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

Лабораторна робота № 1

ПРОЄКТУВАННЯ ТА СПЕЦИФІКАЦІЯ
ПРОСТОЇ ІМПЕРАТИВНОЇ МОВИ ПРОГРАМУВАННЯ
ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ
(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 ВСТУП

Мова програмування LAB — імперативна мова загального призначення з LISP-подібним синтаксисом. Назва промовляється як "лаб".

1.1 Обробка

Програма, написана мовою LAB, подається на вхід транслятора для перетворення до цільової форми. Результат трансляції виконується у системі часу виконання, для чого приймає вхідні дані та надає результат виконання програми.

Трансляція передбачає фази лексичного, синтаксичного та семантичного аналізу, а також фазу генерації коду.

1.2 Нотація

Для опису мови LAB використовується граматика ANTLR4. Граматика описує синтаксичні правила мови у форматі, що підтримується генератором парсерів ANTLR4.

ANTLR4 (ANother Tool for Language Recognition) — потужний інструмент для створення парсерів, який підтримує LL(*) граматики. Використання ANTLR4 дозволяє автоматично генерувати лексичний та синтаксичний аналізатори, а також забезпечує зручні засоби для обробки дерева синтаксичного розбору.

У граматиці ANTLR4 правила, що починаються з великої літери, є лексичними правилами (токенами), а правила з маленької літери — синтаксичними правилами. Фрагменти (fragment) використовуються для допоміжних лексичних правил, які не створюють окремих токенів.

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 Пробільні символи

Пробільні символи ігноруються лексичним аналізатором. У мові LAB пробільні символи (пробіл, табуляція, символи нового рядка та повернення каретки) використовуються тільки для розділення токенів та не мають семантичного значення. Директива `-> skip` в ANTLR4 вказує лексеру проігнорувати ці символи та не передавати їх парсеру. Це дозволяє писати код з довільною кількістю пробілів та переносів рядків для покращення читабельності.

Програміст може використовувати відступи та пробіли для форматування коду згідно з власними уподобаннями, оскільки вони не впливають на семантику програми. Рекомендується використовувати послідовне форматування для покращення читабельності коду.

```
WS: [ \t\r\n]+ -> skip;
```

Рисунок 2.1 — Пробільні символи

2.2 Дужки

Дужки є основними структурними елементами мови LAB. Вони використовуються для групування виразів та визначення меж функціональних конструкцій. Кожна відкриваюча дужка '(' повинна мати відповідну закриваючу дужку ')'.

```
LPAREN: '(';
RPAREN: ')';
```

Рисунок 2.2 — Ліва дужка

Рисунок 2.3 — Права дужка

2.3 Типи даних

Мова LAB підтримує чотири базові типи даних.

```
TYPE: TYPE_INTEGER | TYPE_REAL | TYPE_STRING | TYPE_BOOL;
TYPE_INTEGER: 'int';
TYPE_REAL: 'real';
```

```
TYPE_STRING: 'string';
TYPE_BOOL: 'bool';
```

