НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет прикладної математики

Кафедра системного програмування і
спеціалізованих комп'ютерних систем

«До захисту допущено»
Завідувач кафедри
_____ В.П. Тарасенко
(підпис)
"___" _____ 2015 р.

# Дипломний проект

**освітньо-кваліфікаційного рівня "Бакалавр"**

з напряму підготовки 6.050102 "Комп'ютерна інженерія"

на тему: «Система оцінювання якості програм мовою Lisp»

Виконав: студент 4 курсу, групи КВ-12
Солодрай Ігор Іванович _____
(підпис)

Керівник доц., к.т.н. Марченко О. І. _____
(підпис)

Рецензент с.н.с., к.т.н. Сергієнко А. М. _____
(підпис)

Консультант по нормо-контролю, доц. Плахотний М.В. _____
(підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2015

**Національний технічний університет України**

**“Київський політехнічний інститут”**

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп’ютерних систем

Освітньо-кваліфікаційний рівень “Бакалавр”

Напрям підготовки 6.050102 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ В.П. Тарасенко

“___” _____ 2015 р.

# З А В Д А Н Н Я

**НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ**

Солодраю Ігорю Івановичу

1. Тема роботи: «Система оцінювання якості програм мовою Lisp», керівник роботи Марченко Олександр Іванович, к.т.н., доцент,
затверджені наказом по університету від “19” травня 2015 року № 1039-С.

2. Строк подання студентом роботи: “16” червня 2015 р.

3. Вихідні дані для дипломної роботи:
   - науково-технічна література по створенню програмного комплексу, що вирішує задачі статичного аналізу програмного коду, описує й аналізує алгоритми, що були реалізовані.

4. Перелік задач, які потрібно вирішити:
   - реалізувати алгоритми обчислення шести типів метрик для мови Lisp;
   - провести аналіз існуючих рішень;
   - обґрунтувати вибір дипломної роботи;
   - розробити програмну модель системи;
   - проаналізувати реалізовані алгоритми.

5. Перелік обов’язкового ілюстративного матеріалу:
   - Система оцінювання якості програм мовою Lisp. Схема структурна;
   - Робота системи. Схема алгоритму;
   - Побудова графа викликів та матриці залежностей імен. Схеми алгоритмів;

– Обчислення рекурсивної складності. Схема алгоритму.

6. Консультанти:

| Питання | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Нормо-контроль | Плахотний М.В., доцент | | |

7. Дата видачі завдання: “31” жовтня 2014 р.

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів виконання дипломної роботи | Строк виконання етапів | Примітка |
|---|---|---|---|
| 1. | чення літератури за тематикою роботи та збір даних | 10.12.2015 | |
| 2. | ведення порівняльного аналізу існуючих додатків та ресурсів що до даної теми | 10.01.2015 | |
| 3. | готовка матеріалів першого розділу дипломної роботи | 30.03.2015 | |
| 4. | готовка матеріалів другого розділу дипломної роботи | 15.03.2015 | |
| 5. | готовка матеріалів третього розділу дипломної роботи | 10.04.2015 | |
| 6. | готовка матеріалів четвертого розділу дипломної роботи | 20.04.2015 | |
| 7. | готовка графічної частини дипломної роботи | 23.05.2015 | |
| 8. | рмлення дипломної роботи | 25.05.2015 | |

**Студент _____ Солодрай І.І.**

**Керівник роботи _____ Марченко О.І.**

# 2. РОЗРОБКА КОМПОНЕНТІВ СИСТЕМИ

## 2.1. Загальна структура системи

Архітектура системи передбачає послідовну залежність модулів, тобто кожен наступний модуль використовує функції попереднього (рис. 2.1). Система складається з модулів, кожен з яких має власне функціональне призначення і в свою чергу складається з менших модулів. Ключові модулі:

- синтаксичний аналізатор;
- модуль допоміжних функцій;
- модуль обчислення метрик;
- головний модуль.

Структуру системи зображено на рис. 2.1.

