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»)

Звіт студента І курсу, групи ІП-51мн
спеціальності F2 Інженерія програмного забезпечення
Панченко С. В. _____
(Прізвище, ім’я, по батькові) (Підпис)

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

Київ – 2025

Зміст

1 МЕТА

Проєктування та програмна реалізація лексичного аналізатора мови програмування з використанням діаграми станів.

2 ВИКОНАННЯ

Лабораторна робота присвячена реалізації лексичного аналізатора для префіксної S-подібної мови, заданої в ANTLR4-специфікації. Метод діаграми станів (finite state machine, FSM) використано для побудови прозорої логіки переходів, класифікації лексем і діагностики помилок. Особливість мови — широке правило ID, яке дозволяє як буквено-цифрові, так і операторні символи (+ - * / ^ < > =), тому «оператори» на кшталт +, -, ==, **, le, concat лексично є ідентифікаторами (функціями) і не мають окремих токенів.

Лексер реалізовано на Python як окрему програму, що читає файл з кодом, виконує сканування символів і будує три таблиці: розбору (послідовність токенів із рядком і, за потреби, індексом), ідентифікаторів (унікальні ID з індексами) та констант (рядкові/числові/булеві літерали з типом). Помилки (заборонені символи, незакритий рядок, некоректна крапка в числі) супроводжуються повідомленням із локалізацією (рядок, позиція). Машина станів підтримує екранування у рядках і розпізнає числа з опційним знаком, якщо знак безпосередньо передує цифрі.

Для опису діаграми станів використовується PlantUML. Стани позначено S0, S_ID тощо; «Accept» — прийняття лексеми; unget — повернення останнього непридатного символа до потоку. Пробіли/переноси ігноруються; дужки (і) є окремими токенами.

Метасимвол	Значення
-->	перехід між станами
S0	початковий стан / пропуск пробілів
AcceptX	прийняття токена X
Error	помилка розбору
unget	повернення символа

Розробка лексера включала: визначення класів символів на основі алфавіту мови; проєктування FSM з мінімально достатніми станами для ID, чисел і рядків;

реалізацію семантичних процедур прийняття лексем (пріоритетне зіставлення з типами/ключовими словами/булевими значеннями, ведення таблиць); програмну реалізацію сканера з операціями get/unget, підрахунком рядка і позиції; тестування на еталонній програмі та edge-кейсах; підготовку таблиць і короткої діагностики.

Семантика: Лексер не виконує жодних синтаксичних перевірок (немає відповідності арностей тощо) і працює незалежно від парсера, забезпечуючи стабільну послідовність токенів для наступної фази.

Алфавіт поділено на:

- **WS**: пробіл, табуляція, переноси рядків — ігноруються.
- **LPAREN / RPAREN**: (та).
- **QUOTE**: " — початок/кінець рядка.
- **DIGIT**: 0–9.
- **SIGN**: + або - — частина числа лише якщо відразу за нею йде цифра.
- **ID_START**: [a-zA-Z_] або один із + - * / ^ < > =.
- **ID_BODY**: ID_START + цифри.
- **OTHER**: заборонені символи → помилка.

Семантика: Правило ID включає послідовності операторних символів; отже, (** 2 6) і (== 55 104) лексично коректні як виклики функцій ** та ==.

Діаграма моделює розбір з обробкою дужок, рядків з екрануванням, чисел із крапкою/знаком та ідентифікаторів/«операторів».

```
@startuml
skinparam state {
  BackgroundColor Snow
  BorderColor Black
  ArrowColor Black
}

[*] --> S0

state S0 : start / skip WS
S0 --> LPAREN_ACC : '('
S0 --> RPAREN_ACC : ')'
S0 --> S_STR : '"'
```

```
S0 --> S_NUM_SIGN : '+' or '-' (if next is DIGIT)
S0 --> S_NUM_INT : DIGIT
S0 --> S_ID : [A-Za-z_+\-*/^<>=]
S0 --> ERROR : other

state S_ID
S_ID --> S_ID : [A-Za-z0-9_+\-*/^<>=]
S_ID --> ID_ACC : other (unget)

state S_NUM_SIGN
S_NUM_SIGN --> S_NUM_INT : DIGIT

state S_NUM_INT
S_NUM_INT --> S_NUM_INT : DIGIT
S_NUM_INT --> S_NUM_DOT : '.'
S_NUM_INT --> INT_ACC : other (unget)

state S_NUM_DOT
S_NUM_DOT --> S_NUM_REAL : DIGIT
S_NUM_DOT --> ERROR : other

state S_NUM_REAL
S_NUM_REAL --> S_NUM_REAL : DIGIT
S_NUM_REAL --> REAL_ACC : other (unget)

state S_STR
S_STR --> S_STR_ESC : '\\'
S_STR --> STR_ACC : '"'
S_STR --> S_STR : any except [" \\ \r \n]
S_STR --> ERROR : EOL/EOF

state S_STR_ESC
S_STR_ESC --> S_STR : any

state LPAREN_ACC
LPAREN_ACC --> S0
state RPAREN_ACC
RPAREN_ACC --> S0
state ID_ACC
ID_ACC --> S0
state INT_ACC
INT_ACC --> S0
state REAL_ACC
REAL_ACC --> S0
state STR_ACC
STR_ACC --> S0

state ERROR : report and stop
@enduml
```