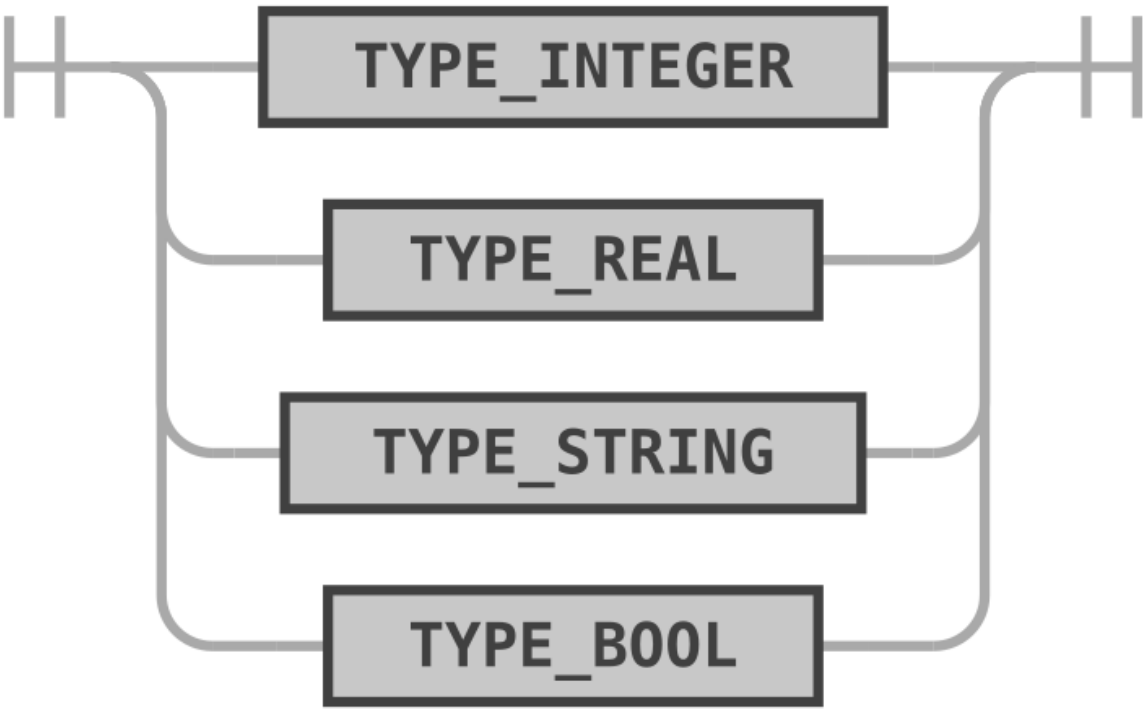

Рисунок 2.4 — Типи даних

2.4 Ключові слова

```
KEYWORD_FN: 'fn';
KEYWORD_LET: 'let';
KEYWORD_IF: 'if';
KEYWORD_WHILE: 'while';
```

Рисунок 2.5 — Ключове слово “fn”

Рисунок 2.6 — Ключове слово “if”

Рисунок 2.7 — Ключове слово “let”

Рисунок 2.8 — Ключове слово “while”

2.5 Літерали

Літерали використовуються для представлення постійних значень.

```
LITERAL: LITERAL_BOOL | LITERAL_REAL | LITERAL_INTEGER |
LITERAL_STRING;
LITERAL_BOOL: 'true' | 'false';
LITERAL_REAL: ('+'|'-')?DIGIT+ '.' DIGIT+;
LITERAL_INTEGER: ('+'|'-')?DIGIT+;
LITERAL_STRING: '"' ( ~["\\\r\n] | '\\' . )* '"';
fragment DIGIT : [0-9];
```

