

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії**

Лабораторна робота № 2

**Лексичний аналіз методом діаграми станів
(з дисципліни «Побудова компіляторів»)**

Учасники проєкту:

Пашковський Євгеній Сергійович

Звіт студента I курсу, групи ІП-41мн
спеціальності 121 «Інженерія програмного забезпечення»
Пашковського Євгенія Сергійовича

(Прізвище, ім'я, по батькові)

(Підпис)

Перевірів: доцент, к.т.н. Стативка Ю.І.
(Посада, науковий ступінь, прізвище та ініціали)

(Підпис)

Київ – 2024

Зміст

1. Коротка характеристика мови.....	3
1.1. Обробка.....	3
1.2. Нотація.....	3
2. Лексична структура.....	5
2.1. Алфавіт.....	5
2.2. Спеціальні символи.....	5
2.3. Ідентифікатори.....	6
2.4. Константи.....	7
2.5. Ключові слова.....	9
2.6. Токени.....	9
3. Граматика мови.....	12
4. Графічна форма остаточної версії діаграми станів (JFLAP).....	14
5. Форма остаточної версії діаграми станів у коді:.....	14
6. Таблиця семантичних процедур.....	16
7. Приклад програми вхідною мовою для тестування лексичного аналізатора.....	17
8. План тестування.....	18
9. Протокол тестування у формі текстових копій термінала:.....	19
Як і очікувалось, лексичний аналізатор трактує помилково написані ключові слова як ідентифікатори.....	24
10. Висновки.....	24
11. Додаток А. Код лексичного аналізатора.....	24

1. Коротка характеристика мови

Представлена тут мова програмування KotlinScript – імперативна мова загального призначення. Прототип синтаксичних конструкцій – мова Kotlin. Назва промовляється як ”котлін скрипт”. Назва асоціюється з мовою Kotlin, яка виступає прототипом синтаксичних конструкцій для цієї мови.

1.1. Обробка

Програма, написана мовою KotlinScript, подається на вхід інтерпретатора для трансляції до цільової форми (мови). Результат трансляції виконується у системі часу виконання (run-time system), для чого приймає вхідні дані та надає результат виконання програми.

Трансляція передбачає фази лексичного, синтаксичного та семантичного аналізу, а також фазу генерації коду. Фаза лексичного аналізу здійснюється окремим проходом.

1.2. Нотація

Для опису мови KotlinScript використовується розширена нотація (форма) Бекуса-Наура, див. табл. 1.

Ланцюжки, що починаються з великої літери вважаються нетерміналами (нетермінальними символами). Термінали (термінальні символи) — ланцюжки, що починаються з маленької літери, або ланцюжки, що знаходяться між одинарними чи подвійними лапками. Для графічного представлення граматики використовуються синтаксичні діаграми Вірта.

Метасимвол	Значення
=	визначається як
	альтернатива
[x]	0 або 1 екземпляр x
{ x }	0 або більше екземплярів x
(x y)	групування: будь -який з x або y
Abc	нетермінал
abc	термінал
'1'	термінал (не літера)
”1”	термінал (не літера)
”+”	термінал (не літера)

Таблиця 1 – Прийнята нотація РБНФ

2. Лексична структура

Лексичний аналіз виконується окремим проходом, отже не залежить від синтаксичних та семантичних аспектів. Лексичний аналізатор розбиває текст програми на лексеми.

2.1. Алфавіт

Текст програми мовою KotlinScript може містити тільки такі символи (characters): літери, цифри та спеціальні знаки.

```
LowerLetter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
'k'
| 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
| 'x' | 'y' | 'z';
UpperLetter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
'K'
| 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W'
| 'X' | 'Y' | 'Z';
Letter = UpperLetter | LowerLetter;
NonZeroDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | NonZeroDigit;
SpecSign = '.' | ',' | ':' | ';' | '(' | ')'
| '=' | '+' | '-' | '*' | '/' | '<' | '>' | '!'
| WhiteSpace | EOL;
WhiteSpace = ' ' | '\t';
EOL = ['\r'], '\n';
```

2.2. Спеціальні символи

У програмі мовою KotlinScript використовуються лексеми, що класифікуються як спеціальні символи, ідентифікатори, беззнакові цілі константи, беззнакові дійсні константи, логічні константи та ключові слова.

```
SpecSymbols = ArithmOps | RelOps | BracketsOps | AssignOp | Punct;
ArithmOps = AddOps | MultOps;
AddOps = '+' | '-';
MultOps = '*' | '/' | '^';
```

```

RelOps = '===' | '==' | '<=' | '<' | '>' | '>=' | '!=';
BracketsOps = '(' | ')';
AssignOp = '=';
Punct = '.' | ',' | ':' | ';';
InlineCommentOp = '//';

```

2.3. Ідентифікатори

У мові KotlinScript ідентифікатори використовуються для позначення змінних, функцій, пакетів тощо.

Першим символом ідентифікатора може бути тільки літери, наступні символи, якщо вони є, можуть бути цифрами або літерами, або у формі правила:

```
Id = Letter, { Letter | Digit };
```

Довжина ідентифікатора не обмежена.

Жоден ідентифікатор не може збігатись із ключовим (вбудованим, зарезервованим) словом або словами true, false. Якщо у фазі лексичного аналізу визначена лексема, яка синтаксично могла би бути ідентифікатором, але збігається із ключовим словом, то вона вважається ключовим словом. Якщо збігається з словами true чи false, то вважається значенням логічної константи.

