НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота № 3

СИНТАКСИЧНИЙ ТА СЕМАНТИЧНИЙ АНАЛІЗ

(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

Програмна реалізація синтаксичного аналізатора (парсера) заданої мови методом рекурсивного спуску.

2 ВИКОНАННЯ

2.1 Короткий опис

Розроблена мова є простою Lisp-подібною процедурною мовою, орієнтованою на дослідницьке моделювання та генерацію коду. Її синтаксис побудований повністю на дужкових виразах, що забезпечує однорідну деревоподібну структуру програми й спрощує як лексичний, так і синтаксичний аналіз. Програма складається з набору визначень функцій у вигляді списків виду `(fn <ім’я> <тип-повернення> <аргументи> <змінні> <оператори>)`.

Мова підтримує базові типи даних `int`, `float`, `bool`, а також булеві константи `true` і `false`. Набір операторів включає присвоєння `set`, умовний оператор `if`, цикл `while` і `return`. Вирази формуються з атомів (ідентифікаторів, чисел, булевих значень) або викликів функцій, включно з вбудованими арифметичними та порівняльними операціями (`+`, `-`, `*`, `/`, `==`, `<`, `>`, `<=`, `>=`). Уся програма є правильно вкладеним деревом списків, що робить її придатною до рекурсивного синтаксичного аналізу без лівої рекурсії.

2.2 Повна граматика

Повна граматика мови, вільна від ліворекурсивних правил та представлена у форматі ANTLR4, описує кореневий елемент `program` як послідовність оголошень функцій. **Оголошення функції** (`function_definition`) є ключовою структурою і являє собою список з фіксованою структурою: `(fn <назва> <тип_повернення> (<аргументи>) (<змінні>) (<оператори>...))`. Це правило гарантує, що синтаксичний аналізатор очікує рівно шість елементів у списку, починаючи з ключового слова `'fn'`.

```
grammar MiniLispLang;

// --- Початкове правило ---
```

```
program
    : '(' functionDef+ ')' EOF
    ;

// --- Визначення функції ---
functionDef
    : '(' 'fn' IDENT TYPE argList varList stmtList ')'
    ;

// --- Списки аргументів і змінних ---
argList
    : '(' (nameTypePair)* ')'
    ;

varList
    : '(' (nameTypePair)* ')'
    ;

nameTypePair
    : '(' IDENT TYPE ')'
    ;

// --- Типи ---
TYPE
    : 'int' | 'float' | 'bool'
    ;

// --- Список операторів ---
stmtList
    : '(' statement+ ')'
    ;

// --- Оператори ---
statement
    : '(' 'set' IDENT expr ')'
    | '(' 'if' expr stmtList stmtList ')'
    | '(' 'while' expr stmtList ')'
    | '(' 'return' expr ')'
    ;

// --- Вирази ---
expr
    : atom
    | funcCall
    ;

funcCall
    : '(' IDENT expr* ')'
    ;
```

```
// --- Атоми ---
atom
    : IDENT
    | INT
    | FLOAT
    | BOOL
    ;

// --- Лексеми ---
IDENT
    : [a-zA-Z!$%&*/+\-:<=>?^_~\.0-9]+
    ;

INT
    : [+\-]?[0-9]+
    ;

FLOAT
    : [+\-]?[0-9]+'.'[0-9]+
    ;

BOOL
    : 'true' | 'false'
    ;

LPAREN : '(' ;
RPAREN : ')' ;
WS : [ \t\r\n]+ -> skip ;
```

Лексичний аналізатор реалізований у вигляді послідовного проходу файлу, який ідентифікує три основні типи токенів: ліві дужки `(`, праві дужки `)`, ідентифікатори або константи (`TokenIdentifier`). Використовуються регулярні вирази для пошуку збігів у тексті:

- `[(]` та `[)]` для дужок;
- `[a-zA-Z!$%&*/+\-:<=>?^_~\.0-9]+` для атомів;
- `[ \t\r]+` для пропусків;
- `[\n]` для нових рядків.

Під час токенізації зберігаються координати (номер рядка і стовпця) для діагностики помилок. Усі нерозпізнані символи викликають аварійне завершення з повідомленням про позицію помилки. Таким чином, лексер побудований без

залежностей і може бути використаний як окремий модуль для будь-якої Lisp-подібної мови.

2.3 Синтаксичний аналізатор

Синтаксичний аналізатор побудований на рекурсивному розборі вкладених списків (`LispList`). Після токенізації створюється дерево, де кожен вузол відповідає відкритій дужці, а його нащадки — елементи списку. Рекурсія завершується при зустрічі правої дужки.

Далі відбувається етап синтаксичного розбору, який перетворює це дерево у внутрішнє представлення (`SyntaxFunctionDefinition`, `SyntaxStatementSet`, `SyntaxStatementIf`, тощо). На цьому етапі відбувається також перевірка типів, коректності ідентифікаторів і структури функцій (правильна кількість аргументів, повернення значення відповідного типу, відсутність дублікатів імен).