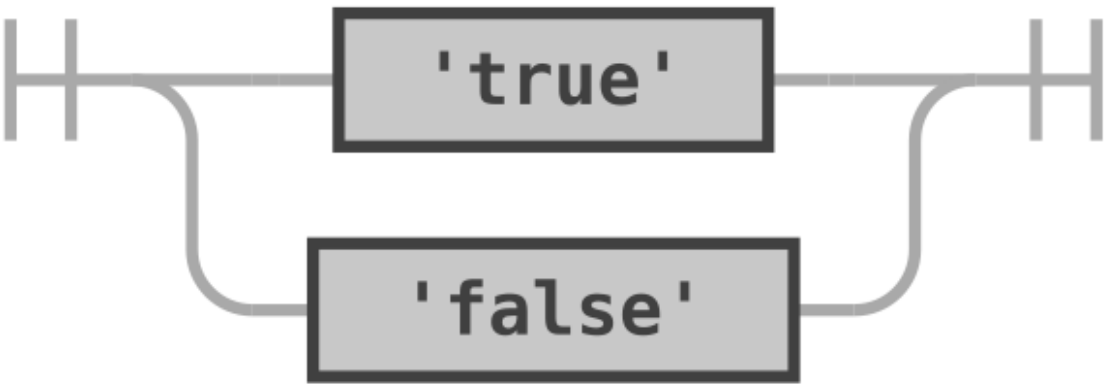

Рисунок 2.10 — Булевий літерал

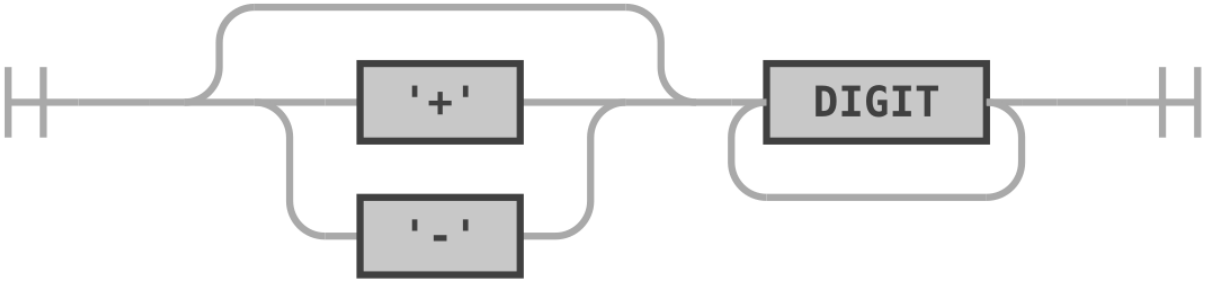

Рисунок 2.11 — Ціле число

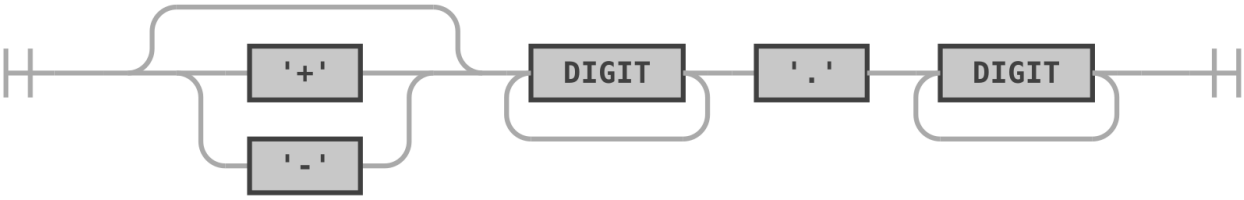

Рисунок 2.12 — Дійсне число

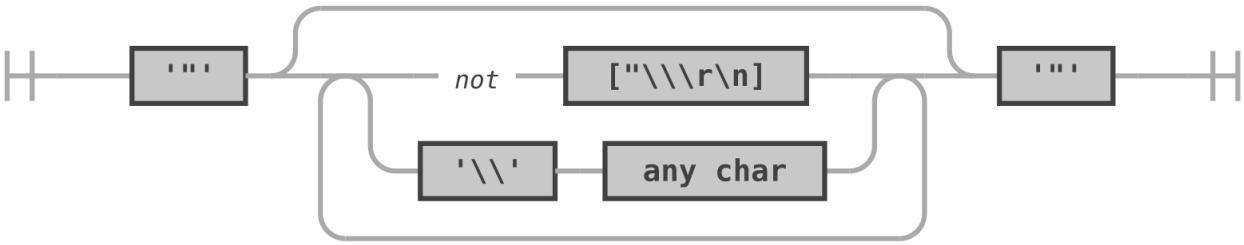

Рисунок 2.13 — Рядок

Рисунок 2.14 — Цифра

2.6 Ідентифікатори

Ідентифікатори можуть містити літери, підкреслення та символи операторів. Це дозволяє використовувати символічні імена функцій як `+`, `-`, `*`, `/`, `==`, `le` тощо.

Унікальною особливістю мови LAB є можливість використання математичних та логічних символів як частини ідентифікаторів. Це успадковано від традицій LISP, де оператори є звичайними функціями з символічними іменами. Таким чином, `+` є просто ідентифікатором функції додавання, а не спеціальним синтаксичним символом.

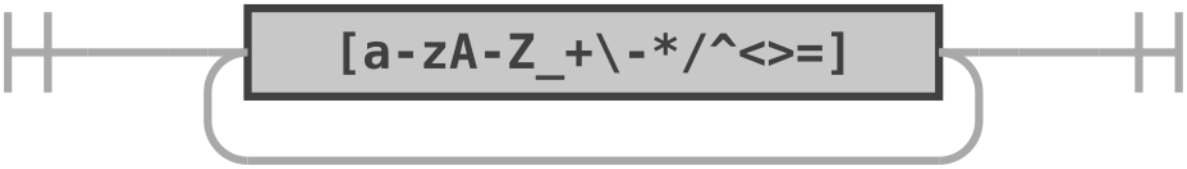

Рисунок 2.15 — Ідентифікатор

3 СИНТАКСИЧНА СТРУКТУРА

3.1 Програма

Програма у мові LAB є колекцією функцій, серед яких обов'язково повинна бути присутня функція з іменем `main`, що служить точкою входу для виконання програми. Мінімальна програма містить тільки одну функцію `main`, але зазвичай програми включають додаткові допоміжні функції.

Порядок визначення функцій у файлі не має значення - функції можуть викликати одна одну незалежно від порядку їх оголошення. Це дозволяє організовувати код у зручному для розробника порядку.

Кожне визначення функції є самостійною сутністю з власною областю видимості. Глобальних змінних у мові LAB не передбачено - вся комунікація між функціями відбувається через параметри та повернені значення.