Семантика: Після прийняття лексеми виконується класифікація з пріоритетом: TYPE_* → KEYWORD_* → LITERAL_BOOL → ID.

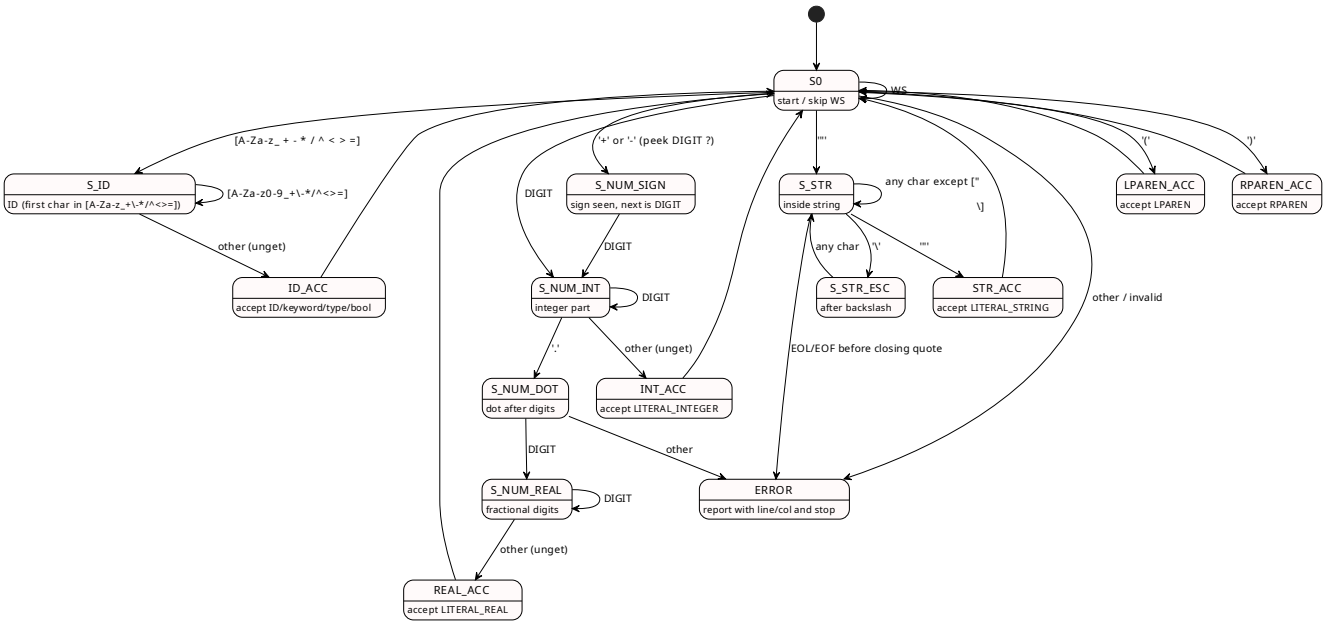

Рисунок 2.1 — Діграма станів

Семантичні процедури виконуються при прийнятті лексеми. keyword_or_type_or_bool(lexeme) застосовує пріоритетне зіставлення: рядок → типи int|real|string|bool (токени TYPE_*), далі ключові слова fn|let|if|while (токени KEYWORD_*), далі булеві true|false (як LITERAL_BOOL + запис у таблицю констант), в іншому разі — ID (запис у таблицю ідентифікаторів). add_to_const_table(lexeme, type) додає/повертає індекс для LITERAL_STRING, LITERAL_INTEGER, LITERAL_REAL, LITERAL_BOOL. add_to_id_table(lexeme) веде словник унікальних ідентифікаторів. add_to_table(token, line, index?) фіксує послідовність токенів у таблиці розбору. error(message, line, col) зупиняє лексер із діагностикою.