Для операторів `if`, `while`, `set`, `return` викликаються окремі функції-парсери, які формують об’єкти класів-структур. Для виразів реалізовано перевірку правильності типів і відповідності функцій — користувацьких або вбудованих. Система також перевіряє збалансованість дужок та обов’язкову наявність оператору `return` у кінці кожної функції.

Завдяки повністю рекурсивній побудові й відсутності лівої рекурсії синтаксичний аналізатор є простим, надійним і легко розширюваним: нові конструкції можна додавати через нові гілки в `syntax_parse_*` функціях або відповідні правила у граматиці ANTLR4.

2.4 Семантичний аналізатор

емантичний аналізатор у розробленій мові є другим етапом після синтаксичного розбору. Його завдання полягає у перевірці правильності змісту програми — не лише того, що структура коду відповідає граматиці, а й того, що всі

операції, змінні та функції мають логічний сенс. Семантичний аналіз базується на побудованому синтаксичному дереві, де кожен вузол відповідає певній конструкції мови. Аналізатор проходить це дерево рекурсивно, перевіряючи оголошення функцій, допустимість операцій і коректність типів.

На початку аналізу кожна функція перевіряється на унікальність імені. Це забезпечує відсутність дублювань і конфліктів у глобальному просторі. Далі аналізатор перевіряє унікальність імен аргументів і локальних змінних усередині кожної функції. Якщо імена повторюються, виводиться повідомлення про помилку з точною вказівкою рядка і стовпця, де виявлено дубль. Така перевірка гарантує, що кожен ідентифікатор у межах однієї функції має однозначне значення.

Важливою частиною є перевірка типів. Семантичний аналізатор стежить за тим, щоб усі вирази були типово узгодженими. Для операторів присвоєння перевіряється, що змінна ліворуч і вираз праворуч мають однаковий тип. Для умов у конструкціях `if` і `while` вимагається тип `bool`. Для операторів `return` аналізатор звіряє тип поверненого виразу з типом, який оголошено у сигнатурі функції. Будь-яка невідповідність призводить до зупинки аналізу з помилкою.

Під час проходу тіла функції аналізатор також перевіряє, чи всі використані змінні й аргументи справді оголошені у списку параметрів або локальних змінних. Якщо зустрічається невідомий ідентифікатор, програма вважається некоректною. Аналогічно перевіряються всі виклики функцій — аналізатор порівнює кількість і типи аргументів із визначенням цільової функції або з відомими вбудованими функціями. Якщо сигнатури не збігаються, повідомляється про помилку семантичного типу.

Кожна функція повинна мати хоча б одну інструкцію `return`, і ця інструкція має бути останньою в списку операторів. Якщо `return` відсутній або розташований не в кінці, аналізатор видає діагностичне повідомлення. Завдяки цьому забезпечується логічна завершеність кожної функції — вона обов’язково повертає значення свого типу.

Семантика самої мови побудована на строгій статичній типізації. Це означає, що всі типи — `int`, `float` і `bool` — визначаються під час аналізу, а не під час виконання. Кожен вираз має визначений тип, який або явно заданий (як у констант), або успадковується від контексту. Будь-яке порушення узгодженості типів виявляється ще до генерації коду. Такий підхід забезпечує надійність і запобігає типово небезпечним ситуаціям, властивим динамічним мовам.

В основі семантики лежить принцип однозначного відповідника між синтаксичними конструкціями та семантичними правилами. Наприклад, оператор `(set x expr)` означає присвоєння значення виразу змінній, яка вже існує і має відповідний тип. Конструкція `(if cond then else)` описує умовне виконання блоків операторів, де `cond` — логічний вираз, а обидві гілки — списки дій. Цикл `(while cond body)` повторює виконання, доки умова істинна, а `(return expr)` завершує функцію, повертаючи обчислене значення.

Семантична модель мови також визначає поведінку вбудованих функцій, які фактично є частиною стандартної бібліотеки. Серед них арифметичні операції `+`, `-`, `*`, `/`, порівняльні `==`, `<`, `>`, `<=`, `>=`, а також логічні операції для типу `bool`. Кожна така функція має фіксовану сигнатуру, відому аналізатору, що дозволяє автоматично визначати тип результату під час перевірки виразів.

2.5 Тестування

Тестування синтаксичного та семантичного аналізаторів проводилось у кілька етапів, відповідно до плану перевірки коректності структури, типів, вкладеності та обробки помилок. Усі приклади наведені у форматі вихідного коду мови та очікуваних результатів, які генерує інтерпретатор у випадку помилки або успішного аналізу.

2.5.1 Пропуск термінала

Мета: перевірити реакцію на відсутню закриваючу дужку. **Вхідний код:**

```
(fn main int () () (
    (set x 5)
    (return x)
```

Очікуваний результат:

```
Error: Unmatched paren at line 1, column 1
```

Аналізатор виявляє, що кількість відкритих і закритих дужок не збігається, і повідомляє про позицію останньої відкритої дужки, яка залишилася непарною.