Програма повинна містити принаймні одну функцію. Порожні програми не допускаються та викликають синтаксичну помилку.

```
program: funcDef+;
```

Рисунок 3.1 — Програма

3.2 Визначення функції

Кожна функція має ім'я, список параметрів (може бути порожнім) та тіло функції.

Визначення функції починається з відкриваючої дужки, за якою слідує ключове слово `fn`, ім'я функції та список параметрів у дужках. Після цього йде тіло функції, також укладене в дужки, і закриваюча дужка всієї конструкції.

Список параметрів може бути порожнім для функцій, які не приймають аргументів. Якщо параметри присутні, кожен параметр оголошується як пара `(ім'я тип)`, де ім'я - це ідентифікатор параметра, а тип - один з базових типів мови.

Параметри функції створюють локальну область видимості всередині тіла функції. Кожен параметр діє як локальна змінна, ініціалізована значенням відповідного аргументу при виклику функції.

Функції у LAB мають лексичну область видимості - вони можуть звертатися тільки до своїх параметрів та локально оголошених змінних. Доступ до змінних з зовнішніх областей видимості не підтримується.

Тіло функції представляє собою список інструкцій, який виконується послідовно при виклику функції. Функція завершує виконання після обробки всіх інструкцій у своєму тілі.

```
funcDef: LPAREN KEYWORD_FN ID LPAREN funcParamList? RPAREN
statementsList RPAREN;
funcParamList: funcParam+;
funcParam: LPAREN ID TYPE RPAREN;
```

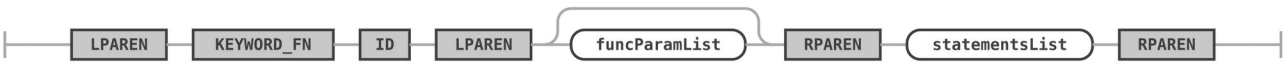

Рисунок 3.2 — Функція

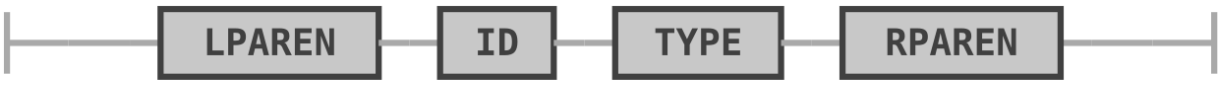

Рисунок 3.3 — Параметр функції

Рисунок 3.4 — Список параметрів функції

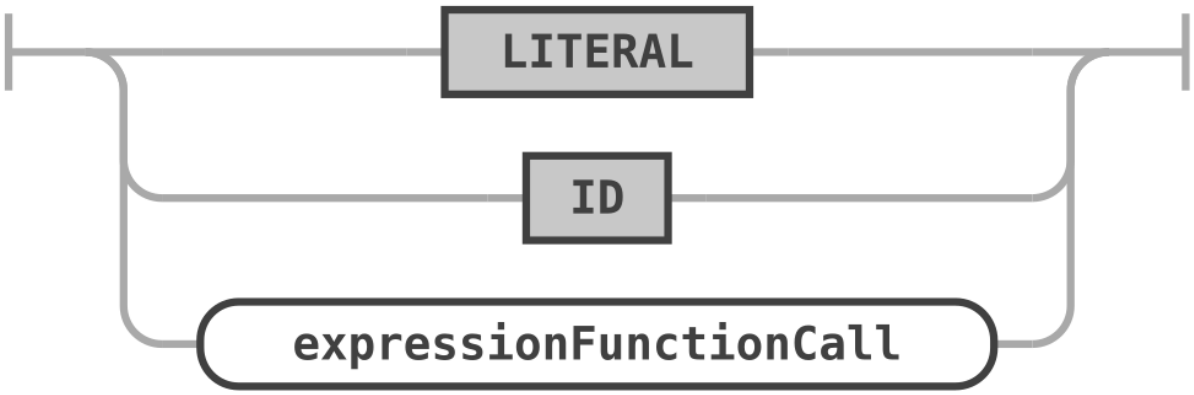

Рисунок 3.5 — Аргумент виклику функції

Приклад:

```
(fn main ((a int) (b real) (c string)) (
    // тіло функції
))
```

3.3 Інструкції

Інструкції визначають дії, які виконуються в програмі. У мові LAB підтримуються чотири типи інструкцій: оголошення змінних, умовні інструкції, цикли та виклики функцій.