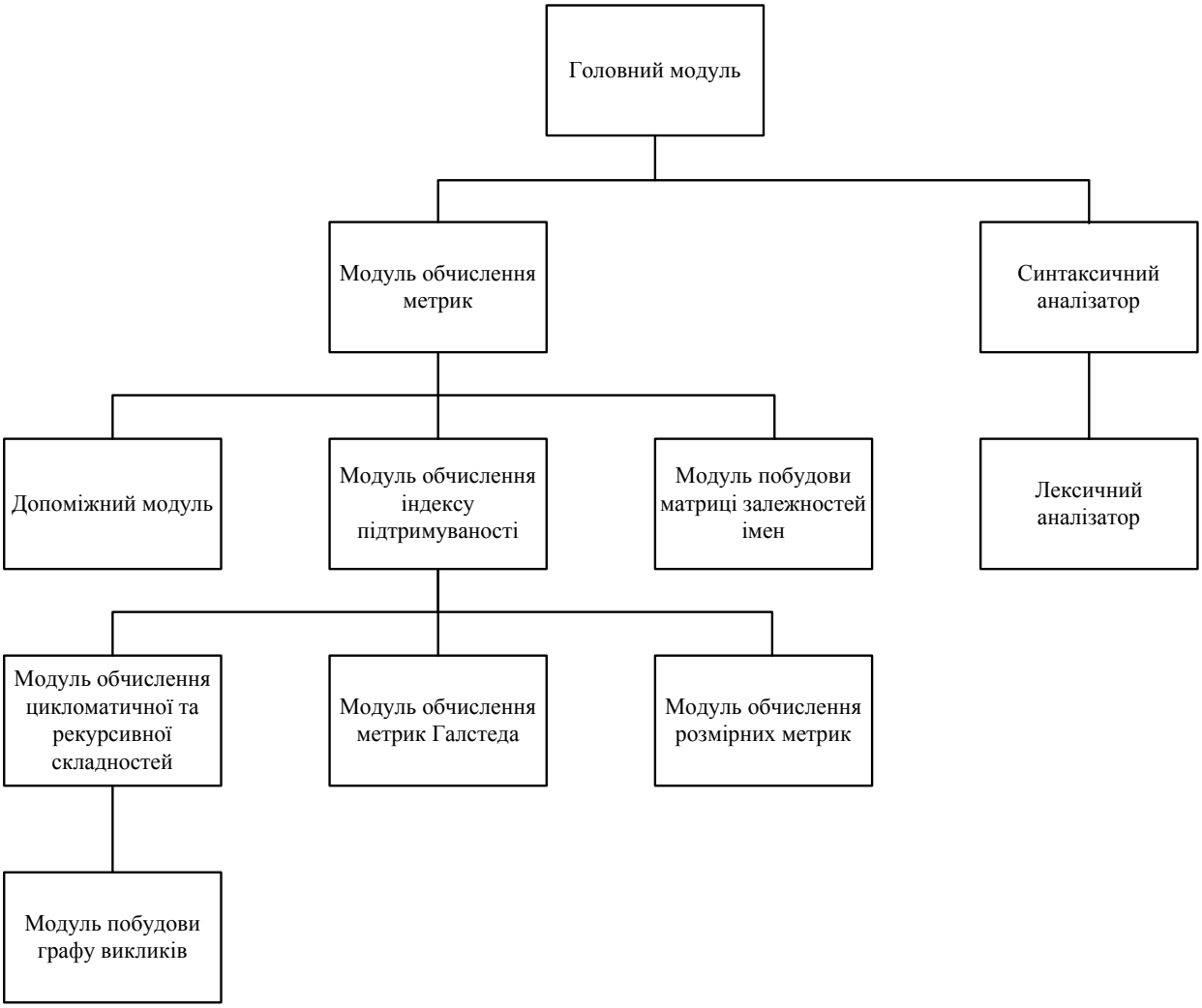


Рис. 2.1. Структура системи

2.1.1. Модулі лексичного та синтаксичного аналізаторів

Лексичний аналізатор – це модуль, що виконує лексичний аналіз тексту вхідної програми. Він зчитує вхідну послідовність символів, і перетворює цю послідовність на рядок лексем, який надалі оброблятиме синтаксичний аналізатор. Під час лексичного аналізу визначаються та локалізуються лексичні помилки. Загалом роботу лексичного аналізатора можна описати в дві стадії: сканування (виділення лексеми із послідовності символів) та аналізу (визначення типу лексеми).

На першій стадії, сканування, робота лексичного аналізатора організована за принципом скінченного автомата. У цьому автоматі міститься інформація про можливі підпослідовності символів, які можуть зустрічатись у вхідній послідовності символів (ідентифікатори, числові константи та ін.). Після фази сканування буде отримано рядок нетипізованих лексем. Для визначення типу лексеми (напр., для ідентифікатора – чи це ключове слово, чи ідентифікатор, визначений користувачем) виконується друга фаза лексичного аналізу. Інформація про тип лексеми отримується завдяки аналізу символів, що формують лексему (напр. крапка в константі каже про те, що це дійсне число) та інформації з таблиці лексем (наперед визначені ідентифікатори та ін.).

Синтаксичний аналізатор на основі побудованої таблиці виконує синтаксичний аналіз рядка лексем, що надходить від лексичного аналізатора. Якщо синтаксичний аналіз виконано успішно, то на виході маємо дерево розбору вхідної програми. У разі ж виявлення синтаксичних помилок – формується відповідне повідомлення про помилку.

2.1.2. Модуль обчислення метрик та допоміжний модуль

Модуль обчислення метрик містить реалізацію таких алгоритмів:

- обчислення розмірних метрик;
- обчислення метрик Галстеда;
- обчислення індексу підтримки;
- побудову графу викликів функцій;
- побудову матриці структури проекту «використання-визначення»;
- обчислення рекурсивної складності.

Цей модуль використовує допоміжний модуль, котрий містить набір функцій для роботи зі внутрішньою формою подання вхідної програми — деревом розбору.

### 2.1.3. Головний модуль

Головний модуль містить точку входу в програму, інтерфейс взаємодії з користувачем та систему виведення результату.

Тут оброблюються параметри, задані користувачем, і викликаються відповідні функції з модуля обчислення метрик. Потім результат виводиться в заданій формі.

### 2.1.4. Загальний алгоритм роботи системи

Алгоритм роботи системи (рис. 2.2) в цілому подібний до загального алгоритму роботи транслятора.

Спочатку відбувається зчитування файлів з текстом вхідної програми. Лексичний аналізатор оброблює вхідний текст і перетворює його на послідовність лексем. Далі синтаксичний аналізатор перетворює її у внутрішню форму подання, яка в нашому випадку є деревом розбору та інформаційними таблицями.

Модуль обчислення метрик працює виключно із внутрішньою формою, виконує задані функції й передає результати в головний модуль, де виконується перетворення їх у форму, придатну для виведення, і власне виведення.

Оскільки розроблена система підтримує конфігурацію обчислення метрик користувачем, загальний алгоритм оброблює задані параметри. Детальна схема алгоритму цього процесу зображена в додатку 1.

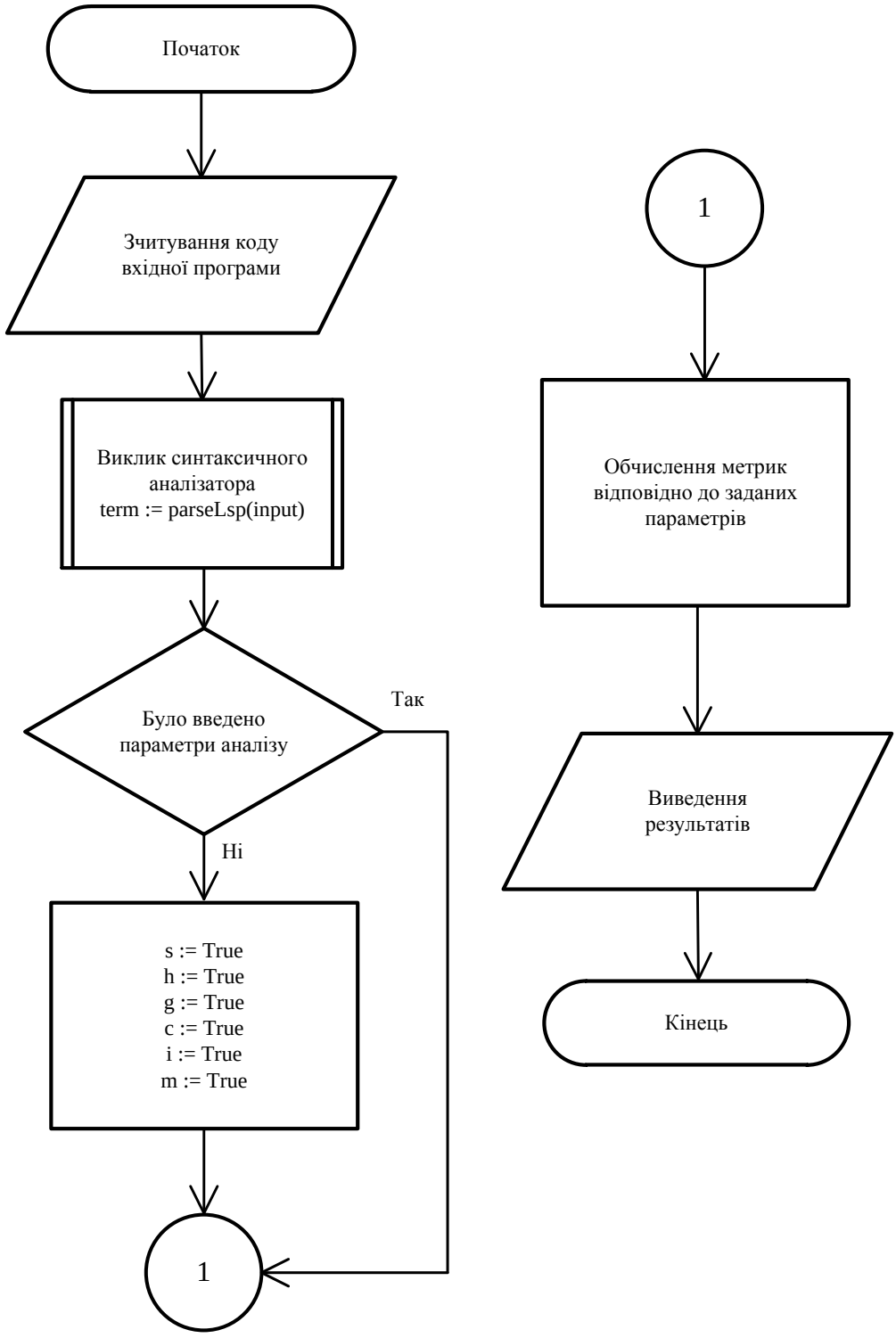


Рис. 2.2. Загальний алгоритм роботи системи

## 2.2. Мова програмування Haskell

Для розробки системи оцінювання якості програм мовою Lisp обрано мову програмування Haskell.