Діаграма Вірта ідентифікатора представлена на рис. 2.1.

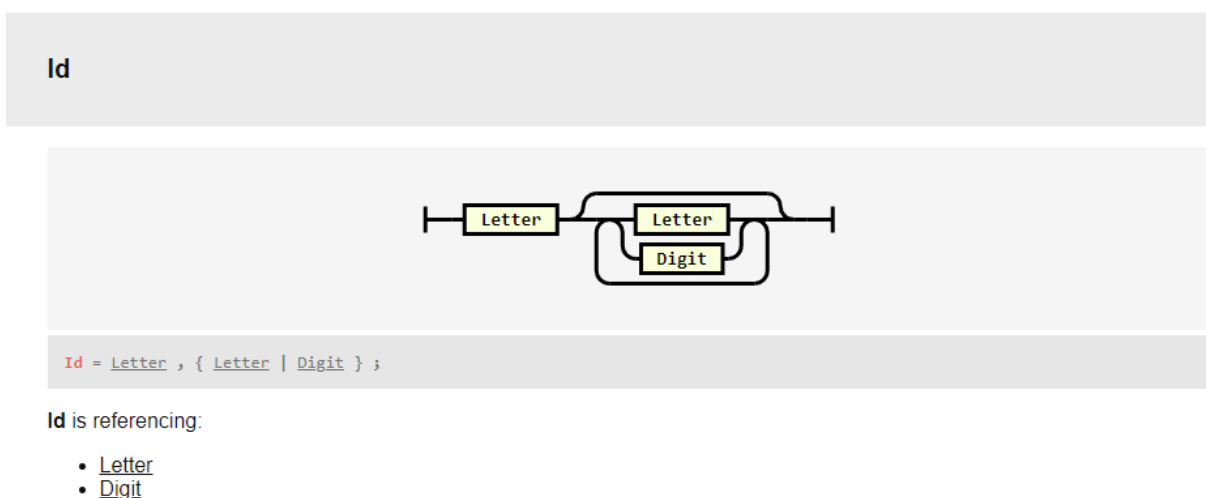


Рисунок 2.1 – Синтаксична діаграма ідентифікатора

Приклади:

b, o3, var11

2.4. Константи

Літерали використовуються для представлення постійних значень: цілих числових констант `IntLiteral`, дійсних числових констант `RealLiteral` та логічних констант `BoolLiteral`.

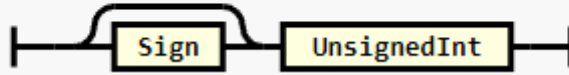
Їх синтаксис:

```
Literal = IntLiteral | RealLiteral | BoolLiteral;
IntLiteral = [Sign], UnsignedInt;
RealLiteral = [Sign], UnsignedReal;
Sign = '+' | '-';
UnsignedInt = Digit, {Digit};
UnsignedReal = ('.', UnsignedInt)
| (UnsignedInt, '.')
| (UnsignedInt, '.', UnsignedInt);
BoolLiteral = 'true' | 'false';
```

Тип кожної константи визначається її формою, а значення має знаходитись у діапазоні репрезентативних значень для її типу.

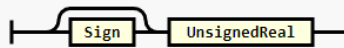
На етапі лексичного аналізу виявляються тільки беззнакові цілі константи `UnsignedInt`, беззнакові дійсні константи `UnsignedReal` та логічні константи `BoolLiteral`. Унарні мінус чи плюс розпізнаються при синтаксичному розборі. Синтаксичні діаграми констант представлені на рис. 2.2.

IntLiteral



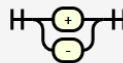
```
IntLiteral = [ Sign ], UnsignedInt ;
```

RealLiteral



```
RealLiteral = [ Sign ], UnsignedReal ;
```

Sign

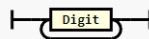


```
Sign = "+" | "-" ;
```

Items referencing **Sign**:

- [IntLiteral](#)
- [RealLiteral](#)
- [ArithmExpression](#)

UnsignedInt



```
UnsignedInt = { Digit }- ;
```

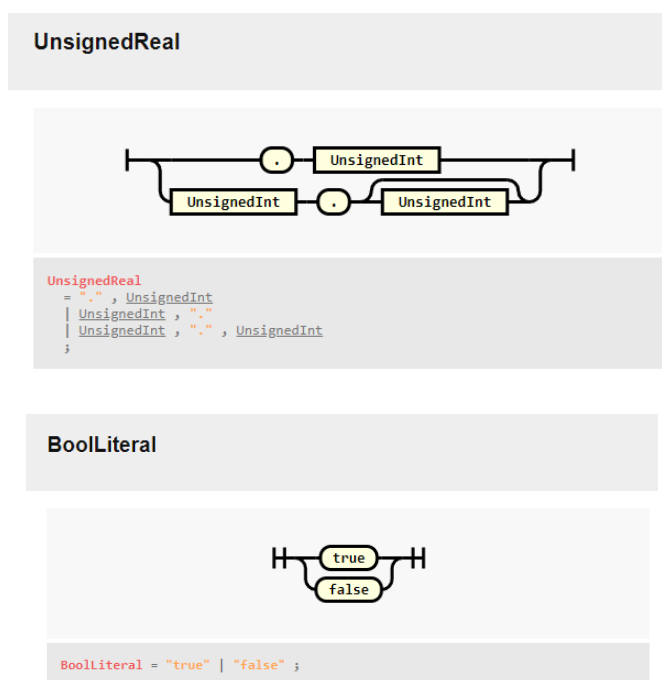



Рисунок 2.2 – Константи

Приклади констант:

12, 234, 1.54, 34.567, 23., true, false

2.5. Ключові слова

Мова KotlinScript містить такі ключові слова:

fun, var, while, if, else, Int, Float, package

2.6. Токени

З потоку символів вхідної програми на етапі лексичного аналізу виокремлюються послідовності символів з певним сукупним значенням – токени. Список.tokenів мови KotlinScript див. у табл. 2.1.