Список інструкцій завжди укладається в дужки і може містити нуль або більше окремих інструкцій. Порожній список інструкцій `()` є валідним та не виконує жодних дій.

Інструкції виконуються послідовно у порядку їх появи в списку. Кожна інструкція повністю завершується перед початком виконання наступної, що забезпечує детерміновану поведінку програми.

Виклики функцій можуть використовуватися як окремі інструкції (для функцій з побічними ефектами, як `display`) або як частина інших конструкцій (для отримання значень у виразах).

Область видимості змінних, оголошених усередині списку інструкцій, обмежується цим списком та всіма вкладеними списками інструкцій.

```
statement
  : statementVariableDeclaration
  | statementIf
  | statementWhile
  | expressionFunctionCall
  ;

statementsList: LPAREN statement* RPAREN;
```

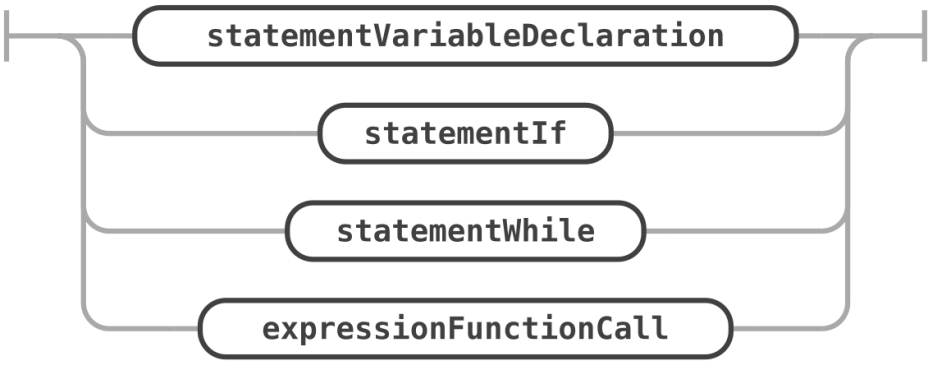

Рисунок 3.6 — Інструкція

3.3.1 Оголошення змінних

Оголошення змінної створює нове ім'я в локальній області видимості та пов'язує його з обчисленим значенням виразу. Ключове слово `let` вказує на створення нової змінної, за яким слідує ім'я змінної та вираз для ініціалізації.

Значення змінної обчислюється один раз у момент оголошення шляхом обчислення відповідного виразу. Тип змінної виводиться автоматично на основі типу значення, що присвоюється.

Область видимості змінної починається з місця її оголошення та поширюється до кінця поточного списку інструкцій. Змінна може затінювати (shadow) змінні з зовнішніх областей видимості з тим самим іменем.

```
statementVariableDeclaration: LPAREN KEYWORD_LET ID
functionCallArgument RPAREN;
```

Рисунок 3.7 — Оголошення змінної

Приклад:

```
(let d (concat c "Hello KPI"))
(let i 0)
```

3.3.2 Умовна інструкція

Цикл while забезпечує повторне виконання блоку інструкцій доти, доки умова залишається істинною. Конструкція складається з ключового слова while, умовного виразу та списку інструкцій для виконання.

Умова перевіряється перед кожною ітерацією циклу, включаючи першу. Якщо умова хибна з самого початку, тіло циклу не виконується жодного разу. Це робить while циклом з передумовою.

Умовний вираз повинен повертати значення типу bool. Якщо умова повертає true, виконується тіло циклу, після чого умова перевіряється знову. Цикл завершується, коли умова стає хибною (false).

Програміст повинен забезпечити, що умова циклу в кінцевому підсумку стане хибною, інакше виникне нескінченний цикл. Зазвичай це досягається зміною змінних, які використовуються в умові, всередині тіла циклу.

Тіло циклу може містити будь-які валідні інструкції, включаючи інші цикли (вкладені цикли) та умовні конструкції. Кожна ітерація циклу виконує всі інструкції в тілі послідовно.

```
statementIf: LPAREN KEYWORD_IF expressionFunctionCall statementsList
statementsList RPAREN;
```

Рисунок 3.8 — Умовна інструкція

Приклад:

```
(if (== 55 104) (
    (display "TRUE")
) (
    (display "FALSE")
))
```

3.3.3 Інструкція циклу

Цикл while забезпечує повторне виконання блоку інструкцій доти, доки умова залишається істинною. Конструкція складається з ключового слова while, умовного виразу та списку інструкцій для виконання.

Умова перевіряється перед кожною ітерацією циклу, включаючи першу. Якщо умова хибна з самого початку, тіло циклу не виконується жодного разу. Це робить while циклом з передумовою.