2.5.2 Зайвий термінал

Мета: перевірити поведінку при наявності зайвої дужки. **Вхідний код:**

```
(fn test int () () (
    (return 0)
))
)
```

Очікуваний результат:

```
Error: Unrecognized symbol at line 5, position 1
```

Токенайзер зустрічає дужку, яка не належить жодній конструкції, тому повідомляє про невпізнаний символ.

2.5.3 Помилка у наборі ключового слова

Мета: перевірити правильність розпізнавання ключових слів. **Вхідний код:**

```
(fn man int () () (
    (retun 0)
))
```

Очікуваний результат:

```
Error: Statement name is not valid at line 3, column 6
```

У цьому прикладі замість `return` написано `retun`, що не збігається з жодним допустимим оператором. Аналізатор розпізнає перший елемент списку як невідомий і повідомляє про помилку.

2.5.4 Помилковий оператор у виразах

Мета: перевірити реакцію на використання невідомої функції.
Вхідний код:

```
(fn calc int ((x int)(y int)) () (
    (set z (??? x y))
    (return z)
))
```

Очікуваний результат:

```
Error: Function call does not match any functions at line 3, column 11
```

Під час семантичного аналізу виявлено виклик функції з ім’ям `???`, якої немає ані серед користувацьких, ані серед вбудованих.

2.5.5 Невідповідність типів

Мета: перевірити, чи аналізатор помічає типову помилку в операторі `set`.
Вхідний код:

```
(fn wrong int ((x int)) ((y bool)) (
    (set y (+ x 1))
    (return x)
))
```

Очікуваний результат:

```
Error: type mismatch between dest and src values in set statement at line 3, column 6
```

Оскільки змінна `y` має тип `bool`, а результат `( + x 1 )` має тип `int`, аналізатор повідомляє про несумісність типів.

2.5.6 Помилка в умові if

Мета: перевірити правильність типу виразу в умовній конструкції.
Вхідний код:

```
(fn condtest int ((a int)) () (
    (if a
        ((return 1))
        ((return 0)))
))
```

Очікуваний результат:

```
Error: Condition must be bool at line 3, column 5
```

Аналізатор очікує, що вираз у `if` повертає тип `bool`, але отримує `int`.

2.5.7 Вкладені конструкції

Мета: перевірити правильну роботу вкладених операторів.
Вхідний код:

```
(fn nested int ((x int)) () (
    (if (> x 0)
```

```
        ((while (> x 0)
            ((set x (- x 1)))))
        ((return 0)))
    (return x)
))
```

Очікуваний результат:

```
Success: Program parsed and validated correctly
```

Усі вкладені конструкції оброблені коректно — типи умов `bool`, вирази всередині мають правильні типи, програма завершується без помилок.

2.5.8 Відсутність return

Мета: перевірити наявність обов’язкового завершення функції.
Вхідний код:

```
(fn no_return int () () (
    (set x 10)
))
```

Очікуваний результат:

```
Error: Last statement in function definition statement list must be return statement at line 1, column 1
```

Функція не завершується оператором `return`, тому аналізатор повідомляє про порушення правил структури функції.

2.5.9 Дубльоване ім’я змінної

Мета: перевірити реакцію на однакові імена аргументів і змінних.
Вхідний код:

```
(fn dup int ((x int)) ((x int)) (
    (return x)
```

```
))
```

Очікуваний результат:

```
Error: Duplicate argument name at line 1, column 11 and at line 1, column 22
```

Аналізатор виявляє, що ім’я `x` використано двічі в межах однієї функції, що заборонено семантикою мови.

2.5.10 Висновок

У результаті тестування було підтверджено, що синтаксичний і семантичний аналізатори правильно обробляють як коректні програми, так і різноманітні класи помилок. Повідомлення про помилки мають чіткий формат `Error: ...`, що дозволяє легко локалізувати місце й причину збою. Тести з правильним кодом завершуються повідомленням про успіх, що свідчить про повну відповідність семантиці та граматиці мови.