Haskell — стандартизована, винятково функціональна мова програмування з нестрогою семантикою. Названа на честь американського математика Гаскелла Каррі, роботи якого в галузі математичної логіки є базовими для функціонального програмування. Haskell базується на лямбда численні. Найважливішими реалізаціями є компілятор Glasgow Haskell Compiler (GHC) та інтерпретатор Hugs.

Головні особливості мови:

- Haskell є чистою функціональною мовою програмування. Функції не мають жодних побічних ефектів. Це означає, що для одних і тих самих значень вхідних параметрів завжди повертатимуться однакові результати обчислень.

- Функціональні мови програмування відрізняються від процедурних мов програмування тим, що програміст не повинен визначати порядок обчислення функцій. Розробнику слід лише описати залежність між даними, а транслятор вже самотужки визначає порядок обчислень.

- Відсутні оператори зміни значення змінних. Через це, відсутня різниця між константами та змінними. Як наслідок, відпадає необхідність у декларації const або final, які є, наприклад, в мовах програмування C та Java відповідно.

- Відсутня різниця між ідентичністю та рівністю об'єктів.

- Усунення проблем від наявності побічних ефектів, значною мірою полегшує спостереження за послідовністю роботи програми.

- Haskell є мовою програмування з нестрогою семантикою. Обчислюються лише вирази, значення яких необхідне для обчислення результатів.

## 2.3. Розробка синтаксичного аналізатора

Синтаксичний аналізатор (або ж парсер) призначений для перетворення послідовності лексем у внутрішню форму, з якою працюють алгоритми аналізу коду та обчислення метрик.

### 2.3.1. Вхідні та вихідні дані

На вході системи очікуємо множину файлів з текстом програми мовою Lisp. Як правило, такі файли мають розширення .lsp або .lisp.

Спочатку відбувається зчитування тексту з кожного файлу окремо і послідовність символів передається синтаксичному аналізатору. Реалізацію лексичного аналізатора приховано у синтаксичному, завдяки можливостям бібліотеки Parsec (детальніше про неї в розділі 2.3.2.).

Оскільки програма мовою Lisp синтаксично є послідовністю символьних виразів, а символьні вирази є комбінацією списків та символів, найповніше інформацію про такий програмний код передає дерево розбору у формі вкладеного списку (рис. 2.4.). Символи та константи зберігаються у вигляді рядка і оброблюються за необхідності.

Отже на виході синтаксичного аналізатора одержуємо спеціально визначену структуру даних — дерево розбору (рис. 2.3.).

```
data Node = Sym String | Lisp [Node]
instance Show Node where
    show (Sym s) = s
    show (Lisp l) = show l
```

Рис. 2.3. Визначення типу дерева розбору

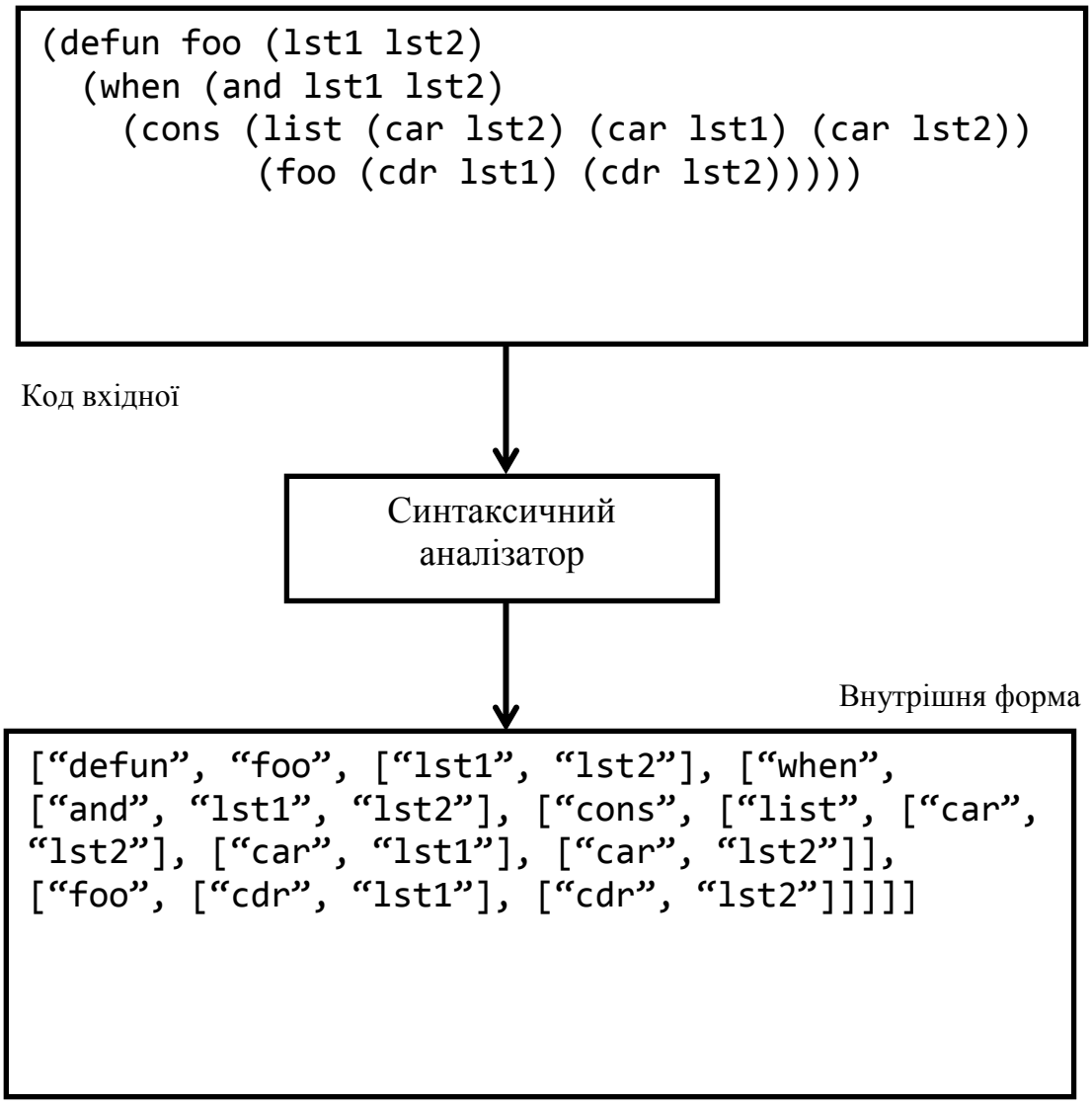


Рис. 2.4. Відповідність вхідного коду і внутрішньої форми

Дерево розбору (тип Node) — список, котрий може містити або рядки (конструктор Sym), або список гілок (конструктор Lisp). Таким чином внутрішня форма повторює структуру коду програми мовою Lisp, що й потрібно для його коректного аналізу.

