<identifier>
INTEGER
V1


<statements-list>
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<empty>
<incomplete-condition-statement>
<alternative-part>
IF
<conditional-expression>
THEN
<statements-list>
ELSE
<statements-list>
<expretion>
=
<expretion>
<empty>
<empty>
<unsigned-integer>
<variable-identifier>
100
<identifier>
V2

Приклад №2

input.sig:

```
PROGRAM PR;
        VAR V1:INTEGER;
         V2:FLOAT;
           V3:INTEGER;
BEGIN
END.
```

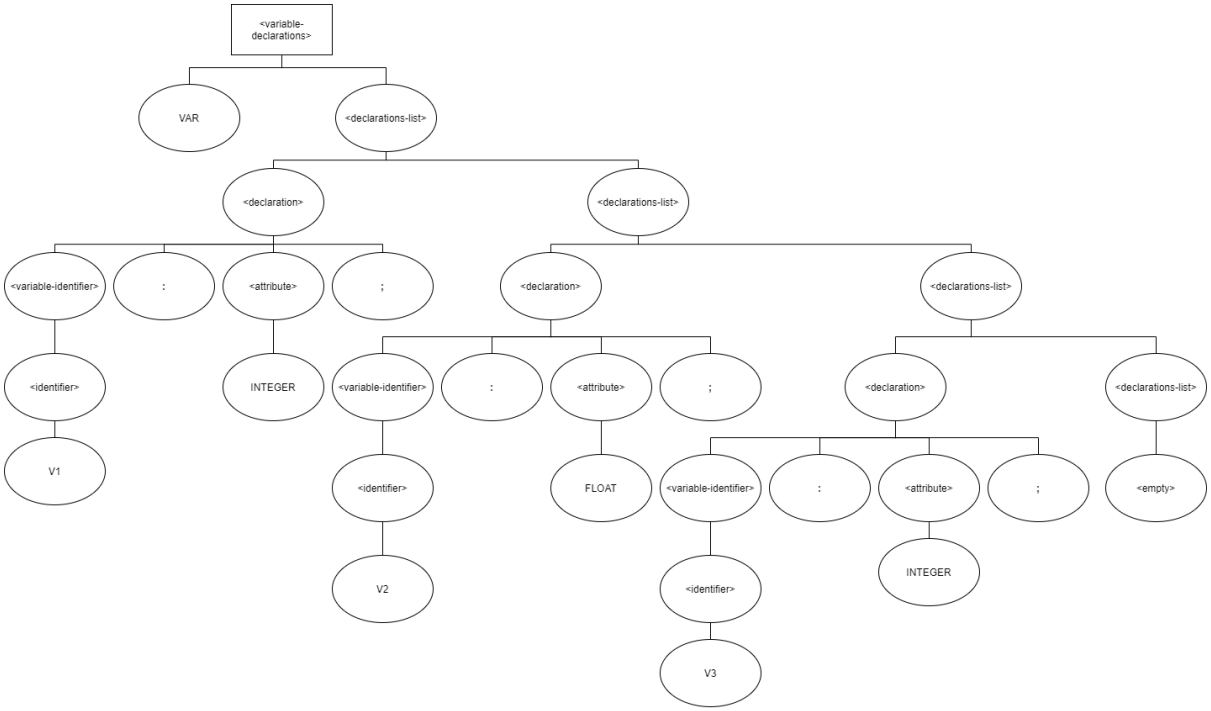
expected.txt:

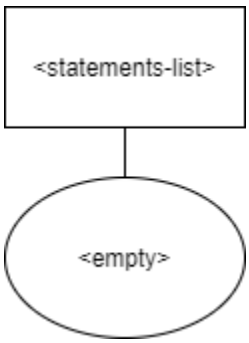
```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
2 16 59 ;
3 13 1003 V2
3 15 58 :
3 16 406 FLOAT
3 21 59 ;
4 6 1001 V3
4 8 58 :
4 9 405 INTEGER
4 16 59 ;
5 1 402 BEGIN
6 1 403 END
6 4 46 .
```

```
<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
```

.....<variable-declaration>
.....404 VAR
.....<declarations-list>
.....<declaration>
.....<variable-identifier>
.....<identifier>
.....1002 V1
.....58 :
.....<attribute>
.....405 INTEGER
.....59 ;
.....<declarations-list>
.....<declaration>
.....<variable-identifier>
.....<identifier>
.....1003 V2
.....58 :
.....<attribute>
.....406 FLOAT
.....59 ;
.....<declarations-list>
.....<declaration>
.....<variable-identifier>
.....<identifier>
.....1001 V3
.....58 :
.....<attribute>
.....405 INTEGER
.....59 ;
.....<declarations-list>
.....<empty>
.....402 BEGIN
.....<statements-list>
.....<empty>
.....403 END
....46 .

Дерево:





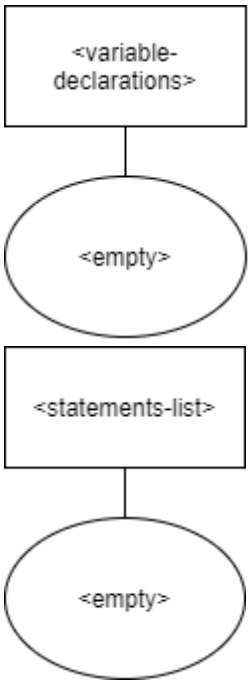
Приклад №3

input.sig:
PROGRAM PR;
BEGIN
END.
expected.txt:
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
3 1 403 END
3 4 46 .