Єдиний приклад засвідчує унікальність лексеми. Текст "символ ...", наприклад "символ for", означає, що вказаний символ (symbol) є терміналом.

Код	Приклади лексем	Токен	Неформальний опис
1	a, r1, b1s1	id	ідентифікатор
2	123, 0, 521	int_literal	ціле число
3	.012, 34.76, 876.	real_literal	дійсне число
4	true	bool_literal	логічне значення
5	false	bool_literal	логічне значення
6	fun	keyword	символ fun
7	var	keyword	символ var
8	while	keyword	символ while
9	if	keyword	символ if
10	else	keyword	символ else
11	Int	keyword	символ Int
12	Float	keyword	символ Float
13	package	keyword	символ package
14	=	assign_op	символ =
15	+	add_op	символ +
16	-	add_op	символ -
17	*	mult_op	символ *
18	/	mult_op	символ /
19	^	mult_op	символ ^
20	<	rel_op	символ <
21	<=	rel_op	символ <=
22	==	rel_op	символ ==
23	===	rel_op	символ ===

24	>=	rel_op	символ >=
25	!=	rel_op	символ !=
26	(brackets_op	символ (
27)	brackets_op	символ)
28	.	punct	символ .
29	,	punct	символ ,
30	:	punct	символ :
31	;	punct	символ ;
32	\32	ws	пробільні символи (пробіл)
33	\t	ws	пробільні символи (г. табуляція)
34	\n	eol	роздільник рядків (новий рядок)
35	\r\n	eol	роздільник рядків

Таблиця 2 – Таблиця лексем мови KotlinScript

3. Граматика мови

```

LowerLetter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
'k'
| 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
| 'x' | 'y' | 'z';
UpperLetter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
'K'
| 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W'
| 'X' | 'Y' | 'Z';
Letter = UpperLetter | LowerLetter;
NonZeroDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | NonZeroDigit;
SpecSign = '.' | ',' | ':' | ';' | '(' | ')'
| '=' | '+' | '-' | '*' | '/' | '<' | '>' | '!'
| WhiteSpace | EOL;
WhiteSpace = ' ' | '\t';
EOL = ['\r'], '\n';

```

```

SpecSymbols = ArithmOps | RelOps | BracketsOps | AssignOp | Punct;
ArithmOps = AddOps | MultOps;
AddOps = '+' | '-';
MultOps = '*' | '/' | '^';
RelOps = '===' | '==' | '<=' | '<' | '>' | '>=' | '!=';
BracketsOps = '(' | ')';
AssignOp = '=';
Punct = '.' | ',' | ':' | ';';
InlineCommentOp = '//';

```

```

Id = Letter, { Letter | Digit };
IdList = Id, {' ', Id };

```

```

Literal = IntLiteral | RealLiteral | BoolLiteral;
IntLiteral = [Sign], UnsignedInt;
RealLiteral = [Sign], UnsignedReal;
Sign = '+' | '-';
UnsignedInt = Digit, {Digit};

```

```

UnsignedReal = ('.', UnsignedInt)
| (UnsignedInt, '.')
| (UnsignedInt, '.', UnsignedInt);
BoolLiteral = 'true' | 'false';

Keyword = 'fun' | 'var' | 'while' | 'if' | 'else' | 'package' | Type;

Type = 'Int' | 'Float' | 'Bool';
VarTypeSpecifier = ':', Type;

VarDeclaration = Id, [VarTypeSpecifier];
VarDeclarationsList = VarDeclaration, { ',', VarDeclaration };

Expression = ArithmExpression | BoolExpression;

ArithmExpression = [Sign], Term
| (ArithmExpression, '+', Term)
| (ArithmExpression, '-', Term);

Term = Factor
| (Term, '*', Factor)
| (Term, '/', Factor)
| (Term, '^', Factor);

Factor = Id
| Literal
| ('(', ArithmExpression, ')');

BoolExpression = (Expression, RelOps, Expression) | BoolLiteral;

Block = '{', [StatementsList], '}';
StatementsList = Statement, {';', Statement};

Statement = AssignStatement | InStatement | OutStatement | WhileStatement |
IfStatement;

AssignStatement = ['var'], VarDeclaration, AssignOp, Expression;

```

```
InStatement='var', VarDeclaration, AssignOp, 'readln', '(', ')';
```

```
OutStatement='print','(', Expression , ')';
```

```
WhileStatement='while', '(', Expression , ')', Block;
```

```
IfStatement='if', '(', Expression, ')', Block, ['else', (IfStatement | Block)];
```

```
PackageId = Id, { '.', Id };
```