### 2.3.2. Бібліотека Parsec

Parsec — бібліотека мови програмування Haskell, створена від початку як потужний інструментарій для розробки СА. Вона входить до стандартного набору бібліотек мови Haskell в рамках розділу обробки текстів.

Можливості бібліотеки дозволяють створювати СА для контекстно-залежних мов, з потенційно нескінченним переглядом символів наперед, але, звісно, найкращу продуктивність демонструє на LL(1) мовах.

Parsec реалізована як монадний (monadic) комбінатор СА. Замість того, щоб генерувати кінцевий автомат із формального опису певної граматики, як зазвичай відбувається в процесі розробки синтаксичних аналізаторів, кожен СА розглядається як комбінація інших. Базується все на елементарних СА, здатних оброблювати, наприклад, один конкретний символ.

Особливістю СА, розроблених за допомогою Parsec, є той факт, що вони є безпосередніми елементами мови Haskell, а не кодом, що генерується додатковими інструментами. Таким чином отриманий аналізатор не є просто «чорною скринькою», яка перетворює текст у внутрішню форму.

Оскільки головний СА (той, який на вхід отримує аксіому граматики) є комбінацією інших СА (тобто інших продукцій граматики), — це дозволяє за потреби використовувати їх окремо як СА підмножин мови. Це значно розширює можливості аналізу коду і спрощує розробку підсистем, які працюють із внутрішньою формою.

Недоліком цієї бібліотеки є певна незручність в процесі розробки, яка полягає в тому, що розробнику потрібно подати граматику мови не у звичному формальному вигляді (як от БНФ — форма Бакуса-Наура), а у вигляді комбінації СА за допомогою спеціальних операторів бібліотеки Parsec.

Для розробки синтаксичного аналізатора системи оцінювання якості програм мовою Lisp було обрано саме цю бібліотеку з двох причин. По-перше, граматика Lisp синтаксично — проста, і розробити її у формі комбінації СА було нескладно, тож головний недолік бібліотеки таким чином нівелювався. По-друге, як уже згадувалося, така форма СА значно спрощує програмування аналізу внутрішньої форми, що й було важливим критерієм у виборі інструментів розробки.

2.3.3. Реалізація

Як уже згадувалося, завдяки можливостям бібліотеки Parsec, лексичний аналізатор реалізований разом з синтаксичним аналізатором в одному модулі, тож розробник не мусить створювати окремий модуль для перетворення послідовності символів у послідовність лексем.

Завдання ж створення синтаксичного аналізатора зводиться до розробки правильної комбінації часткових СА граматики заданої мови.

Атомарними мовними одиницями в Lisp є:

- символи;
- цілі константи;
- дроби;
- десяткові дроби;
- числа в експоненціальній формі;

Отже саме ці СА і будуть базовими для розробленого синтаксичного аналізатора (рис. 2.5.).

```
symbol ::= alpha alphanum*
numeric-token ::= integer | ratio | float
integer  ::=   [sign]  digit+  decimal-point | [sign]  digit+
ratio ::=  [sign] {digit}+ slash  {digit}+
float ::= [sign] {digit}* decimal-point {digit}+ [exponent] |
          [sign] {digit}+ [decimal-point {digit}*] exponent
exponent ::=  exponent-marker [sign] {digit}+

sign :: = '-'
slash ::= '/'
decimal-point :: = '.'
exponent-marker ::= 'd' | 'e' | 'f' | 'l' | 's'
alphanum ::= alpha | digit
digit ::= '0' | '1' | .. | '9'
alpha ::= 'A' | 'B' | .. | 'Z'
```

Рис. 2.5. Опис базових СА

Згідно зі синтаксисом, програма мовою Lisp — це послідовність символьних виразів. Кожен символьний вираз, у свою чергу, є або списком символьних виразів, або символом, або константою (рис. 2.6.).

```
lisp-program ::= sexpr*
sexpr ::= sexprs | symbol | numeric-token
sexprs ::= ‘(’ sexpr* ‘)’
```

Рис. 2.6. Опис головного СА

Реалізована комбінація СА, розроблена засобами бібліотеки Parsec, виглядає так (рис. 2.7.):

```
lspProgram :: GenParser Char st [Node]
lspProgram = manyTill sexpr eof

sexpr  :: GenParser Char st Node
sexpr = do skippable
           result <- (sexprs <|> symbol)
           skippable
           return result

sexprs :: GenParser Char st Node
sexprs =
    do char '('
       result <- many sexpr
       char ')' >> skippable
       return $ Lisp result
```

Рис. 2.7. Реалізація головного СА

Частковий СА `skippable` відповідає за символи, які можна пропустити. В Lisp це пробіл, символи табуляції та переходу на новий рядок. Однак через відсутність безпосереднього доступу до лексичного аналізатора, виділення коментарів довелося додати до реалізації СА як окремий частковий СА `skippable` (рис. 2.8.). В Lisp два типи коментарів — рядковий і блоковий. Рядковий — послідовність, що починається зі символу ‘;’ (крапка з комою), — всі символи до переходу на новий рядок вважаються коментарем. Блоковий коментар — текст обмежений послідовностями символів ‘#|’ та ‘|#’.

```
skippable :: GenParser Char st ()
skippable = skipMany ((space >> return ()) <|> comment)

comment :: GenParser Char st ()
comment = commentLine <|> commentBlock

commentLine :: GenParser Char st ()
commentLine =
    do char ';'
       manyTill anyChar (char '\n')
       return ()

commentBlock :: GenParser Char st ()
commentBlock =
    do (try (string "#|"))
       manyTill anyChar (try (string "|#"))
       return ()
```

Рис. 2.8. Реалізація СА `skippable`

Розробленими СА можна користуватися шляхом виклику спеціальної функції `parse` (рис. 2.9.). Аргументи функції: конкретний СА та вхідна послідовність символів. Повертає вона результат типу, що вказаний в СА як результат обробки вхідного рядка, або об'єкт спеціального типу `ParseError`, в разі виникнення помилки під час його обробки.

```
parse lispProgram “error” inputString
```

Рис. 2.9. Виклик функції `parse`

Для зручності розроблено дві додаткові функції `parseLsp`, яка аналізує вхідний рядок як цілу програму мовою Lisp, та `parse'`, яка аналізує вхідний рядок як один символьний вираз (рис. 2.10.).

```
parseLsp :: String -> Either ParseError [Node]
parseLsp input = parse lspFile "(unknown)" input

parse' :: String -> Node
parse' input =
    let node = parse lisp "(unknown)" input in
        either (\_ -> Sym "") (\x -> x) node
```

Рис. 2.10. Реалізація функцій `parseLsp` та `parse'`

## 2.4. Розробка модуля обчислення метрик

### 2.4.1. Допоміжний модуль

Для прискорення розробки функцій обчислення метрик спершу було розроблено допоміжний модуль, що містить функції для роботи із внутрішньою формою представлення вхідної програми.

Далі терміном «гілка» називаємо об’єкт визначеного типу `Node`, що є основним елементом дерева розбору.