2.6 Програмна реалізація

```
import os
import sys
import re
import dataclasses
import enum
import typing
import pathlib
import typing

class TokenLparen(typing.NamedTuple):
    line_number: int
    column_number: int

class TokenRparen(typing.NamedTuple):
    line_number: int
    column_number: int

class TokenIdentifier(typing.NamedTuple):
    line_number: int
    column_number: int
```

```
    value: str

Token: typing.TypeAlias = TokenLparen | TokenRparen |
TokenIdentifier


def panic(msg: str = '') -> typing.NoReturn:
    import traceback
    traceback.print_stack()
    print(msg)
    sys.exit(-1)

def tokenize(filepath: pathlib.Path | str) ->
typing.Iterator[Token]:
    lparen_re = re.compile(r'[(]')
    rparen_re = re.compile(r'[)]')
    identifier_re = re.compile(r'[a-zA-Z!$%&*/+\-:<=>?^_~\.0-9]+')
    empty_re = re.compile(r'[ \t\r]+')
    newline_re = re.compile(r'[\n]')

    with open(filepath, 'r') as file:
        data = file.read()

    offset = 0
    line_number = 1
    last_newline_offset = 0

    while offset < len(data):
        column_number = offset - last_newline_offset + 1
        subdata = data[offset:]
        if (resmatch := lparen_re.match(subdata)):
            match_len = resmatch.end()
            offset += match_len
            yield TokenLparen(line_number, column_number)
        elif (resmatch := rparen_re.match(subdata)):
            match_len = resmatch.end()
            offset += match_len
            yield TokenRparen(line_number, column_number)
        elif (resmatch := identifier_re.match(subdata)):
            match_len = resmatch.end()
            offset += match_len
            yield TokenIdentifier(line_number, column_number,
resmatch.group(0))
        elif (resmatch := empty_re.match(subdata)):
            match_len = resmatch.end()
            offset += match_len
        elif (resmatch := newline_re.match(subdata)):
            line_number += 1
            last_newline_offset = offset + 1
```

```
            offset += 1
        else:
            panic(f"Error: Unrecognized symbol at line
{line_number}, position {column_number}")

@dataclasses.dataclass(frozen=True, slots=True)
class LispList:
    lparen: TokenLparen
    elements: list[typing.Union['LispList', 'TokenIdentifier']] =
dataclasses.field(default_factory=list)

def build_lisp_tree(lisp_list: LispList, tokenizer:
typing.Iterator[Token], lparens: list[TokenLparen]):
    for token in tokenizer:
        if isinstance(token, TokenLparen):
            lparens.append(token)
            sub_lisp_list = LispList(token)
            build_lisp_tree(sub_lisp_list, tokenizer, lparens)
            lisp_list.elements.append(sub_lisp_list)
        elif isinstance(token, TokenRparen):
            lparens.pop()
            return
        else:
            lisp_list.elements.append(token)

def at_line(token: Token | LispList) -> str:
    if isinstance(token, Token):
        return f'at line {token.line_number}, column
{token.column_number}'
    else:
        return f'at line {token.lparen.line_number}, column
{token.lparen.column_number}'

def check_lisp_element_is_type(el: list[TokenIdentifier] |
TokenIdentifier):
    assert isinstance(el, TokenIdentifier)

import enum

class VarType(enum.StrEnum):
    INT = 'int'
    FLOAT = 'float'
    BOOL = 'bool'

IDENTIFIER_RE = re.compile("[a-zA-Z!$%&*/:<=>?^_~][a-zA-Z!$%&*/:<=>?
^_~0-9]*|[+]|[-]")
INT_RE = re.compile("[+-]?[0-9]+")
FLOAT_RE = re.compile("[+-]?[0-9]+[.][0-9]+")
```

```
class VarTypePair(typing.NamedTuple):
    token: TokenIdentifier
    vartype: VarType

@dataclasses.dataclass(slots=True, frozen=True)
class SyntaxList[T]:
    lisp_list: LispList
    syntax_list: list[T] = dataclasses.field(default_factory=list)

def syntax_parse_arg_list(arg_list: LispList | TokenIdentifier) ->
SyntaxList[VarTypePair]:
    if isinstance(arg_list, LispList):
        syntax_list = SyntaxList[VarTypePair](arg_list)
        for name_type_pair in arg_list.elements:
            if isinstance(name_type_pair, LispList):
                if len(name_type_pair.elements) != 2:
                    panic(f'Error: Name type pair must have 2
elements {at_line(name_type_pair.lparen)}')
                arg_name = name_type_pair.elements[0]
                if isinstance(arg_name, TokenIdentifier):
                    if IDENTIFIER_RE.fullmatch(arg_name.value) is
None:
                        panic(f'Error: Argument name does not match
identifier pattern {at_line(arg_name)}')
                    arg_type = name_type_pair.elements[1]
                    if isinstance(arg_type, TokenIdentifier):
                        if arg_type.value not in VarType:
                            panic(f'Error: Argument type is not
valid {at_line(arg_type)}')

syntax_list.syntax_list.append(VarTypePair(arg_name,
VarType(arg_type.value)))
                    else:
                        panic(f'Error: Argument type must be an atom
{at_line(arg_type.lparen)}')
                else:
                    panic(f'Error: Argument name must be an atom
{at_line(arg_name.lparen)}')
            else:
                panic(f'Error: Name type pair must be a list
{at_line(name_type_pair)}')
        return syntax_list
    else:
        panic(f'Error: Argument list must be a list
{at_line(arg_list)}')

def check_atom_identifier(lisp_element: LispList | TokenIdentifier)
-> TokenIdentifier:
    if isinstance(lisp_element, TokenIdentifier):
```

```
        if IDENTIFIER_RE.fullmatch(lisp_element.value):
            return lisp_element
        else:
            panic(f'Error: Element is not valid identifier
{at_line(lisp_element)}')
    else:
        panic(f'Error: Element must be an atom
{at_line(lisp_element.lparen)}')

class SyntaxBool(enum.StrEnum):
    TRUE = 'true'
    FALSE = 'false'

@dataclasses.dataclass(slots=True)
class SyntaxConstantBool:
    value: TokenIdentifier

@dataclasses.dataclass(slots=True)
class SyntaxConstantFloat:
    value: TokenIdentifier

@dataclasses.dataclass(slots=True)
class SyntaxConstantInt:
    value: TokenIdentifier

@dataclasses.dataclass(slots=True)
class SyntaxVariable:
    value: TokenIdentifier

SyntaxVariableOrConstant = SyntaxVariable | SyntaxConstantInt |
SyntaxConstantFloat | SyntaxConstantBool

def syntax_parse_var_or_const(lisp_element: LispList |
TokenIdentifier) -> SyntaxVariableOrConstant:
    if isinstance(lisp_element, TokenIdentifier):
        if lisp_element.value in SyntaxBool:
            return SyntaxConstantBool(lisp_element)
        elif INT_RE.fullmatch(lisp_element.value):
            return SyntaxConstantInt(lisp_element)
        elif FLOAT_RE.fullmatch(lisp_element.value):
            return SyntaxConstantFloat(lisp_element)
        else:
            return SyntaxVariable(lisp_element)
    else:
        panic(f'Error: Element must be an atom
{at_line(lisp_element.lparen)}')

@dataclasses.dataclass(slots=False)
class SyntaxFunctionCall:
```

```
    lisp_list: LispList
    name: TokenIdentifier
    arguments: tuple[SyntaxVariableOrConstant, ...]

SyntaxVarOrConstOrFuncCall = SyntaxVariableOrConstant |
SyntaxFunctionCall

def syntax_parse_var_or_const_or_func_call(statement_argument:
LispList | TokenIdentifier) -> SyntaxVarOrConstOrFuncCall:
    if isinstance(statement_argument, TokenIdentifier):
        return syntax_parse_var_or_const(statement_argument)
    else:
        return SyntaxFunctionCall(
            statement_argument,
            check_atom_identifier(statement_argument.elements[0]),
            tuple(syntax_parse_var_or_const(el) for el in
statement_argument.elements[1:])
        )

@dataclasses.dataclass(slots=True)
class SyntaxStatementSet:
    lisp_list: LispList
    dest: TokenIdentifier
    src: SyntaxVarOrConstOrFuncCall

@dataclasses.dataclass(slots=True)
class SyntaxStatementReturn:
    lisp_list: LispList
    value: SyntaxVarOrConstOrFuncCall

@dataclasses.dataclass(slots=True)
class SyntaxStatementIf:
    lisp_list: LispList
    condition: SyntaxVarOrConstOrFuncCall
    true_branch: SyntaxList['SyntaxStatement']
    false_branch: SyntaxList['SyntaxStatement']

@dataclasses.dataclass(slots=True)
class SyntaxStatementWhile:
    lisp_list: LispList
    condition: SyntaxVarOrConstOrFuncCall
    statements: SyntaxList['SyntaxStatement']

SyntaxStatement = SyntaxStatementSet | SyntaxStatementIf |
SyntaxStatementWhile | SyntaxStatementReturn

def syntax_parse_statement_list(lisp_statement_list: LispList |
TokenIdentifier) -> SyntaxList[SyntaxStatement]:
    if isinstance(lisp_statement_list, LispList):
```

```
        statements = SyntaxList[SyntaxStatement]
(lisp_statement_list)
        for lisp_statement in lisp_statement_list.elements:
            if isinstance(lisp_statement, LispList):
                if len(lisp_statement.elements) == 0:
                    panic(f'Error: Statement must be a non-empty
list {at_line(lisp_statement.lparen)}')
                statement_name = lisp_statement.elements[0]
                if isinstance(statement_name, TokenIdentifier):
                    if statement_name.value == 'set':
                        if len(lisp_statement.elements) != 3:
                            panic(f'Error: Set statement list must
have 3 elements {at_line(lisp_statement.lparen)}')
                        statements.syntax_list.append(
                            SyntaxStatementSet(
                                lisp_statement,

check_atom_identifier(lisp_statement.elements[1]),

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[2])
                            )
                        )
                    elif statement_name.value == 'if':
                        if len(lisp_statement.elements) != 4:
                            panic(f'Error: If statement list must
have 4 elements {at_line(lisp_statement.lparen)}')
                        statements.syntax_list.append(
                            SyntaxStatementIf(
                                lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1]),

syntax_parse_statement_list(lisp_statement.elements[2]),

syntax_parse_statement_list(lisp_statement.elements[3]),
                            )
                        )
                    elif statement_name.value == 'while':
                        if len(lisp_statement.elements) != 3:
                            panic(f'Error: While statement list must
have 3 elements {at_line(lisp_statement.lparen)}')
                        statements.syntax_list.append(