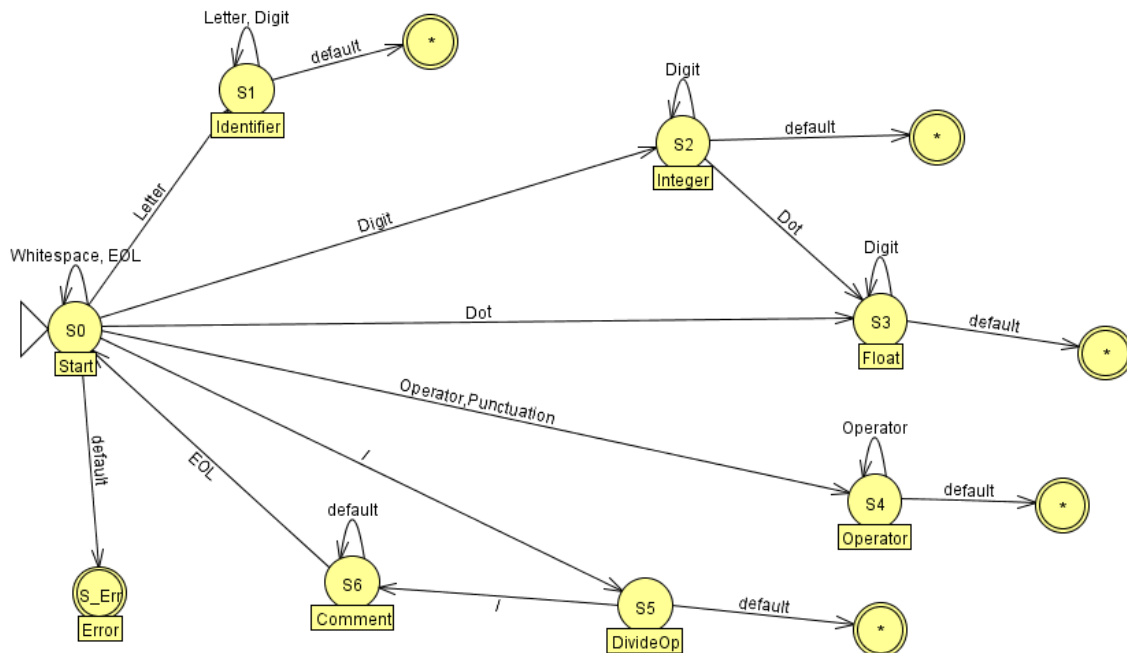
```
PackageDeclaration = 'package', PackageId, ';';
```

```
EntryPointFunName = 'main';
```

```
EntryPoint = 'fun', EntryPointFunName, '(', [VarDeclarationsList], ')', Block;
```

```
Grammar = PackageDeclaration, EntryPoint;
```

4. Графічна форма остаточної версії діаграми станів (JFLAP)



5. Форма остаточної версії діаграми станів у коді:

```
enum CharClass {
```

```

    LETTER = "Letter",
    DIGIT = "Digit",
    DOT = "Dot",
    WHITESPACE = "Whitespace",
    EOL = "EOL",
    OPERATOR = "Operator",
    PUNCTUATION = "Punctuation",
    SLASH = "Slash",
    OTHER = "Other",
}

enum State {
    START = "S0",
    IDENTIFIER = "S1",
    INTEGER = "S2",
    FLOAT = "S3",
    OPERATOR = "S4",
    DIVIDE_OP = "S5",
    COMMENT = "S6",
    ERROR = "S_Err",
}

const transitionTable: Record<State, Record<string, State | undefined>>
= {
    [State.START]: {
        [CharClass.LETTER]: State.IDENTIFIER,
        [CharClass.DIGIT]: State.INTEGER,
        [CharClass.DOT]: State.FLOAT,
        [CharClass.WHITESPACE]: State.START,
        [CharClass.EOL]: State.START,
        [CharClass.OPERATOR]: State.OPERATOR,
        [CharClass.PUNCTUATION]: State.OPERATOR,
        [CharClass.SLASH]: State.DIVIDE_OP,
    },
    [State.IDENTIFIER]: {
        [CharClass.LETTER]: State.IDENTIFIER,
        [CharClass.DIGIT]: State.IDENTIFIER,
        default: State.START,
    },
    [State.INTEGER]: {
        [CharClass.DIGIT]: State.INTEGER,
        [CharClass.DOT]: State.FLOAT,
    },

```

```

    default: State.START,
  },
  [State.FLOAT]: {
    [CharClass.DIGIT]: State.FLOAT,
    default: State.START,
  },
  [State.OPERATOR]: {
    [CharClass.OPERATOR]: State.OPERATOR,
    default: State.START,
  },
  [State.DIVIDE_OP]: {
    [CharClass.SLASH]: State.COMMENT,
    default: State.START,
  },
  [State.COMMENT]: {
    default: State.COMMENT,
    [CharClass.EOL]: State.START,
  },
  [State.ERROR]: {},
};

```

6. Таблиця семантичних процедур

Стан	Токен	Семантичні процедури
S1	id або keyword	<p>Визначити: id чи keyword?</p> <p>Якщо id – обробити таблицю ідентифікаторів</p> <p>Занести лексему в таблицю розбору</p> <p>Повернути необроблений символ у вхідний потік</p> <p>Перейти у стартовий стан</p>
S2	Integer	<p>Обробити таблицю констант</p> <p>Занести лексему в таблицю розбору</p> <p>Повернути необроблений символ у вхідний потік</p> <p>Перейти у стартовий стан</p>