<signal-program>

..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
......<variable-declaration>
........<empty>
......402 BEGIN
......<statements-list>
........<empty>
......403 END
....46 .

Дерево:



Приклад №4

input.sig:

```
PROGRAM PR;
BEGIN
        IF 100 = 200 THEN
                IF 21 = 12 THEN
                ENDIF;
        ELSE
                IF 21 = 12 THEN
```

```
                ENDIF;
        ENDIF;
END.
```

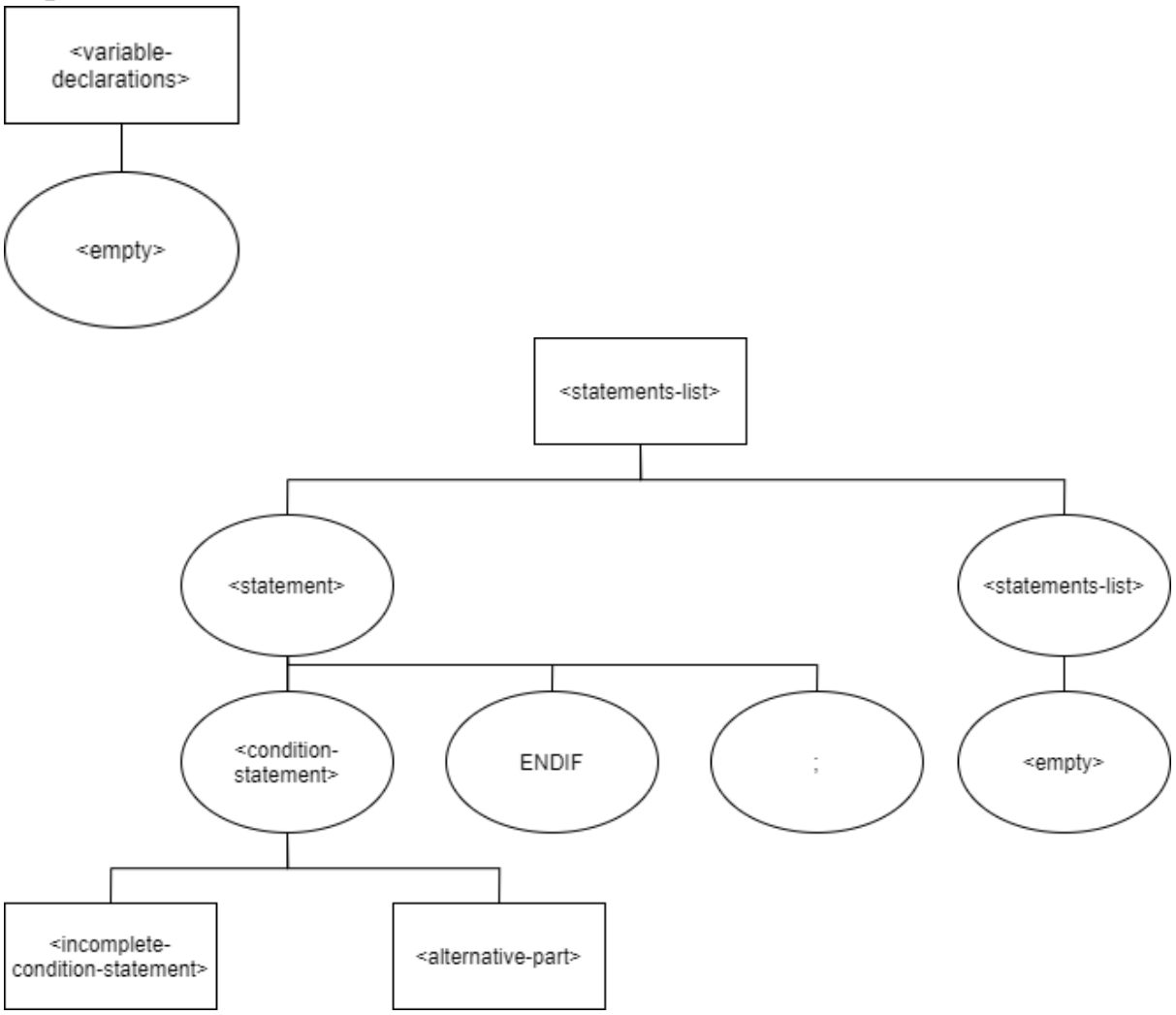
expected.txt:

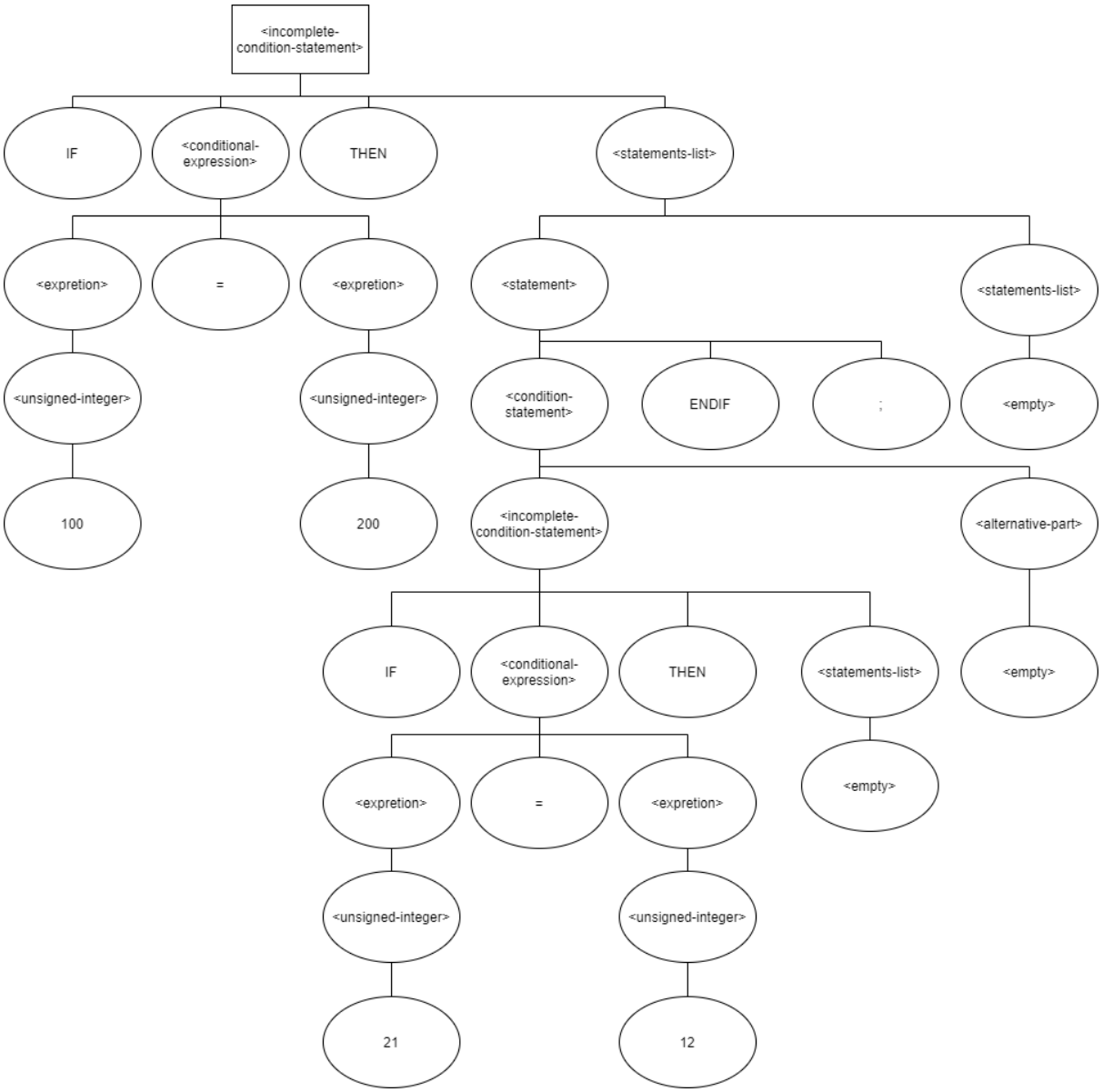
```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
3 2 408 IF
3 5 501 100
3 9 61 =
3 11 501 200
3 15 409 THEN
4 3 408 IF
4 6 502 21
4 9 61 =
4 11 503 12
4 14 409 THEN
5 3 407 ENDIF
5 8 59 ;
6 2 410 ELSE
7 3 408 IF
7 6 502 21
7 9 61 =
7 11 503 12
7 14 409 THEN
8 3 407 ENDIF
8 8 59 ;
9 2 407 ENDIF
9 7 59 ;
10 1 403 END
10 4 46 .

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
```