В процесі розробки було визначено такий набір інтерфейсних функцій допоміжного модуля:

- `nodeEq` — функція строгої рівності двох гілок дерева розбору;
- `nodeEq'` — функція нестрогої рівності двох гілок дерева розбору;
- `forthis` — «для цієї гілки» — повертає задане значення, якщо вхідна гілка задовольняє описані правила;
- `foreachCount` — рахує кількість гілок вхідного дерева, які задовольняють описані правила;
- `foreachCollect` — збирає всі гілки вхідного дерева, які задовольняють описані правила.

Тепер детальніше про кожну з них.

`nodeEq` — приймає дві гілки як параметри і визначає, чи вони строго рівні. Гілки вважаються строго рівними тоді, коли всі їхні елементи ідентичні.

`nodeEq'` — приймає дві гілки як параметри і визначає, чи вони нестрого рівні. В загальному випадку, працює як `nodeEq`, однак передбачає, що будь-яка гілка може містити на місці будь-якого символьного виразу один із двох спеціальних елементів: «будь-який символьний вираз» та «будь-яка кількість будь-яких символьних виразів» (рис. 2.11.).

`forthis` — приймає гілку та набір правил у вигляді списку пар, кожна з яких містить рядок і значення. Рядок — код мовою Lisp, який аналізується СА під час виконання функції і порівнюється зі вхідною гілкою (першим аргументом). Якщо гілки рівні, — повертає друге значення відповідної пари.

```
> nodeEq (parse' "(defun foo (x) (* x x))")
         (parse' "(defun foo (x) (* x x))")
> True
> nodeEq (parse' "(defun foo (x) (* x x))")
         (parse' "(defun foo (x) (* x 2))")
> False
> nodeEq' (parse' "(defun foo (x) (* x x))")
          (parse' "(defun _ _ _)")
> True
> nodeEq' (parse' "(defun foo (x) (* x x))")
          (parse' "(defun _*)")
> True
> nodeEq' (parse' "(defun foo (x) (* x x))")
          (parse' "(defvar _*)")
> False
```

Рис. 2.11. Робота функцій `nodeEq` та `nodeEq'`

`foreachCount` — приймає гілку та набір правил у вигляді списку рядків. Гілка та всі її підгілки аналізуються за допомогою `forthis`. Результатом функції є кількість підгілок, що задовольняють хоча б одне із заданих правил.

`foreachCollect` — працює аналогічно до `foreachCount`, тільки в результаті повертає не кількість, а список гілок, що задовольняють хоча б одне із правил.

Роботу функцій `forthis`, `foreachCount` та `foreachCollect` показано на рис. 2.12.

```
> forthis (parse' "(defun foo (x) (* x x))")
          [("(defun _*)", 1),
           ("(defvar _*)", 2),
           ("(defparameter _)", 3)]
> 1

> foreachCount (parse' "(defun foo (x) (* x x))")
               ["x"]
> 3
> foreachCount (parse' "(defun foo (x) (* x x))")
               ["(* _ _)"]
> 1

> foreachCollect (parse' "(defun foo (x) (* 2 (* x x)))")
                 ["(defun _*)",
                  "(* _ _)"]
> ["(defun foo (x) (* 2 (* x x)))", "(* 2 (* x x))", "(* x x)"]

> foreachCollect (parse' "(defun foo (x) (* 2 (* x x)))")
                 ["(defvar _*)", "(defparameter _ _)"]
> []
```

Рис. 2.12. Робота функцій `forthis`, `foreachCount` та `foreachCollect`

### 2.4.2. Метрики розміру

Найпоширенішою характеристикою розмірів програми серед програмістів залишається LOC (lines of code) — кількість рядків коду. Очевидно, що цей показник,

хоч і дає відносне уявлення про розміри системи, не є об’єктивним. Кількість рядків коду змінюється залежно від мови програмування, стандартів стилю, які використовуються, врешті-решт від конкретного програміста.

Тому було вирішено крім цієї характеристики врахувати низку додаткових, специфічних для мови Lisp. Серед них:

- кількість символьних виразів;
- кількість функцій;
- середній розмір функції в LOC;
- середня кількість символьних виразів на функцію;
- відношення кількості символьних виразів до LOC;

Ці показники дають можливість оцінити та порівняти розміри різних програм мовою Lisp, а також функцій в межах однієї програми.

Проте середні значення показують лише узагальнену характеристику розмірів. З метою одержання конкретної інформації про громіздкі частини системи використовуються статистичні квантилі.

Квантиль — одна з числових характеристик випадкових величин, що застосовується в математичній статистиці. Квантилі відсікають в межах ряду певну частину його членів. Тобто, квантиль розподілення значень — це таке число $x_p$, що значення *p*-ї частини сукупності менше або рівне $x_p$. Наприклад, квантиль 0.25 змінної — це таке значення $x_p$, що 25% (*p*) значень змінної є меншими за нього.

Випадковою величною, коли йдеться про метрики розміру, є, власне, розмір якогось елементу програми, наприклад, функції чи файлу. Спочатку збирається і зберігається в масив інформація про розмір кожної функції. Потім цей масив сортується за зростанням. Далі вибираються певні квантилі, які цікавлять оцінювача. Як правило, обираються Q25, Q50, Q75 та Q95, однак це легко конфігурується.

Маючи ці дані легко зрозуміти наскільки громіздкими є елементи системи. Наприклад, якщо розмір функції у квантилі 75 становить 22 рядки, це означає, що три чверті всіх функцій за розмірами менші або рівні за дану. А це свідчить про велику

кількість малих функцій, що в свою чергу дозволяє припускати, що програму легко підтримувати.

Таким чином в системі оцінювання якості програм мовою Lisp було реалізовано обчислення таких метрик розміру:

- `countOfSexprs` — функція, що обчислює кількість символьних виразів у дереві розбору, що передається параметром;
- `countOfFunctions` — функція, що обчислює кількість функцій, визначених за допомогою `defun` та `defmethod`;
- `averageLOC` — функція обчислення середнього розміру функції в LOC;
- `averageSexprs` — функція обчислення середньої кількості символьних виразів на функцію;
- `sexprPerLOC` — відношення кількості символьних виразів до LOC;
- `functionSizes` — функція, що збирає масив розмірів функцій;
- `fileSizes` — функція, що збирає масив розмірів файлів, як аргумент приймає хеш-таблицю з деревами розбору файлів;
- `quantile` — функція, що приймає масив та число у проміжку [0..1], і повертає відповідний квантиль;

2.4.3. Метрики Галстеда

Метрики Галстеда запропонував Моріс Говард Галстед у 1977 році як частину свого трактату про створення емпіричної науки в галузі розробки ПЗ [4]. Галстед зауважив, що метрики ПЗ мають відображати реалізацію алгоритмів різними мовами програмування, але бути незалежними від того, на якій платформі ці алгоритми виконуються. А отже ці метрики мають обчислюватися статистично з коду. Метою Галстеда було виявити вимірні властивості ПЗ та зв'язки між ними.

Базовими параметрами в метриках Галстеда є чотири показники:

- $n_1$ — кількість різних операторів;
- $n_2$ — кількість різних операндів;
- $N_1$ — кількість усіх операторів;
- $N_2$ — кількість усіх операндів.