S3	Float	Обробити таблицю констант Занести лексему в таблицю розбору Повернути необроблений символ у вхідний потік Перейти у стартовий стан
S4	operator, punctuation	Визначити: operator чи punctuation Якщо operator, занести лексему в таблицю розбору Перейти у стартовий стан
S5	divideOp	занести лексему в таблицю розбору перейти у стартовий стан повернути необроблений символ у вхідний потік
S6	comment	Пропустити рядок

7. Приклад програми вхідною мовою для тестування лексичного аналізатора

Базовий приклад з вкладеностями:

```
package com.example.lab;

fun main() {
    var a = 1;
    var b = 2;
    var c = a + b;

    var d: Int;

    d = 3;

    d = d * a;

    var e: Float = .2;

    e = e / 2 - 1;

    var readNum: Int = readln();

    var f = -1 + e * (a ^ c) + b * (1.2 + a * (readNum - 1));

    // this is variable i
    var i = 0;
    var l = 2;
    while(i < 3) {
```

```

if (l > 410) {
    l = l / 2;
} else if (l <= 0) {
    l = l + 1;
} else if (l >= 20) {
    l = l + 12;
} else {
    l = l * 2;
}

```

```

if (i == true) {
    if (false == 0) {
        print(true);
    }
}

```

```

if (i === 2) {
} else if (i != 1) {
    print(f + i);
}

```

```

    i = i + 1;
}
}

```

Наявність недопустимих лексем:

```
fun main() {}
```

```
// this should cause error
&
```

Помилково написані ключові слова:

```
package com.example.lab;
```

```
fun main() {
    iff(true) {
        vaar i = 1;
    }
}

```

8. План тестування

№	Тип випробування	Очікуваний результат	Кількість випробувань
1	базовий приклад вкладеностями	з успішне виконання, таблиці	1

		ідентифікаторів, констант	
2	наявність недопустимих лексем	повідомлення про помилку	1
3	помилково написані ключові слова	ключові слова розпізнаються як ідентифікатори	1

9. Протокол тестування у формі текстових копій терміналу:

9.1. Базовий приклад з вкладеностями

```
PS      E:\Магістр\semestr    1\compilers\lab2>    node    .\build\lexer.js
.\code-example1.txt
№      Рядок    Лексема      Токен      Індекс
1      1      package     keyword     -
2      1      com         identifier  1
3      1      example     identifier  2
4      1      lab         identifier  3
5      1      ;           punctuation -
6      3      fun         keyword     -
7      3      main        identifier  4
8      3      (           bracket_open -
9      3      )           bracket_close -
10     3      {           punctuation -
11     4      var         keyword     -
12     4      a           identifier  5
13     4      =           operator    -
14     4      1           int_literal 1
15     4      ;           punctuation -
16     5      var         keyword     -
17     5      b           identifier  6
18     5      =           operator    -
19     5      2           int_literal 2
20     5      ;           punctuation -
21     6      var         keyword     -
22     6      c           identifier  7
23     6      =           operator    -
24     6      a           identifier  5
25     6      +           operator    -
26     6      b           identifier  6
27     6      ;           punctuation -
28     8      var         keyword     -
29     8      d           identifier  8
30     8      :           punctuation -
31     8      Int         keyword     -
32     8      ;           punctuation -
33     10     d           identifier  8
```

34	10	=	operator	-
35	10	3	int_literal	3
36	10	;	punctuation	-
37	12	d	identifier	8
38	12	=	operator	-
39	12	d	identifier	8
40	12	*	operator	-
41	12	a	identifier	5
42	12	;	punctuation	-
43	14	var	keyword	-
44	14	e	identifier	9
45	14	:	punctuation	-
46	14	Float	keyword	-
47	14	=	operator	-
48	14	.2	float_literal	4
49	14	;	punctuation	-
50	16	e	identifier	9
51	16	=	operator	-
52	16	e	identifier	9
53	16	/	divideOp	-
54	16	2	int_literal	2
55	16	-	operator	-
56	16	1	int_literal	1
57	16	;	punctuation	-
58	18	var	keyword	-
59	18	readNum	identifier	10
60	18	:	punctuation	-
61	18	Int	keyword	-
62	18	=	operator	-
63	18	readln	identifier	11
64	18	(bracket_open	-
65	18)	bracket_close	-
66	18	;	punctuation	-
67	20	var	keyword	-
68	20	f	identifier	12
69	20	=	operator	-
70	20	-	operator	-
71	20	+	operator	-
72	20	e	identifier	9
73	20	*	operator	-
74	20	(bracket_open	-
75	20	a	identifier	5
76	20	^	operator	-
77	20	c	identifier	7
78	20)	bracket_close	-
79	20	+	operator	-
80	20	b	identifier	6
81	20	*	operator	-
82	20	(bracket_open	-
83	20	1.2	float_literal	5
84	20	+	operator	-
85	20	a	identifier	5
86	20	*	operator	-
87	20	(bracket_open	-
88	20	readNum	identifier	10
89	20	-	operator	-
90	20	1	int_literal	1
91	20)	bracket_close	-
92	20)	bracket_close	-