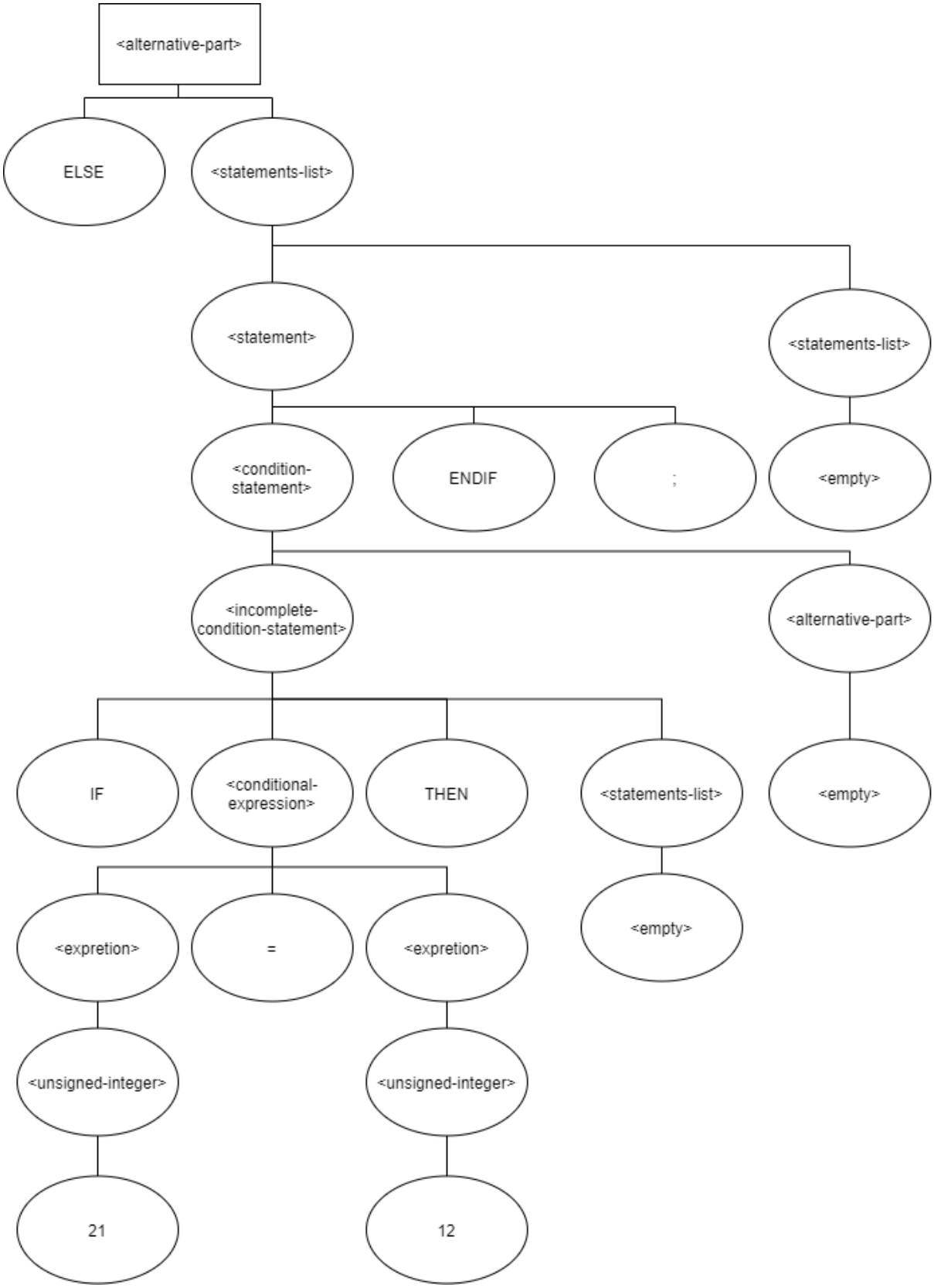
.....<identifier>
.....1001 PR
....59 ;
....<block>
.....<variable-declaration>
.....<empty>
.....402 BEGIN
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....501 100
.....61 =
.....<expretion>
.....<unsigned-integer>
.....501 200
.....409 THEN
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....502 21
.....61 =
.....<expretion>
.....<unsigned-integer>
.....503 12
.....409 THEN
.....<statements-list>
.....<empty>

.....<alternative-part>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....502 21
.....61 =
.....<expretion>
.....<unsigned-integer>
.....503 12
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....403 END
....46 .

Дерево:


<variable-declarations>
<empty>
<statements-list>
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<empty>
<incomplete-condition-statement>
<alternative-part>


<incomplete-condition-statement>
IF
<conditional-expression>
THEN
<statements-list>
<expretion>
=
<expretion>
<statement>
<statements-list>
<unsigned-integer>
<unsigned-integer>
<condition-statement>
ENDIF
;
<empty>
100
200
<incomplete-condition-statement>
<alternative-part>
IF
<conditional-expression>
THEN
<statements-list>
<empty>
<expretion>
=
<expretion>
<empty>
<unsigned-integer>
<unsigned-integer>
21
12


<alternative-part>
ELSE
<statements-list>
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<empty>
<incomplete-condition-statement>
<alternative-part>
IF
<conditional-expression>
THEN
<statements-list>
<empty>
<expretion>
=
<expretion>
<empty>
<unsigned-integer>
<unsigned-integer>
21
12

Приклад №5

input.sig:

```
PROGRAM PR;
	VAR V1:INTEGER;
BEGIN
	IF 100 = V1 THEN
		IF 12 = 12 THEN
		ENDIF;
	ELSE
	ENDIF;
	IF 100 = V1 THEN
	ELSE
		IF 12 = 12 THEN
		ENDIF;
	ENDIF;
END.
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
2 16 59 ;
3 1 402 BEGIN
4 2 408 IF
4 5 501 100
4 9 61 =
4 11 1002 V1
4 14 409 THEN
5 3 408 IF
5 6 503 12
5 9 61 =
5 11 503 12
5 14 409 THEN
6 3 407 ENDIF
```