Під оператором в метриках Галстеда розуміють послідовність символів, яка описує дію, що виконується, а операндом — те, над чим ця дія виконується.

В загальному випадку, найскладнішим в алгоритмі підрахунку цих параметрів (рис 2.13.) є розрізнення операторів та операндів. Адже, наприклад, виклики функцій з точки зору метрик Галстеда є операторами, а це означає, що на етапі підрахунку вже має бути відомо, які ідентифікатори є іменами функцій, які — іменами змінних тощо.

З базових параметрів обчислюються такі метрики:

- $n = n_1 + n_2$ — словник програми;
- $N = N_1 + N_2$ — довжина програми;
- $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$ — розрахункова довжина програми ;
- $V = N \times \log_2 n$ — об'єм програми;
- $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$ — складність програми;
- $E = D \times V$ — робота, потрібна для розробки;

Обчислена складність характеризує складність написання та розуміння програми. З обчисленого значення роботи за формулою (1) можна оцінити реальний час, потрібний на розробку.

$$T = \frac{E}{18} \quad (1)$$

Ще однією метрикою Галстеда є кількість потенційних помилок у реалізації програми. Класична формула (2) обчислення цієї метрики — нижче.

$$B = \frac{E^{\frac{2}{3}}}{3000} \quad (2)$$

Модифікована формула (3).

$$B = \frac{V}{3000} \quad (3)$$

Оскільки модифікована формула не враховує показник складності, в системі було вирішено використовувати класичну (2).

В процедурних мовах програмування операторами, як правило, вважаються арифметичні та логічні операції, умовні конструкції, спеціальні мовні конструкції, ключові слова, виклики функцій тощо. Операндами — ідентифікатори змінних, параметри функцій, константи.

Розглянемо приклад простої програми мовою С (рис 2.14.).

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

Рис. 2.14. Приклад програми мовою С

Тут унікальними операторами є: `main, (), {}, int, scanf, &, =, +, /, printf.`

Унікальні операнди: `a, b, c, avg, "%d %d %d", 3, "avg = %d".`

Круглі дужки вважаються за оператор, тому що вони показують порядок виконання арифметичних операцій, тобто виконують якусь дію. Фігурні дужки оголошують про початок нового смислового блоку (в даному випадку — тіла функції).

Дужки у випадку Lisp не варто вважати за окремий оператор, оскільки вони мають виключно синтаксичний сенс, — показують початок списку.

В мові програмування Lisp відсутні арифметичні та логічні оператори, натомість в ній використовуються відповідні наперед визначені функції. Виклик функції в Lisp — це символьний вираз, який є списком, перший елемент якого — символ-ідентифікатор функції, а наступні — конкретні параметри виклику. Наприклад:

```
(cons 1 (list 2 3))
(+ 4 5 6)
(append *list* '(a b c))
(if (< a 0) (* a -1) (/ a 2))
```

Таким чином операторами з точки зору метрик Галстеда в Lisp будуть усі виклики функцій — тобто усі перші елементи списку, які є символом (а не списком) . Операндами — всі параметри всіх викликів.

Таблиця 2.1.

Оператори та операнди метрик Галстеда в Lisp

| Символьний вираз | Унікальні оператори | Унікальні операнди |
|---|---|---|
| `(+ 4 5 6)` | `+` | `4, 5, 6` |
| `(append *list* '(a b c))` | `append` | `*list*, '(a b c)` |
| `(cons 1 (list 2 3))` | `cons, list` | `1, (list 2 3), 2, 3` |
| `(if (< a 0) (* a -1) (/ a 2))` | `if, <, *, /` | `(< a 0), a, 0, (* a -1), -1, (/ a 2), 2` |

Отже для того, щоб отримати базові параметри метрик Галстеда з коду програми мовою Lisp, потрібно зібрати усі виклики функцій і визначити кількість викликів кожної функції окремо та кількість разів використання кожного параметра окремо.

Функція `halsteadOperators` (рис. 2.15.) отримує на вхід дерево розбору і повертає пару цілих чисел — кількість усіх операторів та кількість унікальних операторів ($N_1$ та $n_1$ відповідно).

```
halsteadOperators :: [Node] -> (Int, Int)
halsteadOperators term =
  let allLists = foreachCollect term ["(_*)"]
      allOperators = filter nodeSymbol $ map (\n -> head
$ nodeComponents n) allLists
      uniqueOperators = removeDuplicates allOperators
  in (length allOperators, length uniqueOperators)
```

Рис. 2.15. Реалізація функції `halsteadOperators`

Функція `halsteadOperands` (рис. 2.16.) отримує на вхід дерево розбору і так само повертає пару цілих чисел — кількість усіх операндів та кількість унікальних операндів ($N_2$ та $n_2$ відповідно).

```
halsteadOperands :: [Node] -> (Int, Int)
halsteadOperands term =
  let allLists = foreachCollect term ["(_*)"]
      allOperands = foldl (++) [] $ map (\n -> tail $
nodeComponents n) allLists
      uniqueOperands = removeDuplicates allOperands
  in (length allOperands, length uniqueOperands)
```

Рис. 2.16. Реалізація функції `halsteadOperands`

Нарешті функція `halsteadMetrics`, яка використовує описані вище функції, приймає на вхід дерево розбору, обчислює всі метрики Галстеда і повертає хеш-таблицю, в якій ключем є рядок із позначенням метрики (наприклад, “n1” для кількості унікальних операторів, чи “D” для складності), а значенням — відповідне число типу `Double`.

### 2.4.4. Граф викликів

За визначенням граф викликів — це орієнтований граф, який показує зв’язки між частинами програми через виклики ними одна одну. Кожна вершина представляє процедуру або функцію, а кожна дуга (f, g) — виклик процедурою f процедури g. Цикли в таких графах вказують на наявність рекурсивних викликів.

Граф викликів — базовий результат аналізу програми, здатний візуально показати її роботу. Також граф викликів часто є основою для інших аналізів, таких як, наприклад, пошук коду, що не використовується.

Розрізняють статичні та динамічні графи викликів.

Динамічний граф викликів — запис одного виконання програми, іншими словами результат роботи профайлера. Незважаючи на те, що динамічний граф викликів — точний, він описує лише одне конкретне виконання програми з конкретними вхідними даними.

Статичний граф викликів показує усі можливі варіанти виконання програми. Визначення точного статичного графу викликів — алгоритмічно нерозв'язна задача, відтак алгоритми побудови статичних графів викликів зазвичай дають дуже узагальнений результат. Тобто будь-який виклик (дуга) у такому графі може бути таким, що ніколи не виконається в реальних запусках програми.

Графи викликів можуть мати різну точність. Чим точніше граф передає поведінку програми, тим більше ресурсів потрібно для його побудови. Найточнішими є контекстно-чутливі (context-sensitive) графи викликів. Такі графи для кожної процедури створюють окремі вершини для кожного стеку викликів, в який вона потрапляє. Повний контекстно-чутливий граф називається «деревом контекстних викликів». Найменш точними є контекстно-нечутливі графи викликів. Такі графи для кожної процедури створюють лише одну вершину.

В розробленій системі було реалізовано побудову контекстно-нечутливого графу викликів — функцію `callGraph`.

Алгоритм побудови цього графу зображено на рис. 2.17.