93	20	;	punctuation	-	
94	23	var	keyword	-	
95	23	i	identifier		13
96	23	=	operator		-
97	23	0	int_literal		6
98	23	;	punctuation		-
99	24	var	keyword	-	
100	24	l	identifier		14
101	24	=	operator		-
102	24	2	int_literal		2
103	24	;	punctuation		-
104	25	while	keyword	-	
105	25	(bracket_open		-
106	25	i	identifier		13
107	25	<	operator		-
108	25	3	int_literal		3
109	25)	bracket_close		-
110	25	{	punctuation		-
111	26	if	keyword	-	
112	26	(bracket_open		-
113	26	l	identifier		14
114	26	>	operator		-
115	26	410	int_literal		7
116	26)	bracket_close		-
117	26	{	punctuation		-
118	27	l	identifier		14
119	27	=	operator		-
120	27	l	identifier		14
121	27	/	divideOp		-
122	27	2	int_literal		2
123	27	;	punctuation		-
124	28	}	punctuation		-
125	28	else	keyword	-	
126	28	if	keyword	-	
127	28	(bracket_open		-
128	28	l	identifier		14
129	28	<=	operator		-
130	28	0	int_literal		6
131	28)	bracket_close		-
132	28	{	punctuation		-
133	29	l	identifier		14
134	29	=	operator		-
135	29	l	identifier		14
136	29	+	operator		-
137	29	1	int_literal		1
138	29	;	punctuation		-
139	30	}	punctuation		-
140	30	else	keyword	-	
141	30	if	keyword	-	
142	30	(bracket_open		-
143	30	l	identifier		14
144	30	>=	operator		-
145	30	20	int_literal		8
146	30)	bracket_close		-
147	30	{	punctuation		-
148	31	l	identifier		14
149	31	=	operator		-
150	31	l	identifier		14
151	31	+	operator		-

152	31	12	int_literal	9
153	31	;	punctuation	-
154	32	}	punctuation	-
155	32	else	keyword	-
156	32	{	punctuation	-
157	33	1	identifier	14
158	33	=	operator	-
159	33	1	identifier	14
160	33	*	operator	-
161	33	2	int_literal	2
162	33	;	punctuation	-
163	34	}	punctuation	-
164	36	if	keyword	-
165	36	(bracket_open	-
166	36	i	identifier	13
167	36	==	operator	-
168	36	true	keyword	-
169	36)	bracket_close	-
170	36	{	punctuation	-
171	37	if	keyword	-
172	37	(bracket_open	-
173	37	false	keyword	-
174	37	==	operator	-
175	37	0	int_literal	6
176	37)	bracket_close	-
177	37	{	punctuation	-
178	38	print	identifier	15
179	38	(bracket_open	-
180	38	true	keyword	-
181	38)	bracket_close	-
182	38	;	punctuation	-
183	39	}	punctuation	-
184	40	}	punctuation	-
185	42	if	keyword	-
186	42	(bracket_open	-
187	42	i	identifier	13
188	42	===	operator	-
189	42	2	int_literal	2
190	42)	bracket_close	-
191	42	{	punctuation	-
192	43	}	punctuation	-
193	43	else	keyword	-
194	43	if	keyword	-
195	43	(bracket_open	-
196	43	i	identifier	13
197	43	!=	operator	-
198	43	1	int_literal	1
199	43)	bracket_close	-
200	43	{	punctuation	-
201	44	print	identifier	15
202	44	(bracket_open	-
203	44	f	identifier	12
204	44	+	operator	-
205	44	i	identifier	13
206	44)	bracket_close	-
207	44	;	punctuation	-
208	45	}	punctuation	-
209	47	i	identifier	13
210	47	=	operator	-

211	47	i	identifier	13
212	47	+	operator	-
213	47	1	int_literal	1
214	47	;	punctuation	-
215	48	}	punctuation	-
216	49	}	punctuation	-

Таблиця ідентифікаторів:

Індекс	Ідентифікатор
1	com
2	example
3	lab
4	main
5	a
6	b
7	c
8	d
9	e
10	readNum
11	readln
12	f
13	i
14	l
15	print

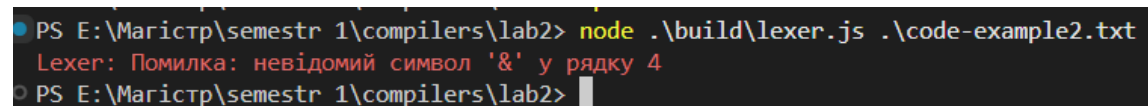
Таблиця констант:

Індекс	Константа
6	0
1	1
2	2
3	3
9	12
8	20
7	410
4	.2
5	1.2

Lexer: Лексичний аналіз завершено успішно

Як бачимо, лексичний аналізатор успішно розпізнав усі необхідні лексеми та правильно сформував таблиці ідентифікаторів та констант.

9.2. Наявність недопустимих лексем



```
PS E:\Marictp\semestr 1\compilers\lab2> node .\build\lexer.js .\code-example2.txt
Lexer: Помилка: невідомий символ '&' у рядку 4
PS E:\Marictp\semestr 1\compilers\lab2>
```

Як бачимо, аналізатор обробив помилку щодо недопустимої лексеми і відобразив її в терміналі, при цьому правильно вказавши рядок, на якому знаходиться цей символ.