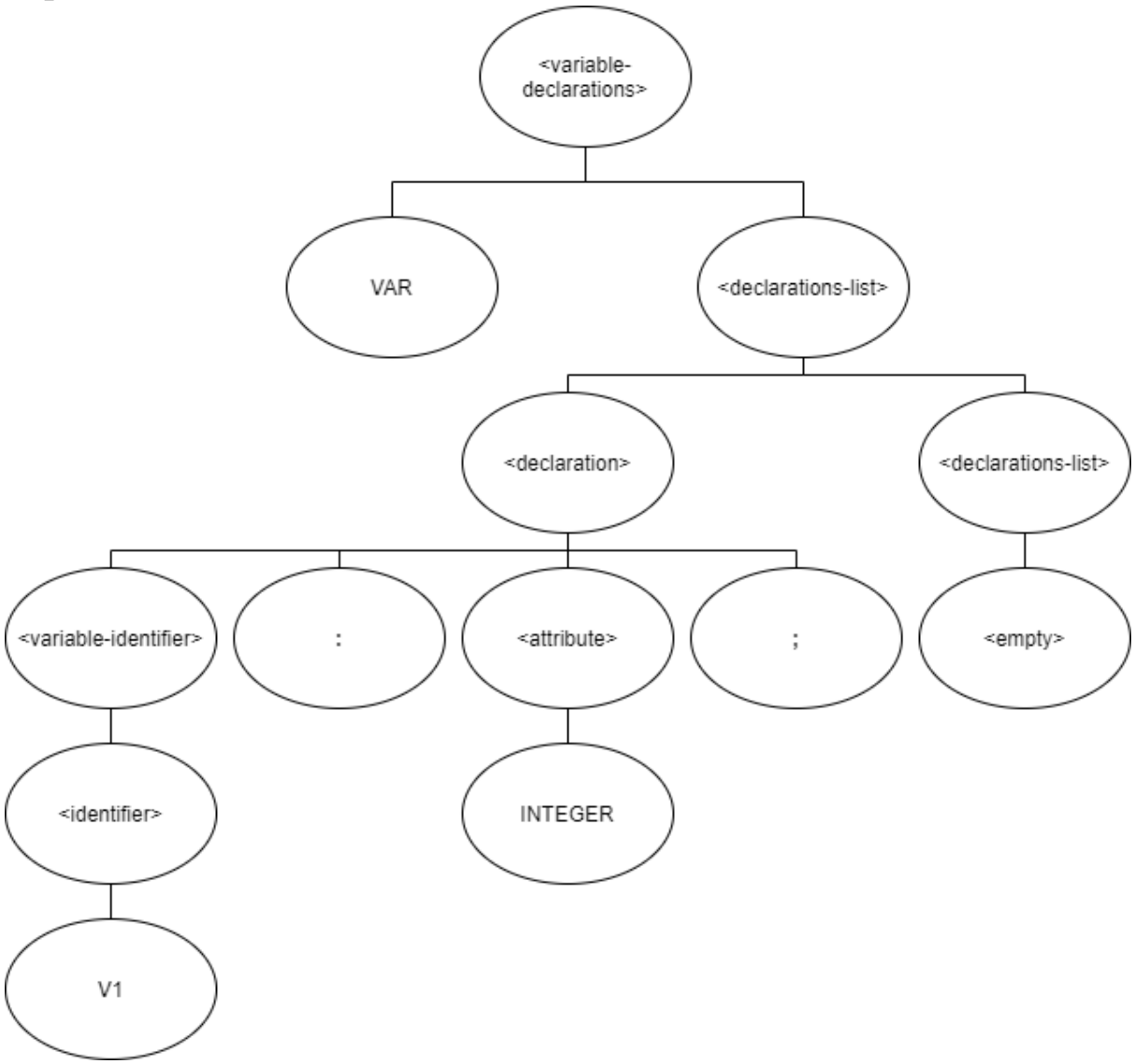
```
6 8 59 ;
7 2 410 ELSE
8 2 407 ENDIF
8 7 59 ;
9 2 408 IF
9 5 501 100
9 9 61 =
9 11 1002 V1
9 14 409 THEN
10 2 410 ELSE
11 3 408 IF
11 6 503 12
11 9 61 =
11 11 503 12
11 14 409 THEN
12 3 407 ENDIF
12 8 59 ;
13 2 407 ENDIF
13 7 59 ;
14 1 403 END
14 4 46 .

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
......<variable-declaration>
........404 VAR
........<declarations-list>
..........<declaration>
............<variable-identifier>
..............<identifier>
................1002 V1
```