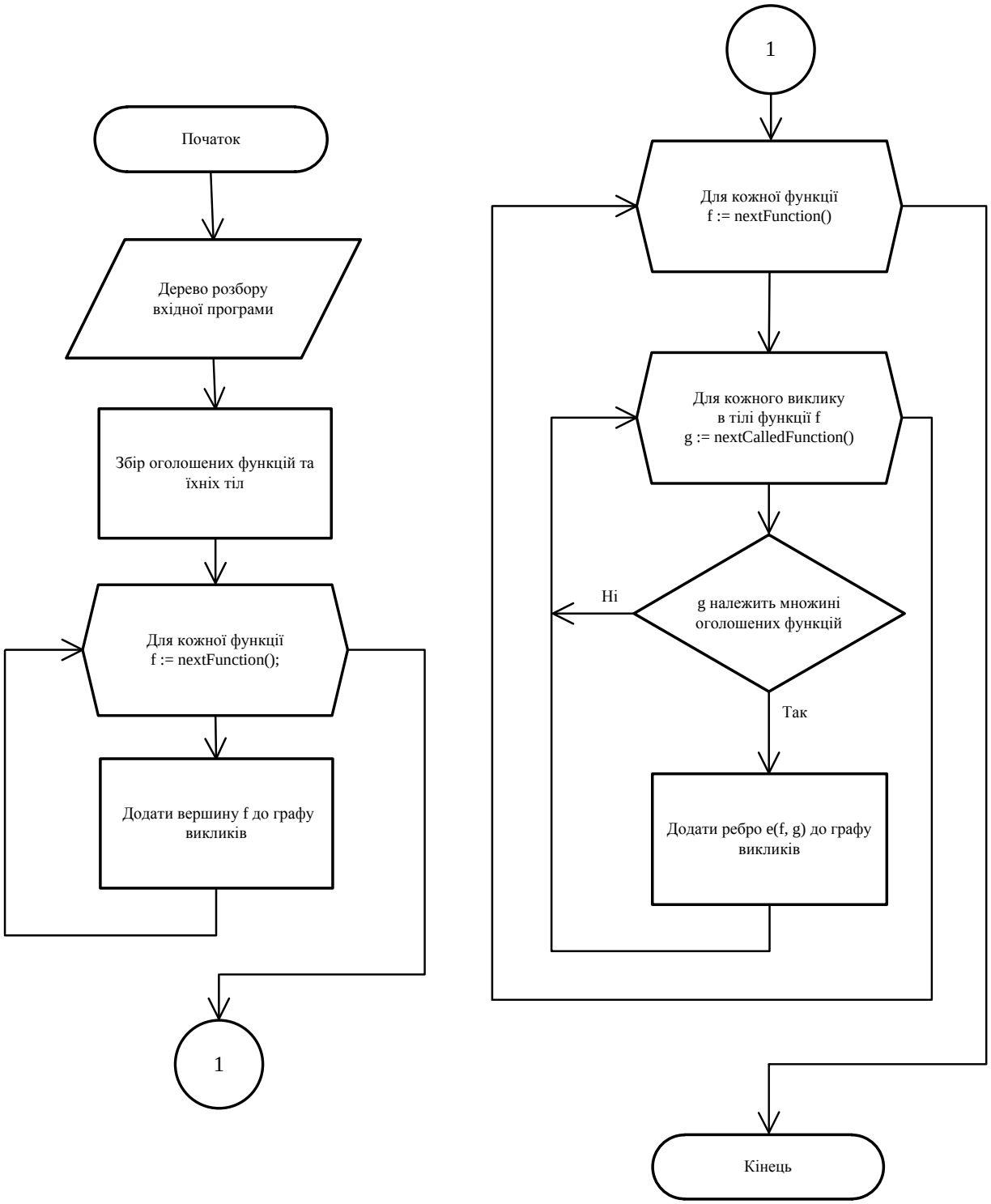


Рис. 2.17. Блок-схема алгоритму побудови графу викликів

## 2.4.5. Матриця залежності «використання-визначення»

Матриці залежності (також матриці структури — DSM) показують взаємозв’язок частин або елементів програми між собою. Існує декілька видів таких матриць: «використання-визначення», матриця залежності ресурсів, матриця залежності файлів та інші (див. розділ 1.3.2.).

Елементами матриці залежності «використання-визначення» (usage-definition DSM) є глобальні змінні та функції. На перетині стовпчика та рядка — ціле число, яке показує кількість зв’язків між елементами, що іменують стовпчик та рядок. Фактично це число можна розуміти як кількість разів використання елементами одне одного.

Для прикладу розглянемо невелику програму (рис. 2.18.).

```
(defparameter *list* '(1 2 3 4))

(defun foo (n)
  (mapcar (lambda (x) (+ x n)) *list*)))

(defun bar (n)
  (if *list*
      (append *list* (foo n))
      '(9 8 7 4)))
```

Рис. 2.18. Приклад програми мовою Lisp

В цій програмі одна глобальна змінна `*list*` та дві функції. Глобальна змінна використовується в обох функціях. Крім того функція `bar` викликає функцію `foo`. Матриця залежності «використання-визначення» для цієї програми виглядатиме так (табл. 2.2.):

Таблиця 2.2.

Матриця залежності «використання-визначення»

| | *list* | foo | bar |
|---|---|---|---|
| *list* | 0 | 1 | 2 |
| foo | 1 | 0 | 1 |
| bar | 2 | 1 | 0 |

На перетині імен `*list*` і `bar` значення 2, оскільки змінна `*list*` використовується в тілі функції `bar` двічі.

Реалізація побудови такої матриці значно спрощується, якщо є побудований граф викликів. Очевидно, що матрична форма такого графу і буде матрицею залежностей «використання-визначення» для функцій. Однак реалізована в системі побудова графу викликів дозволяє лише визначити факт наявності виклику однією функції іншою. Таким чином одержана матрична форма графу буде лише бінарною версією потрібної матриці (є залежність, або немає).

Крім того під час побудови матриці «використання-визначення» необхідно враховувати не лише виклики функцій, а й параметри цих викликів. Також матриця має показувати зв’язки між змінними та функціями. Тому в системі оцінювання якості програм мовою Lisp було реалізовано окремий алгоритм для побудови матриці залежності «використання-визначення» (рис. 2.19.).

Подібно до алгоритму побудови графу викликів, спочатку виконується збір необхідних елементів програми — функцій та глобальних змінних. Потім відбувається ініціалізація матриці як нульової. Далі обробляється визначення кожного елемента (тіло для функцій, вираз ініціалізації для змінної). Аналізуються всі символьні вирази й кінцеві символи перевіряються на присутність в множині оголошених. До відповідної комірки в матриці додається одиниця, якщо виявлено зв’язок між якимись елементами.

Алгоритм реалізовано в функції `useDefDSM`, яка на вхід приймає дерево розбору програми.