Семантика: Знак +/- формує число лише якщо перед цифрою; інакше — це початок ID (оператор-функція).

Нижче наведено реалізацію лексера (Python 3).

```
import sys
```

```
import dataclasses
from typing import Optional, List, Tuple

@dataclasses.dataclass
class TokenEntry:
    token: str
    line: int
    index: Optional[int]


token_table: List[TokenEntry] = []
id_table: dict[str, int] = {}
const_table: dict[Tuple[str, str], int] = {}
_id_counter = 0
_const_counter = 0

def add_token(token: str, line: int, index: Optional[int] = None) ->
str:
    token_table.append(TokenEntry(token, line, index))
    return token

def add_id(lexeme: str) -> Tuple[str, int]:
    global _id_counter
    if lexeme not in id_table:
        _id_counter += 1
        id_table[lexeme] = _id_counter
    return ("ID", id_table[lexeme])

def add_const(lexeme: str, ctype: str) -> Tuple[str, int]:
    global _const_counter
    key = (lexeme, ctype)
    if key not in const_table:
        _const_counter += 1
        const_table[key] = _const_counter
    return (ctype, const_table[key])

class Source:
    def __init__(self, text: str):
        self.text = text
        self.n = len(text)
        self.i = 0
        self.line = 1
        self.col = 0

    def eof(self) -> bool:
        return self.i >= self.n

    def peek(self) -> str:
        if self.i >= self.n:
```

```
            return ''
        return self.text[self.i]

    def get(self) -> str:
        if self.i >= self.n:
            return ''
        ch = self.text[self.i]
        self.i += 1
        if ch == '\n':
            self.line += 1
            self.col = 0
        else:
            self.col += 1
        return ch

    def unget(self):
        if self.i == 0:
            return
        self.i -= 1
        ch = self.text[self.i]
        if ch == '\n':

            self.line -= 1

            j = self.i - 1
            c = 0
            while j >= 0 and self.text[j] != '\n':
                c += 1
                j -= 1
            self.col = c
        else:
            self.col -= 1
            if self.col < 0:
                self.col = 0

def is_ws(c: str) -> bool:
    return c in ' \t\r\n'

def is_digit(c: str) -> bool:
    return '0' <= c <= '9'

def is_id_start(c: str) -> bool:

    return (c.isalpha() or c == '_' or c in '+-*/^<=>')

def is_id_body(c: str) -> bool:

    return (c.isalpha() or c.isdigit() or c == '_' or c in
'+-*/^<=>')
```

```
TYPES = {
    'int': 'TYPE_INTEGER',
    'real': 'TYPE_REAL',
    'string': 'TYPE_STRING',
    'bool': 'TYPE_BOOL',
}
KEYWORDS = {
    'fn': 'KEYWORD_FN',
    'let': 'KEYWORD_LET',
    'if': 'KEYWORD_IF',
    'while': 'KEYWORD_WHILE',
}
BOOLS = {'true', 'false'}

def classify_id(lexeme: str) -> Tuple[str, Optional[int]]:

    if lexeme in TYPES:
        return (TYPES[lexeme], None)
    if lexeme in KEYWORDS:
        return (KEYWORDS[lexeme], None)
    if lexeme in BOOLS:
        tok, idx = add_const(lexeme, 'LITERAL_BOOL')
        return ('LITERAL_BOOL', idx)
    tok, idx = add_id(lexeme)
    return (tok, idx)




def error(line: int, col: int, msg: str):
    print(f"Помилка на рядку {line}, позиція {col}: {msg}")
    sys.exit(1)

def lex(source: Source):
    while True:

        while not source.eof() and is_ws(source.peek()):
            source.get()

        if source.eof():
            break

        ch = source.get()


        if ch == '(':
```

```
            add_token('LPAREN', source.line)
            continue
        if ch == ')':
            add_token('RPAREN', source.line)
            continue


        if ch == '"':
            buf = []
            while True:
                if source.eof():
                    error(source.line, source.col, "Незакритий
рядок")
                c = source.get()
                if c == '"':
                    lexeme = ''.join(buf)
                    tok, idx = add_const(lexeme, 'LITERAL_STRING')
                    add_token('LITERAL_STRING', source.line, idx)