Умовний вираз повинен повертати значення типу bool. Якщо умова повертає true, виконується тіло циклу, після чого умова перевіряється знову. Цикл завершується, коли умова стає хибною (false).

Програміст повинен забезпечити, що умова циклу в кінцевому підсумку стане хибною, інакше виникне нескінченний цикл. Зазвичай це досягається зміною змінних, які використовуються в умові, всередині тіла циклу.

Тіло циклу може містити будь-які валідні інструкції, включаючи інші цикли (вкладені цикли) та умовні конструкції. Кожна ітерація циклу виконує всі інструкції в тілі послідовно.

```
statementWhile: LPAREN KEYWORD_WHILE expressionFunctionCall
statementsList RPAREN;
```

Рисунок 3.9 — Інструкція циклу

Приклад:

```
(while (le i 5) (
    (display (* 2 i))
    (set i (+ i 1))
))
```

3.3.4 Виклики функцій

Виклики функцій мають префіксну нотацію. Аргументами можуть бути літерали, ідентифікатори або інші виклики функцій.

Виклик функції являє собою основну операційну одиницю мови LAB. Усі обчислення, включаючи арифметичні операції, порівняння та маніпуляції з даними, виконуються через виклики функцій. Це відображає функціональну природу мови та її LISP-походження.

Префіксна нотація означає, що ім'я функції завжди стоїть першим, а за ним слідують аргументи. Така нотація забезпечує однорідність синтаксису та спрощує парсинг виразів з різною кількістю аргументів.

Аргументи функції обчислюються зліва направо перед передачею до функції. Це гарантує детерміновану поведінку програми у випадках, коли обчислення аргументів має побічні ефекти.

Виклики функцій можуть бути вкладеними без обмежень глибини. Внутрішні виклики завжди обчислюються перед зовнішніми, дотримуючись правил пріоритету операцій.

Кількість аргументів у виклику повинна точно відповідати кількості параметрів у визначенні функції. Невідповідність кількості аргументів викликає помилку під час компіляції.

```
expressionFunctionCall: LPAREN ID functionCallArgument* RPAREN;
functionCallArgument: LITERAL | ID | expressionFunctionCall;
```

Рисунок 3.10 — Виклик фнукції

Приклад:

```
(display "Hello")
(+ 2 3)
(* 2 (- a (/ i b)))
(concat c "Hello KPI")
```

```
grammar lab;

WS: [ \t\r\n]+ -> skip;

TYPE: TYPE_INTEGER | TYPE_REAL | TYPE_STRING | TYPE_BOOL;
TYPE_INTEGER: 'int';
TYPE_REAL: 'real';
TYPE_STRING: 'string';
TYPE_BOOL: 'bool';

KEYWORD_FN: 'fn';
KEYWORD_LET: 'let';
KEYWORD_IF: 'if';
KEYWORD_WHILE: 'while';

LPAREN: '(';
RPAREN: ')';

statement
 : statementVariableDeclaration
 | statementIf
 | statementWhile
 | expressionFunctionCall
 ;

functionCallArgument: LITERAL | ID | expressionFunctionCall;
expressionFunctionCall: LPAREN ID functionCallArgument* RPAREN;
statementVariableDeclaration: LPAREN KEYWORD_LET ID
functionCallArgument RPAREN;
statementsList: LPAREN statement* RPAREN;
statementIf: LPAREN KEYWORD_IF expressionFunctionCall statementsList
statementsList RPAREN;
statementWhile: LPAREN KEYWORD_WHILE expressionFunctionCall
statementsList RPAREN;

funcParam: LPAREN ID TYPE RPAREN;
funcParamList: funcParam+;
funcDef: LPAREN KEYWORD_FN ID LPAREN funcParamList? RPAREN
statementsList RPAREN;

program: funcDef+;

LITERAL: LITERAL_BOOL | LITERAL_REAL | LITERAL_INTEGER |
LITERAL_STRING;
LITERAL_BOOL: 'true' | 'false';
LITERAL_REAL: DIGIT+ '.' DIGIT* | '.' DIGIT+;
```

```
LITERAL_INTEGER: DIGIT+;
LITERAL_STRING: '"' ( ~["\\\r\n] | '\\' . )* '"';

ID : [a-zA-Z_+\-*/^<>=]+;
fragment DIGIT : [0-9];
```

5 ПРИКЛАД ПРОГРАМИ

```
(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) (
        (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))
    ))
))
```