                            SyntaxStatementWhile(
                                lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1]),

syntax_parse_statement_list(lisp_statement.elements[2]),
                            )
```

```
                        )
                    elif statement_name.value == 'return':
                        if len(lisp_statement.elements) != 2:
                            panic(f'Error: Return statement list
must have 2 elements {at_line(lisp_statement.lparen)}')
                        statements.syntax_list.append(
                            SyntaxStatementReturn(
                                lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1])
                            )
                        )
                    else:
                        panic(f'Error: Statement name is not valid
{at_line(statement_name)}')
                else:
                    panic(f'Error: Statement name must be an atom
{at_line(statement_name.lparen)}')
            else:
                panic(f'Error: Statement must be a list
{at_line(lisp_statement)}')
        return statements
    else:
        panic(f'Error: Statement list must be a list
{at_line(lisp_statement_list)}')

class SyntaxFunctionDefinition(typing.NamedTuple):
    lisp_list: LispList
    name: TokenIdentifier
    return_type: VarTypePair
    arguments: SyntaxList[VarTypePair]
    variables: SyntaxList[VarTypePair]
    statements: SyntaxList[SyntaxStatement]

def syntax_parse_function_definitions(lisp_tree: LispList) ->
typing.Iterator[SyntaxFunctionDefinition]:
    for fn_def in lisp_tree.elements:
        if isinstance(fn_def, LispList):
            if len(fn_def.elements) != 6:
                panic('')
            if isinstance(fn_def.elements[0], TokenIdentifier):
                if fn_def.elements[0].value != 'fn':
                    panic(f'Error: Function must start with fn
{at_line(fn_def.elements[0])}')
                if isinstance(fn_def.elements[1], TokenIdentifier):
                    if
IDENTIFIER_RE.fullmatch(fn_def.elements[1].value) is None:
                        panic(f'Error: Function name does not match
identifier pattern {at_line(fn_def.elements[1])}')
```

```
                    syn_fn_def_name = fn_def.elements[1]
                    if isinstance(fn_def.elements[2],
TokenIdentifier):
                        if fn_def.elements[2].value not in VarType:
                            panic(f'Error: Function return type is
not valid {at_line(fn_def.elements[2])}')
                        syn_fn_def_return_type =
VarTypePair(fn_def.elements[2], VarType(fn_def.elements[2].value))
                        syn_fn_def_arguments =
syntax_parse_arg_list(fn_def.elements[3])
                        syn_fn_def_variables =
syntax_parse_arg_list(fn_def.elements[4])
                        syn_fn_def_statements =
syntax_parse_statement_list(fn_def.elements[5])
                        yield SyntaxFunctionDefinition(fn_def,
syn_fn_def_name, syn_fn_def_return_type, syn_fn_def_arguments,
syn_fn_def_variables, syn_fn_def_statements)
                    else:
                        panic(f'Error: Function return type must be
an atom {at_line(fn_def.elements[2].lparen)}')
                else:
                    panic(f'Error: Function name must be an atom
{at_line(fn_def.elements[1].lparen)}')
            else:
                panic(f'Error: Function must start with atom
{at_line(fn_def.elements[0].lparen)}')
        else:
            panic(f'Error: Function definition must be a list
{at_line(fn_def)}')



def check_indetifier_is_function_param_or_var(fn_def:
SyntaxFunctionDefinition, token: TokenIdentifier):
    for var in fn_def.arguments.syntax_list:
        if var.token.value == token.value:
            return
    for var in fn_def.variables.syntax_list:
        if var.token.value == token.value:
            return
    panic(f'Error: token "{token.value}" is not a variable nor
parameter {at_line(token)}')

def get_variable_type(fn_def: SyntaxFunctionDefinition, token:
TokenIdentifier) -> VarType:
    for var in fn_def.arguments.syntax_list:
        if var.token.value == token.value:
            return var.vartype
    for var in fn_def.variables.syntax_list:
```

```
        if var.token.value == token.value:
            return var.vartype
    panic(f'Error: token "{token.value}" is not a variable nor
parameter {at_line(token)}')

def get_type_var_or_const(fn_def: SyntaxFunctionDefinition, var:
SyntaxVariableOrConstant) -> VarType:
    if isinstance(var, SyntaxVariable):
        return get_variable_type(fn_def, var.value)
    elif isinstance(var, SyntaxConstantBool):
        return VarType.BOOL
    elif isinstance(var, SyntaxConstantFloat):
        return VarType.FLOAT
    elif isinstance(var, SyntaxConstantInt):
        return VarType.INT
    else:
        panic('unreachable')

class BuiltinFunction(typing.NamedTuple):
    name: str
    return_type: VarType
    argtypes: tuple[VarType, ...]
    asm_code: str


BULTIN_FUNCTIONS = (
    BuiltinFunction('==', VarType.BOOL, (VarType.INT, VarType.INT),
"""
cmp x0, x1
cset x0, eq
ret
"""
    ),
    BuiltinFunction('==', VarType.BOOL, (VarType.BOOL,