                    break
                if c == '\n' or c == '\r':
                    error(source.line, source.col, "Рядок перенесено
до закриття '\"'")
                if c == '\\':

                    if source.eof():
                        error(source.line, source.col, "Незавершена
escape-послідовність")
                    e = source.get()

                    if e == 'n':
                        buf.append('\n')
                    elif e == 'r':
                        buf.append('\r')
                    elif e == 't':
                        buf.append('\t')
                    elif e == '"':
                        buf.append('"')
                    elif e == '\\':
                        buf.append('\\')
                    else:

                        buf.append(e)
                else:
                    buf.append(c)
            continue


        if ch in '+-':
            nxt = source.peek()
```

```
            if is_digit(nxt):

                buf = [ch]

                while is_digit(source.peek()):
                    buf.append(source.get())
                if source.peek() == '.':

                    buf.append(source.get())
                    if not is_digit(source.peek()):
                        error(source.line, source.col, "Після крапки
очікується цифра")
                    while is_digit(source.peek()):
                        buf.append(source.get())
                    lexeme = ''.join(buf)
                    tok, idx = add_const(lexeme, 'LITERAL_REAL')
                    add_token('LITERAL_REAL', source.line, idx)
                else:
                    lexeme = ''.join(buf)
                    tok, idx = add_const(lexeme, 'LITERAL_INTEGER')
                    add_token('LITERAL_INTEGER', source.line, idx)
                continue
            else:


                pass


        if ch.isdigit():
            buf = [ch]
            while is_digit(source.peek()):
                buf.append(source.get())
            if source.peek() == '.':
                buf.append(source.get())
                if not is_digit(source.peek()):
                    error(source.line, source.col, "Після крапки
очікується цифра")
                while is_digit(source.peek()):
                    buf.append(source.get())
                lexeme = ''.join(buf)
                tok, idx = add_const(lexeme, 'LITERAL_REAL')
                add_token('LITERAL_REAL', source.line, idx)
            else:
                lexeme = ''.join(buf)
                tok, idx = add_const(lexeme, 'LITERAL_INTEGER')
                add_token('LITERAL_INTEGER', source.line, idx)
            continue
```

```
        if is_id_start(ch):
            buf = [ch]
            while is_id_body(source.peek()):
                buf.append(source.get())
            lexeme = ''.join(buf)
            tok, idx = classify_id(lexeme)
            add_token(tok, source.line, idx)
            continue


        error(source.line, source.col, f"Неприпустимий символ
'{ch}'")

def lexical_analysis(filename: str):
    global token_table, id_table, const_table, _id_counter,
_const_counter
    token_table = []
    id_table = {}
    const_table = {}
    _id_counter = 0
    _const_counter = 0

    with open(filename, 'r', encoding='utf-8') as f:
        text = f.read()

    src = Source(text)
    lex(src)

    print("Таблиця розбору:")
    for e in token_table:
        print((e.token, e.line, e.index))

    print("\nТаблиця ідентифікаторів:")
    for k, v in sorted(id_table.items(), key=lambda kv: kv[1]):
        print(f"{v}: {k}")

    print("\nТаблиця констант:")
    for (lexeme, ctype), idx in sorted(const_table.items(),
key=lambda kv: kv[1]):
        print(f"{idx}: {lexeme} ({ctype})")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Вкажіть шлях до файлу: python lexer.py program.txt")
        sys.exit(0)
    lexical_analysis(sys.argv[1])
```