9.3. Помилково написані ключові слова

```

PS      E:\Магістр\semestr 1\compilers\lab2> node .\build\lexer.js
.\code-example3.txt
№      Рядок   Лексема      Токен      Індекс
1      1      package     keyword     -
2      1      com         identifier  1
3      1      example    identifier  2
4      1      lab        identifier  3
5      1      ;          punctuation -
6      3      fu1n       identifier  4
7      3      main       identifier  5
8      3      (          bracket_open -
9      3      )          bracket_close -
10     3      {          punctuation -
11     4      iff        identifier  6
12     4      (          bracket_open -
13     4      true       keyword     -
14     4      )          bracket_close -
15     4      {          punctuation -
16     5      vaar       identifier  7
17     5      i          identifier  8
18     5      =          operator   -
19     5      1          int_literal 1
20     5      ;          punctuation -
21     6      }          punctuation -
22     7      }          punctuation -

```

Таблиця ідентифікаторів:

Індекс	Ідентифікатор
1	com
2	example
3	lab
4	fu1n
5	main
6	iff
7	vaar
8	i

Таблиця констант:

Індекс	Константа
1	1

Lexer: Лексичний аналіз завершено успішно

Як і очікувалось, лексичний аналізатор трактує помилково написані ключові слова як ідентифікатори.

10. Висновки

Під час виконання цієї лабораторної роботи було спроектовано та побудовано лексичний аналізатор методом діаграми станів, а також протестовано його роботу. Отриманий лексичний аналізатор відповідає поставленим до нього вимогам і правильно визначає лексеми, токени,

правильно будує таблиці ідентифікаторів і констант, оброблює помилку щодо недопустимої лексеми та вміє визначати багатосимвольні оператори (наприклад, “===”).

11. Додаток А. Код лексичного аналізатора

```
import fs from "node:fs";

const keywords = [
  "fun",
  "var",
  "if",
  "else",
  "while",
  "Int",
  "Float",
  "true",
  "false",
  "package",
];

const operators = [
  "+",
  "-",
  "*",
  "/",
  "=",
  "==",
  "!=",
  "^",
  "<",
  "<=",
  ">",
  ">=",
  "===",
];

const punctuation = [";", ":", ",", "(", ")", "{", "}"];

enum CharClass {
```

```

    LETTER = "Letter",
    DIGIT = "Digit",
    DOT = "Dot",
    WHITESPACE = "Whitespace",
    EOL = "EOL",
    OPERATOR = "Operator",
    PUNCTUATION = "Punctuation",
    SLASH = "Slash",
    OTHER = "Other",
}

enum State {
    START = "S0",
    IDENTIFIER = "S1",
    INTEGER = "S2",
    FLOAT = "S3",
    OPERATOR = "S4",
    DIVIDE_OP = "S5",
    COMMENT = "S6",
    ERROR = "S_Err",
}

const transitionTable: Record<State, Record<string, State | undefined>>
= {
    [State.START]: {
        [CharClass.LETTER]: State.IDENTIFIER,
        [CharClass.DIGIT]: State.INTEGER,
        [CharClass.DOT]: State.FLOAT,
        [CharClass.WHITESPACE]: State.START,
        [CharClass.EOL]: State.START,
        [CharClass.OPERATOR]: State.OPERATOR,
        [CharClass.PUNCTUATION]: State.OPERATOR,
        [CharClass.SLASH]: State.DIVIDE_OP,
    },
    [State.IDENTIFIER]: {
        [CharClass.LETTER]: State.IDENTIFIER,
        [CharClass.DIGIT]: State.IDENTIFIER,
        default: State.START,
    },
    [State.INTEGER]: {
        [CharClass.DIGIT]: State.INTEGER,
        [CharClass.DOT]: State.FLOAT,
    },

```

```

        default: State.START,
    },
    [State.FLOAT]: {
        [CharClass.DIGIT]: State.FLOAT,
        default: State.START,
    },
    [State.OPERATOR]: {
        [CharClass.OPERATOR]: State.OPERATOR,
        default: State.START,
    },
    [State.DIVIDE_OP]: {
        [CharClass.SLASH]: State.COMMENT,
        default: State.START,
    },
    [State.COMMENT]: {
        default: State.COMMENT,
        [CharClass.EOL]: State.START,
    },
    [State.ERROR]: {},
};

interface Token {
    line: number;
    lexeme: string;
    token: string;
    index: number | null | undefined;
}

interface IdConstTable {
    identifiers: Record<string, number | undefined>;
    constants: Record<string, number | undefined>;
}

const getCharClass = (char: string) => {
    if (/[a-zA-Z]/.test(char)) return CharClass.LETTER;
    if (/\\d/.test(char)) return CharClass.DIGIT;
    if (char === ".") return CharClass.DOT;
    if (char === "/") return CharClass.SLASH;
    if (/\\s/.test(char)) {
        if (char === "\\n") return CharClass.EOL;
        return CharClass.WHITESPACE;
    }
}

```

```

        if (operators.some((op) => op.startsWith(char))) return
CharClass.OPERATOR;
        if (punctuation.includes(char)) return CharClass.PUNCTUATION;
        return CharClass.OTHER;
    };

    const getNextState = (currentState: State, charClass: CharClass): State
=> {
        const stateTransitions = transitionTable[currentState];
        if (stateTransitions[charClass]) {
            return stateTransitions[charClass];
        } else if (stateTransitions["default"]) {
            return stateTransitions["default"];
        } else {
            return State.ERROR;
        }
    };

    const isFinalState = (state: State) => {
        return (
            state === State.IDENTIFIER ||
            state === State.INTEGER ||
            state === State.FLOAT ||
            state === State.OPERATOR ||
            state === State.DIVIDE_OP
        );
    };

    const processLexeme = (
        state: State,
        lexeme: string,
        lineNumber: number,
        tokenTable: Token[],
        idConstTable: IdConstTable
    ) => {
        if (state === State.IDENTIFIER) {
            if (keywords.includes(lexeme)) {
                tokenTable.push({
                    line: lineNumber,
                    lexeme,
                    token: "keyword",
                    index: null,