VarType.BOOL),
"""
cmp x0, x1
cset x0, eq
ret
"""
),
    BuiltinFunction('*', VarType.INT, (VarType.INT, VarType.INT),
"""
mul x0, x0, x1
ret
"""
),
    BuiltinFunction('/', VarType.INT, (VarType.INT, VarType.INT),
"""
```

```
sdiv x0, x0, x1
ret
"""
),
    BuiltinFunction('+', VarType.INT, (VarType.INT, VarType.INT),
"""
add x0, x0, x1
ret
"""
),
    BuiltinFunction('-', VarType.INT, (VarType.INT, VarType.INT),
"""
sub x0, x0, x1
ret
"""
),

    BuiltinFunction('>', VarType.BOOL, (VarType.INT, VarType.INT),
"""
cmp x0, x1
cset x0, gt
ret
"""
),
    BuiltinFunction('<', VarType.BOOL, (VarType.INT, VarType.INT),
"""
cmp x0, x1
cset x0, lt
ret
"""
),
    BuiltinFunction('>=', VarType.BOOL, (VarType.INT, VarType.INT),
"""
cmp x0, x1
cset x0, ge
ret
"""
),
    BuiltinFunction('<=', VarType.BOOL, (VarType.INT, VarType.INT),
"""
cmp x0, x1
cset x0, le
ret
"""
),
    BuiltinFunction('*', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
"""
fmul d0, d0, d1
```

```
ret
"""
),
    BuiltinFunction('/', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
"""
fdiv d0, d0, d1
ret
"""
),
    BuiltinFunction('+', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
"""
fadd d0, d0, d1
ret
"""
),
    BuiltinFunction('-', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
"""
fsub d0, d0, d1
ret
"""
),
    BuiltinFunction('==', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
"""
fcmp d0, d1
cset x0, eq
ret
"""
),
    BuiltinFunction('>', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
"""
fcmp d0, d1
cset x0, gt
ret
"""
),
    BuiltinFunction('<', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
"""
fcmp d0, d1
cset x0, lt
ret
"""
),
```

```
    BuiltinFunction('>=', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
"""
fcmp d0, d1
cset x0, ge
ret
"""
),
    BuiltinFunction('<=', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
"""
fcmp d0, d1
cset x0, le
ret
"""
),
)

def find_matching_func_def_for_func_call(
    func_call: SyntaxFunctionCall,
    fn_defs: typing.Iterable[SyntaxFunctionDefinition],
    current_fn_def: SyntaxFunctionDefinition
) -> BuiltinFunction | SyntaxFunctionDefinition:
    func_call_types = list(get_type_var_or_const(current_fn_def, el)
for el in func_call.arguments)
    for fn_def in fn_defs:
        if func_call.name.value != fn_def.name:
            continue
        if len(func_call.arguments) !=
len(fn_def.arguments.syntax_list):
            continue
        fn_def_arg_types = list(el.vartype for el in
fn_def.arguments.syntax_list)
        if any(t1 != t2 for t1, t2 in zip(func_call_types,
fn_def_arg_types)):
            continue
        return fn_def
    for builtin in BULTIN_FUNCTIONS:
        if func_call.name.value != builtin.name:
            continue
        if len(func_call.arguments) != len(builtin.argtypes):
            continue
        if any(t1 != t2 for t1, t2 in zip(func_call_types,
builtin.argtypes)):
            continue
        return builtin
    panic(f'Error: Function call does not match any functions
{at_line(func_call.lisp_list)}')
```

```
def check_statement_list(fn_defs:
typing.Iterable[SyntaxFunctionDefinition], current_fn_def:
SyntaxFunctionDefinition, statements: SyntaxList[SyntaxStatement]):
    for statement in statements.syntax_list:
        if isinstance(statement, SyntaxStatementSet):

check_indetifier_is_function_param_or_var(current_fn_def,
statement.dest)
            dest_type = get_variable_type(current_fn_def,
statement.dest)
            if isinstance(statement.src, SyntaxVariableOrConstant):
                src_type = get_type_var_or_const(current_fn_def,
statement.src)
            elif isinstance(statement.src, SyntaxFunctionCall):
                func =
find_matching_func_def_for_func_call(statement.src, fn_defs,
current_fn_def)
                src_type = func.return_type
            else:
                panic('Compiler Error: unreachable')
            if dest_type != src_type:
                panic(f'Error: type mismatch between dest and src
values in set statement {at_line(statement.dest)}')
        elif isinstance(statement, SyntaxStatementIf):
            if isinstance(statement.condition,
SyntaxVariableOrConstant):
                ret_type = get_type_var_or_const(current_fn_def,
statement.condition)
            elif isinstance(statement.condition,
SyntaxFunctionCall):
                func =
find_matching_func_def_for_func_call(statement.condition, fn_defs,
current_fn_def)
                if isinstance(func, SyntaxFunctionDefinition):
                    ret_type = func.return_type.vartype
                elif isinstance(func, BuiltinFunction):
                    ret_type = func.return_type
                else:
                    panic(f'unreachable')
            else:
                panic(f'unreachable')
            if ret_type != VarType.BOOL:
                panic(f'Error: Condition must be bool
{at_line(statement.lisp_list)}')

            check_statement_list(fn_defs, current_fn_def,
statement.true_branch)
            check_statement_list(fn_defs, current_fn_def,
statement.false_branch)
```

```
        elif isinstance(statement, SyntaxStatementWhile):
            if isinstance(statement.condition,
SyntaxVariableOrConstant):