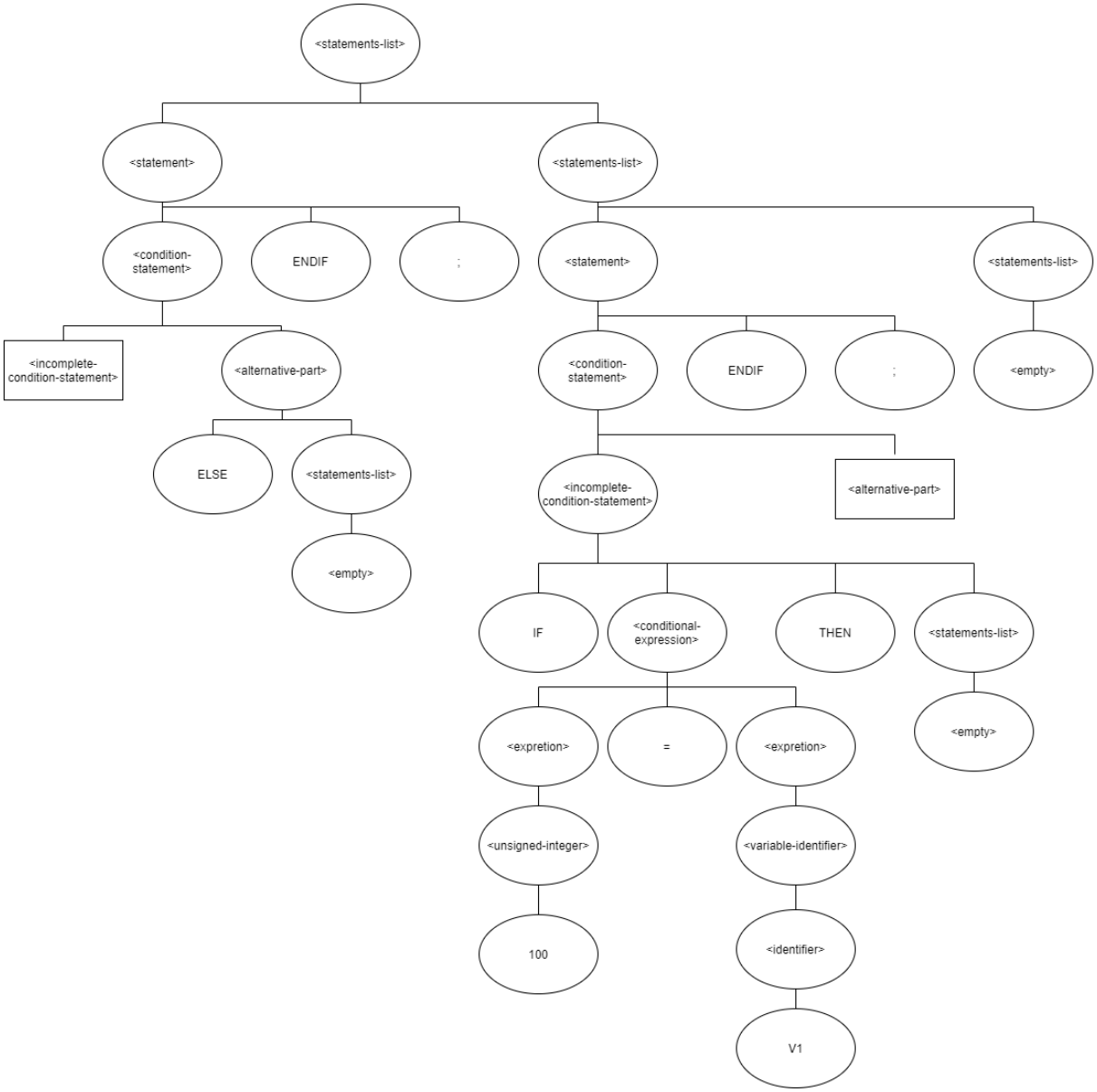
.....58 :
.....<attribute>
.....405 INTEGER
.....59 ;
.....<declarations-list>
.....<empty>
.....402 BEGIN
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....501 100
.....61 =
.....<expretion>
.....<variable-identifier>
.....<identifier>
.....1002 V1
.....409 THEN
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....503 12
.....61 =
.....<expretion>
.....<unsigned-integer>
.....503 12
.....409 THEN
.....<statements-list>

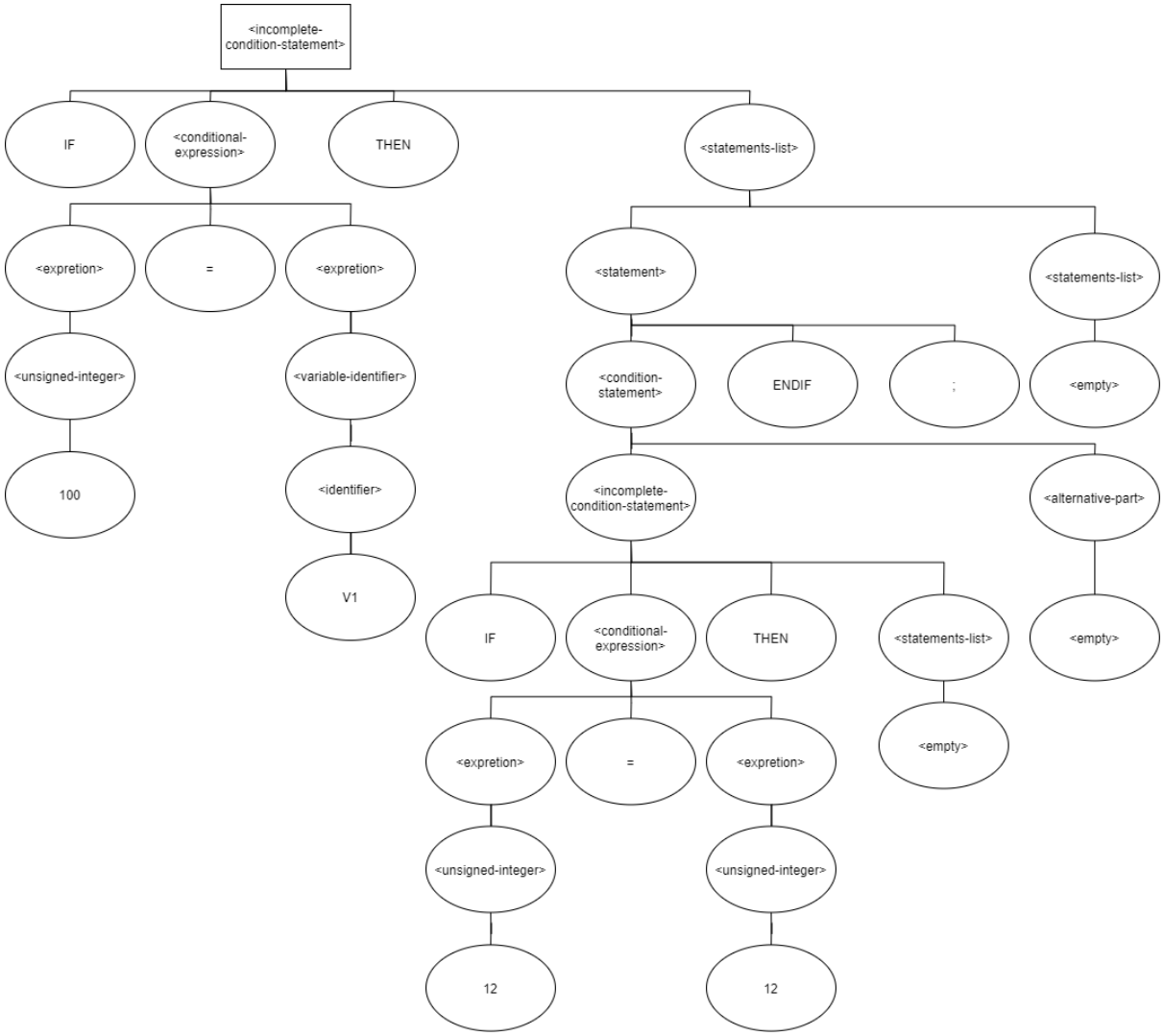
.....<empty>
.....<alternative-part>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....501 100
.....61 =
.....<expretion>
.....<variable-identifier>
.....<identifier>
.....1002 V1
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF

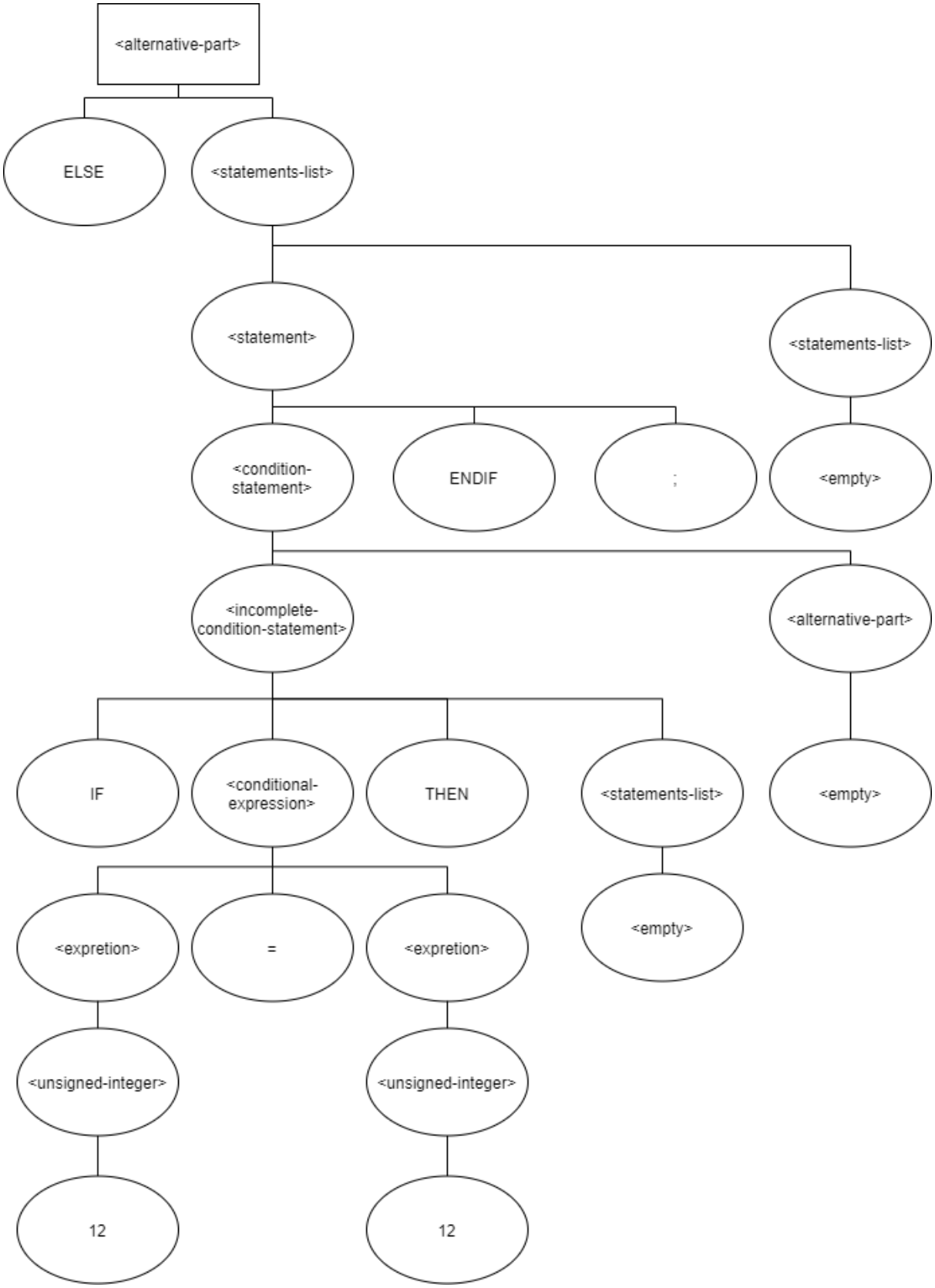
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....503 12
.....61 =
.....<expretion>
.....<unsigned-integer>
.....503 12
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....403 END
....46 .