```

```

    });
  } else {
    let index = idConstTable.identifiers.hasOwnProperty(lexeme)
      ? idConstTable.identifiers[lexeme]
      : Object.keys(idConstTable.identifiers).length + 1;
    if (!idConstTable.identifiers.hasOwnProperty(lexeme)) {
      idConstTable.identifiers[lexeme] = index;
    }
    tokenTable.push({ line: lineNumber, lexeme, token: "identifier",
index });
  }
} else if (state === State.INTEGER) {
  let index = idConstTable.constants.hasOwnProperty(lexeme)
    ? idConstTable.constants[lexeme]
    : Object.keys(idConstTable.constants).length + 1;
  if (!idConstTable.constants.hasOwnProperty(lexeme)) {
    idConstTable.constants[lexeme] = index;
  }
  tokenTable.push({ line: lineNumber, lexeme, token: "int_literal",
index });
} else if (state === State.FLOAT) {
  let index = idConstTable.constants.hasOwnProperty(lexeme)
    ? idConstTable.constants[lexeme]
    : Object.keys(idConstTable.constants).length + 1;
  if (!idConstTable.constants.hasOwnProperty(lexeme)) {
    idConstTable.constants[lexeme] = index;
  }
  tokenTable.push({
    line: lineNumber,
    lexeme,
    token: "float_literal",
    index,
  });
} else if (state === State.OPERATOR) {
  let tokenType = operators.includes(lexeme) ? "operator" :
"punctuation";

  if (tokenType === "punctuation") {
    if (lexeme === "(") {
      tokenType = "bracket_open";
    }
  }
}

```

```

        if (lexeme === ")") {
            tokenType = "bracket_close";
        }
    }

    tokenTable.push({
        line: lineNumber,
        lexeme,
        token: tokenType,
        index: null,
    });
} else if (state === State.DIVIDE_OP) {
    tokenTable.push({
        line: lineNumber,
        lexeme,
        token: "divideOp",
        index: null,
    });
}
};

const lexer = (code: string) => {
    let state = State.START;
    let lexeme = "";
    let tokens: Token[] = [];
    let idConstTable = { identifiers: {}, constants: {} };
    let lines = code.split("\n");

    lines.forEach((line, lineNumber) => {
        let i = 0;
        while (i < line.length) {
            let char = line[i];
            let charClass = getCharClass(char);

            let nextStateVal = getNextState(state, charClass);

            if (
                charClass === CharClass.SLASH &&
                getCharClass(line[i + 1]) === CharClass.SLASH
            ) {
                break;
            }
        }
    });
};

```

```

        if (nextStateVal === State.ERROR) {
            throw new Error(
                `Помилка: невідомий символ '${char}' у рядку ${lineNumber +
1}`
            );
        }

        if (nextStateVal !== state) {
            if (
                isFinalState(state) &&
                !(state === State.INTEGER && nextStateVal === State.FLOAT)
            ) {
                processLexeme(state, lexeme, lineNumber + 1, tokens,
idConstTable);
                lexeme = "";
            }
            state = nextStateVal;
        }

        if (charClass === CharClass.PUNCTUATION) {
            if (lexeme.length > 0) {
                processLexeme(state, lexeme, lineNumber + 1, tokens,
idConstTable);
                lexeme = "";
            }
            processLexeme(
                State.OPERATOR,
                char,
                lineNumber + 1,
                tokens,
                idConstTable
            );
            state = State.START;
            i++;
            continue;
        }

        if (state !== State.START) {
            lexeme += char;
        }

```

```

        i++;
    }

    if (isFinalState(state)) {
        processLexeme(state, lexeme, lineNumber + 1, tokens,
idConstTable);
        lexeme = "";
        state = State.START;
    } else {
        state = State.START;
        lexeme = "";
    }
});

return { tokens, idConstTable };
};

const printTokens = (tokens: Token[]) => {
    console.log("№\tРядок\tЛексема\t\tТокен\t\tІндекс");
    tokens.forEach((token, index) => {
        console.log(
            `${index}
1}\t${token.line}\t${token.lexeme}\t\t${token.token}\t\t${
            token.index !== null ? token.index : "-"
        }`
        );
    });
};

const printSymbolTables = (idConstTable: IdConstTable) => {
    console.log("\nТаблиця ідентифікаторів:");
    console.log("Індекс\t Ідентифікатор");
    for (const [id, idx] of Object.entries(idConstTable.identifiers)) {
        console.log(`${idx}\t${id}`);
    }

    console.log("\nТаблиця констант:");
    console.log("Індекс\tКонстанта");
    for (const [constVal, idx] of Object.entries(idConstTable.constants))
{
        console.log(`${idx}\t${constVal}`);
    }
}

```



```
};

const main = () => {
  try {
    const code = fs.readFileSync(process.argv[2], "utf-8");
    const { tokens, idConstTable } = lexer(code);
    printTokens(tokens);
    printSymbolTables(idConstTable);
    console.log("\nLexer: Лексичний аналіз завершено успішно");
  } catch (e) {
    if (e instanceof Error) console.error(`Lexer: ${e.message}`);
  }
};

main();
```