                ret_type = get_type_var_or_const(current_fn_def,
statement.condition)
            elif isinstance(statement.condition,
SyntaxFunctionCall):
                func =
find_matching_func_def_for_func_call(statement.condition, fn_defs,
current_fn_def)
                if isinstance(func, SyntaxFunctionDefinition):
                    ret_type = func.return_type.vartype
                elif isinstance(func, BuiltinFunction):
                    ret_type = func.return_type
                else:
                    panic(f'unreachable')
            else:
                panic(f'unreachable')
            if ret_type != VarType.BOOL:
                panic(f'Error: Condition must be bool
{at_line(statement.lisp_list)}')
            check_statement_list(fn_defs, current_fn_def,
statement.statements)
        elif isinstance(statement, SyntaxStatementReturn):
            if isinstance(statement.value,
SyntaxVariableOrConstant):
                ret_type = get_type_var_or_const(current_fn_def,
statement.value)
            elif isinstance(statement.value, SyntaxFunctionCall):
                func =
find_matching_func_def_for_func_call(statement.value, fn_defs,
current_fn_def)
                if isinstance(func, SyntaxFunctionDefinition):
                    ret_type = func.return_type.vartype
                elif isinstance(func, BuiltinFunction):
                    ret_type = func.return_type
                else:
                    panic(f'unreachable')
            else:
                panic(f'unreachable')
            if current_fn_def.return_type.vartype != ret_type:
                panic(f'Function does not return value of return
type {at_line(statement.lisp_list)}')
        else:
            panic(f'unreachable')

import itertools
```

```
def check_function_definitions(fn_defs:
tuple[SyntaxFunctionDefinition, ...]):
    for fn_def in fn_defs:
        for el in fn_def.statements.syntax_list[:-1]:
            if isinstance(el, SyntaxStatementReturn):
                panic(f'Error: Return statement must be the last one
{at_line(el.lisp_list)}')
        if len(fn_def.statements.syntax_list) == 0:
            panic(f'Error: Function statement list must have at
least one statement {at_line(fn_def.statements.lisp_list)}')
        if not isinstance(fn_def.statements.syntax_list[-1],
SyntaxStatementReturn):
            panic(f'Error: Last statement in function definition
statement list must be return statement
{at_line(fn_def.statements.lisp_list)}')

        VarTypePair.vartype
        it_first = itertools.chain(fn_def.arguments.syntax_list,
fn_def.variables.syntax_list)
        it_second = itertools.chain(fn_def.arguments.syntax_list,
fn_def.variables.syntax_list)
        for i_one, el_one in enumerate(it_first):
            for i_two, el_two in enumerate(it_second):
                if i_one == i_two:
                    continue
                if el_one.token.value == el_two.token.value:
                    panic(f'Error: Duplicate argument name
{at_line(el_one.token)} and {at_line(el_two.token)}')

    for i_one, fn_def_one in enumerate(fn_defs):
        args_one = [el.vartype for el in
fn_def_one.arguments.syntax_list]
        for i_two, fn_def_two in enumerate(fn_defs):
            if i_one == i_two:
                continue
            if fn_def_one.name.value != fn_def_two.name.value:
                continue
            args_two = [el.vartype for el in
fn_def_two.arguments.syntax_list]
            if len(args_one) != len(args_two):
                continue
            if any(a1 != a2 for a1, a2 in zip(args_one, args_two)):
                continue
            panic(f'Error: Duplicate function definitions
{at_line(fn_def_one.lisp_list)} and
{at_line(fn_def_two.lisp_list)}')

    for fn_def in fn_defs:
```

```
        args_one = [el.vartype for el in
fn_def.arguments.syntax_list]
        for builtin_def in BULTIN_FUNCTIONS:
            if builtin_def.name != fn_def.name.value:
                continue
            if len(args_one) != len(builtin_def.argtypes):
                continue
            if any(a1 != a2 for a1, a2 in zip(args_one,
builtin_def.argtypes)):
                continue
            panic(f'Error: Duplicate function definition with
builtin "{builtin_def.name}" {at_line(fn_def_one.lisp_list)}')

    for fn_def in fn_defs:
        check_statement_list(fn_defs, fn_def, fn_def.statements)

def main():
    filepath = 'example.txt'
    lparens: list[TokenLparen] = []
    lisp_tree = LispList(TokenLparen(0, 0))
    build_lisp_tree(lisp_tree, tokenize(filepath), lparens)

    if len(lparens) > 0:
        panic(f'Error: Unmatched paren {at_line(lparens[-1])}')
    del lparens

    fn_defs = tuple(syntax_parse_function_definitions(lisp_tree))
    check_function_definitions(fn_defs)



if __name__ == '__main__':
    main()
```

3 ВИСНОВОК

У ході роботи створено просту Lisp-подібну мову з власним лексичним, синтаксичним і семантичним аналізаторами. Реалізація забезпечує перевірку структури, типів і логіки програм, а система діагностики чітко локалізує помилки у форматі `Error: ...`. Тестування підтвердило правильність роботи всіх етапів аналізу та повну відповідність реалізації специфікації мови.