Дерево:


<variable-declarations>
VAR
<declarations-list>
<declaration>
<declarations-list>
<variable-identifier>
:
<attribute>
;
<empty>
<identifier>
INTEGER
V1


<statements-list>
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<statement>
<statements-list>
<incomplete-condition-statement>
<alternative-part>
<condition-statement>
ENDIF
;
<empty>
ELSE
<statements-list>
<incomplete-condition-statement>
<alternative-part>
<empty>
IF
<conditional-expression>
THEN
<statements-list>
<expretion>
=
<expretion>
<empty>
<unsigned-integer>
<variable-identifier>
100
<identifier>
V1


<incomplete-condition-statement>
IF
<conditional-expression>
THEN
<statements-list>
<expretion>
=
<expretion>
<unsigned-integer>
100
<variable-identifier>
<identifier>
V1
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<empty>
<incomplete-condition-statement>
<alternative-part>
IF
<conditional-expression>
THEN
<statements-list>
<empty>
<empty>
<expretion>
=
<expretion>
<unsigned-integer>
<unsigned-integer>
12
12


<alternative-part>
ELSE
<statements-list>
<statement>
<statements-list>
<condition-statement>
ENDIF
;
<empty>
<incomplete-condition-statement>
<alternative-part>
IF
<conditional-expression>
THEN
<statements-list>
<empty>
<expretion>
=
<expretion>
<empty>
<unsigned-integer>
<unsigned-integer>
12
12

Приклад №6

input.sig:

```
PROGRAM PR;
        VAR V1:INTEGER;
BEGIN
        IF 100 = V2 THEN
        ELS
        ENDIF;
END.
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
2 16 59 ;
3 1 402 BEGIN
4 2 408 IF
4 5 501 100
4 9 61 =
4 11 1003 V2
4 14 409 THEN
5 2 1001 ELS
6 2 407 ENDIF
6 7 59 ;
7 1 403 END
7 4 46 .

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
```

....<block>
......<variable-declaration>
.......404 VAR
.......<declarations-list>
.........<declaration>
...........<variable-identifier>
.............<identifier>
...............1002 V1
.........58 :
.........<attribute>
...........405 INTEGER
.........59 ;
.........<declarations-list>
...........<empty>
......402 BEGIN
......<statements-list>
.......<statement>
.........<condition-statement>
...........<incomplete-condition-statement>
.............408 IF
.............<conditional-expression>
...............<expretion>
.................<unsigned-integer>
...................501 100
...............61 =
...............<expretion>
.................<variable-identifier>
...................<identifier>
.....................1003 V2
.............409 THEN
.............<statements-list>
...............<empty>
...........<alternative-part>
.............<empty>
Parser: Error(line 5, column 2):Found 'ELS' Expected 'ENDIF'

Приклад №7

input.sig:

```
PROGRAM PR;
        VAR V1:INTEGER;
BEGIN
        IF 100 = V2 THEN
        ELSE
        ENDIF;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
2 16 59 ;
3 1 402 BEGIN
4 2 408 IF
4 5 501 100
4 9 61 =
4 11 1003 V2
4 14 409 THEN
5 2 410 ELSE
6 2 407 ENDIF
6 7 59 ;
7 1 403 END

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
```

.....<variable-declaration>
.....404 VAR
.....<declarations-list>
.....<declaration>
.....<variable-identifier>
.....<identifier>
.....1002 V1
.....58 :
.....<attribute>
.....405 INTEGER
.....59 ;
.....<declarations-list>
.....<empty>
.....402 BEGIN
.....<statements-list>
.....<statement>
.....<condition-statement>
.....<incomplete-condition-statement>
.....408 IF
.....<conditional-expression>
.....<expretion>
.....<unsigned-integer>
.....501 100
.....61 =
.....<expretion>
.....<variable-identifier>
.....<identifier>
.....1003 V2
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;

.....<statements-list>
.....<empty>
.....403 END
Parser: Error(line 7, column 1):File end Expected '.'

Приклад №8

input.sig:

```
PROGRAM PR;
        VAR V1:INTEGER;
BEGIN
        IF 100 = V2 THEN
        ELSE
        ENDIF;
END.
IF 100 = 21 THEN
ENDIF;
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 2 404 VAR
2 6 1002 V1
2 8 58 :
2 9 405 INTEGER
2 16 59 ;
3 1 402 BEGIN
4 2 408 IF
4 5 501 100
4 9 61 =
4 11 1003 V2
4 14 409 THEN
5 2 410 ELSE
6 2 407 ENDIF
6 7 59 ;
7 1 403 END
7 4 46 .
8 1 408 IF
8 4 501 100
```

8 8 61 =
8 10 502 21
8 13 409 THEN
9 1 407 ENDIF
9 6 59 ;

<signal-program>
..<program>
....401 PROGRAM
....<procedure-identifier>
......<identifier>
........1001 PR
....59 ;
....<block>
......<variable-declaration>
........404 VAR
........<declarations-list>
..........<declaration>
............<variable-identifier>
..............<identifier>
................1002 V1
..........58 :
..........<attribute>
............405 INTEGER
..........59 ;
..........<declarations-list>
............<empty>
......402 BEGIN
......<statements-list>
........<statement>
..........<condition-statement>
............<incomplete-condition-statement>
..............408 IF
..............<conditional-expression>
................<expretion>
..................<unsigned-integer>
....................501 100

.....61 =
.....<expretion>
.....<variable-identifier>
.....<identifier>
.....1003 V2
.....409 THEN
.....<statements-list>
.....<empty>
.....<alternative-part>
.....410 ELSE
.....<statements-list>
.....<empty>
.....407 ENDIF
.....59 ;
.....<statements-list>
.....<empty>
.....403 END
....46 .
Parser: Error(line 8, column 1):Found 'IF' Expected end of file

Приклад №9

input.sig(empty file):

expected.txt:

Lexer: Error: Empty file

Empty tree!