Тестування виконано на програмі, наведеній нижче.

```
(fn otherfunc () (
    (display (** 2 6))
))

(fn main ((a int) (b real) (c string)) (
    (let d (concat c "Hello KPI"))
    (let dd true)
    (if (== 55 104) (
        (let d 33)
        (display (concat d "I AM TRUE"))
    ) (
        (display (concat d "I AM FALSE"))
    ))
    (let i 0)
    (while (le i 5) (
        (display (* 2 (- a (/ i b))))
        (set i (+ i 1))
    ))
))
```

Таблиця розбору.

```
Таблиця розбору:
('LPAREN', 1, None)
('KEYWORD_FN', 1, None)
('ID', 1, 1)
('LPAREN', 1, None)
('RPAREN', 1, None)
('LPAREN', 1, None)
('LPAREN', 2, None)
('ID', 2, 2)
('LPAREN', 2, None)
('ID', 2, 3)
('LITERAL_INTEGER', 2, 1)
('LITERAL_INTEGER', 2, 2)
('RPAREN', 2, None)
('RPAREN', 2, None)
('RPAREN', 3, None)
('RPAREN', 3, None)
('LPAREN', 5, None)
('KEYWORD_FN', 5, None)
('ID', 5, 4)
('LPAREN', 5, None)
('LPAREN', 5, None)
('ID', 5, 5)
('TYPE_INTEGER', 5, None)
('RPAREN', 5, None)
```

```
('LPAREN', 5, None)
('ID', 5, 6)
('TYPE_REAL', 5, None)
('RPAREN', 5, None)
('LPAREN', 5, None)
('ID', 5, 7)
('TYPE_STRING', 5, None)
('RPAREN', 5, None)
('RPAREN', 5, None)
('LPAREN', 5, None)
('LPAREN', 6, None)
('KEYWORD_LET', 6, None)
('ID', 6, 8)
('LPAREN', 6, None)
('ID', 6, 9)
('ID', 6, 7)
('LITERAL_STRING', 6, 3)
('RPAREN', 6, None)
('RPAREN', 6, None)
('LPAREN', 7, None)
('KEYWORD_LET', 7, None)
('ID', 7, 10)
('LITERAL_BOOL', 7, 4)
('RPAREN', 7, None)
('LPAREN', 8, None)
('KEYWORD_IF', 8, None)
('LPAREN', 8, None)
('ID', 8, 11)
('LITERAL_INTEGER', 8, 5)
('LITERAL_INTEGER', 8, 6)
('RPAREN', 8, None)
('LPAREN', 8, None)
('LPAREN', 9, None)
('KEYWORD_LET', 9, None)
('ID', 9, 8)
('LITERAL_INTEGER', 9, 7)
('RPAREN', 9, None)
('LPAREN', 10, None)
('ID', 10, 2)
('LPAREN', 10, None)
('ID', 10, 9)
('ID', 10, 8)
('LITERAL_STRING', 10, 8)
('RPAREN', 10, None)
('RPAREN', 10, None)
('RPAREN', 11, None)
('LPAREN', 11, None)
('LPAREN', 12, None)
('ID', 12, 2)
```

```
('LPAREN', 12, None)
('ID', 12, 9)
('ID', 12, 8)
('LITERAL_STRING', 12, 9)
('RPAREN', 12, None)
('RPAREN', 12, None)
('RPAREN', 13, None)
('RPAREN', 13, None)
('LPAREN', 14, None)
('KEYWORD_LET', 14, None)
('ID', 14, 12)
('LITERAL_INTEGER', 14, 10)
('RPAREN', 14, None)
('LPAREN', 15, None)
('KEYWORD_WHILE', 15, None)
('LPAREN', 15, None)
('ID', 15, 13)
('ID', 15, 12)
('LITERAL_INTEGER', 15, 11)
('RPAREN', 15, None)
('LPAREN', 15, None)
('LPAREN', 16, None)
('ID', 16, 2)
('LPAREN', 16, None)
('ID', 16, 14)
('LITERAL_INTEGER', 16, 1)
('LPAREN', 16, None)
('ID', 16, 15)
('ID', 16, 5)
('LPAREN', 16, None)
('ID', 16, 16)
('ID', 16, 12)
('ID', 16, 6)
('RPAREN', 16, None)
('RPAREN', 16, None)
('RPAREN', 16, None)
('RPAREN', 16, None)
('LPAREN', 17, None)
('ID', 17, 17)
('ID', 17, 12)
('LPAREN', 17, None)
('ID', 17, 18)
('ID', 17, 12)
('LITERAL_INTEGER', 17, 12)
('RPAREN', 17, None)
('RPAREN', 17, None)
('RPAREN', 18, None)
('RPAREN', 18, None)
('RPAREN', 19, None)
```

```
('RPAREN', 19, None)

Таблиця ідентифікаторів:
1: otherfunc
2: display
3: **
4: main
5: a
6: b
7: c
8: d
9: concat
10: dd
11: ==
12: i
13: le
14: *
15: -
16: /
17: set
18: +
\нТаблиця констант:
1: 2 (LITERAL_INTEGER)
2: 6 (LITERAL_INTEGER)
3: Hello KPI (LITERAL_STRING)
4: true (LITERAL_BOOL)
5: 55 (LITERAL_INTEGER)
6: 104 (LITERAL_INTEGER)
7: 33 (LITERAL_INTEGER)
8: I AM TRUE (LITERAL_STRING)
9: I AM FALSE (LITERAL_STRING)
10: 0 (LITERAL_INTEGER)
11: 5 (LITERAL_INTEGER)
12: 1 (LITERAL_INTEGER)
```

3 ВИСНОВОК

Реалізовано лексичний аналізатор для S-подібної мови у префіксній формі з використанням FSM. Завдяки широкому правилу ID операторні послідовності не потребують окремого лексичного шару, що спрощує лексер і переносить «смисл операторів» на рівень синтаксису. Машина станів успішно покриває дужки, рядки з екрануванням, числа з опційним знаком та класифікацію зарезервованих слів. Подальші кроки можуть включати підтримку експоненційної нотації чисел і введення коментарів за потреби специфікації.