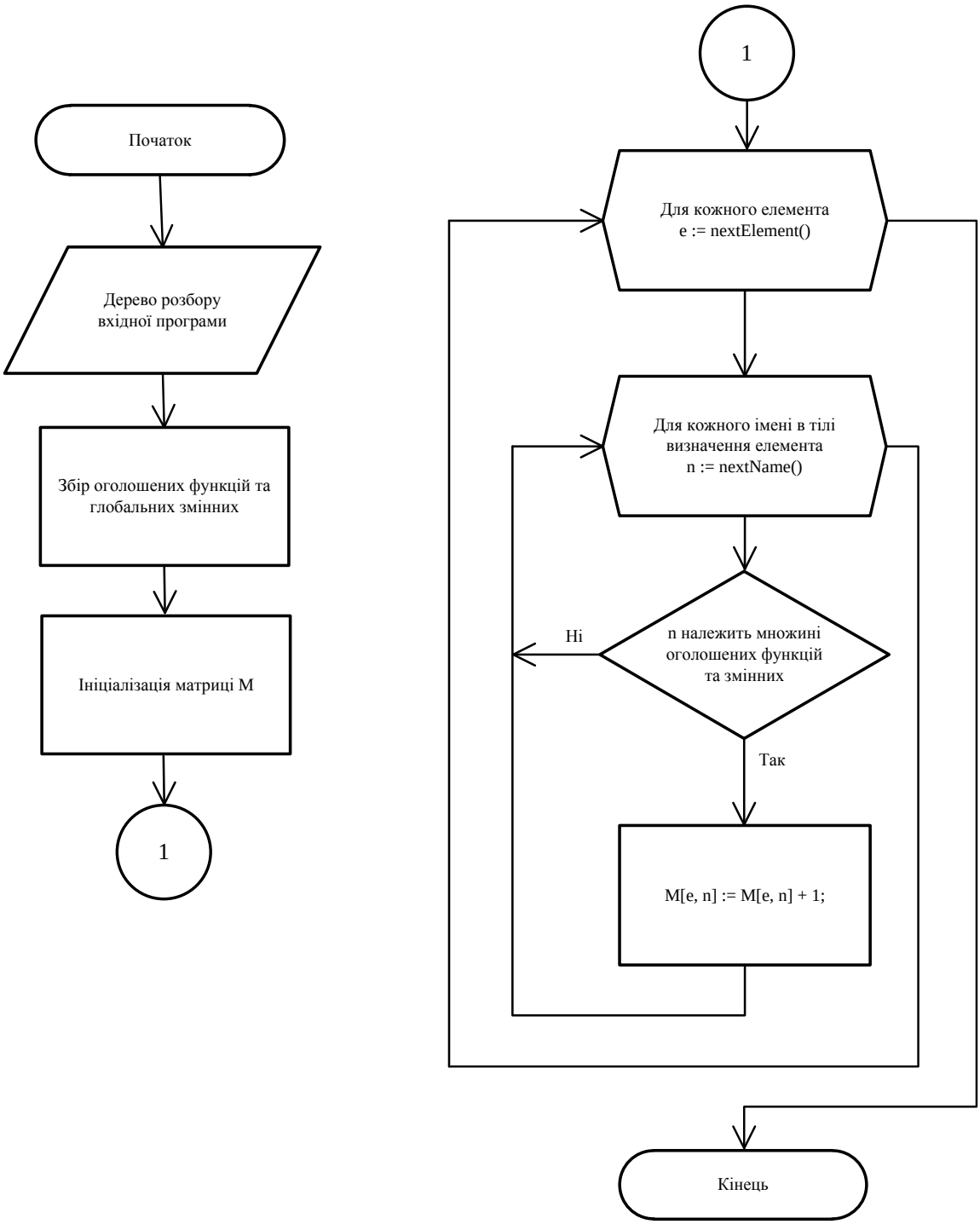


Рис. 2.19. Блок-схема алгоритму побудови
матриці залежності «використання-визначення»

2.4.6. Цикломатична та рекурсивна складність

Цикломатична складність — метрика ПЗ, що має на меті визначення складності програми. Кількісна міра числа незалежних шляхів через програмний код. Розроблена Томасом Дж. МакКейбом у 1976 році [6].

Як правило, цикломатичну складність обчислюють за допомогою побудови графа потоку команд. Вершини цього графа представляють неподільні групи команд програми, а дуги зв’язують дві вершини якщо наступна команда може бути виконана негайно після завершення попередньої.

Значення цикломатичної складності для даного графа потоку команд програми визначається за формулою (4):

$$M = E - N + 2P \tag{4}$$

де:

- $E$ — кількість дуг графа;
- $N$ — кількість вершин графа;
- $P$ — кількість компонент зв’язності графа;

Визначення цикломатичної складності за допомогою графа потоку команд є точним. Однак процес побудови такого графа, особливо коли йдеться про великі програми, вимагає значних обчислювальних ресурсів.

Існує простіший метод визначення цикломатичної складності. Оскільки цикломатична складність фактично є кількістю незалежних потоків команд у програмі, вона відповідає кількості блоків рішень, тобто блоків які впливають на потік команд. Перш за все йдеться про умовні конструкції та цикли.

Для мови програмування Lisp було визначено такі блоки (а точніше — функції) рішень: `if`, `when`, `loop`, `assert`, `map`, `mapc`, `mapcar`, `reduce`, `lambda`, `and`, `or` та інші стандартні функції.

Цей метод значно спрощує реалізацію обчислення цикломатичної складності, оскільки зводиться до перебору символьних виразів і пошуку серед них функцій із заданої множини.

Обчислення цієї метрики реалізовано в функції `cyclomaticComplexity`.

В мові програмування Lisp широко застосовується стиль функціонального програмування. Таким чином для визначення складності програмного коду недостатньо самої лише цикломатичної складності.

Тому в системі оцінювання якості програм мовою Lisp пропонується додатковий показник — рекурсивна складність.

Рекурсивна складність має на меті охарактеризувати кількість та інтенсивність рекурсивних викликів у програмі одним числом.

Очевидно, що виявити рекурсивні виклики найпростіше в графі викликів. Кожен цикл у такому графі показуватиме рекурсивний виклик (необов'язково безпосередній). Отже для графу викликів без циклів рекурсивна складність має дорівнювати нулю. З іншого боку, найгіршим випадком (найбільше значення рекурсивної складності) буде такий, що кожна функція викликає кожну і саму себе.

Найкраще цим вимогам відповідає параметр, відомий в теорії графів як цикломатичне число. Математичне визначення: цикломатичне число — мінімальна кількість ребер неорієнтованого графа, яку потрібно видалити, щоб отримати граф без циклів — ліс. Таким чином рекурсивна складність визначається за формулою (5):

$$M = E - N + P + L \qquad (5)$$

Де $L$ — кількість петель у графі викликів.

Як бачимо, вона майже ідентична до класичної формули визначення цикломатичної складності. Різниця полягає в тому, що цикломатична складність — це цикломатичне число графа потоку команд програми, а рекурсивна складність — цикломатичне число графа викликів.

У графі потоку команд цикли представляють власне циклічні конструкції мови програмування та переходи. У графі викликів цикли представляють виключно рекурсивні послідовності викликів.

Зрозуміло, якщо граф викликів — контекстно-чутливий, то й значення рекурсивної складності точніше відображатиме складність програми.

Для системи оцінювання якості програм мовою Lisp було реалізовано обчислення рекурсивної складності на побудованому контекстно-нечутливому графі викликів. Алгоритм обчислення рекурсивної складності зображено на рис. 2.20.

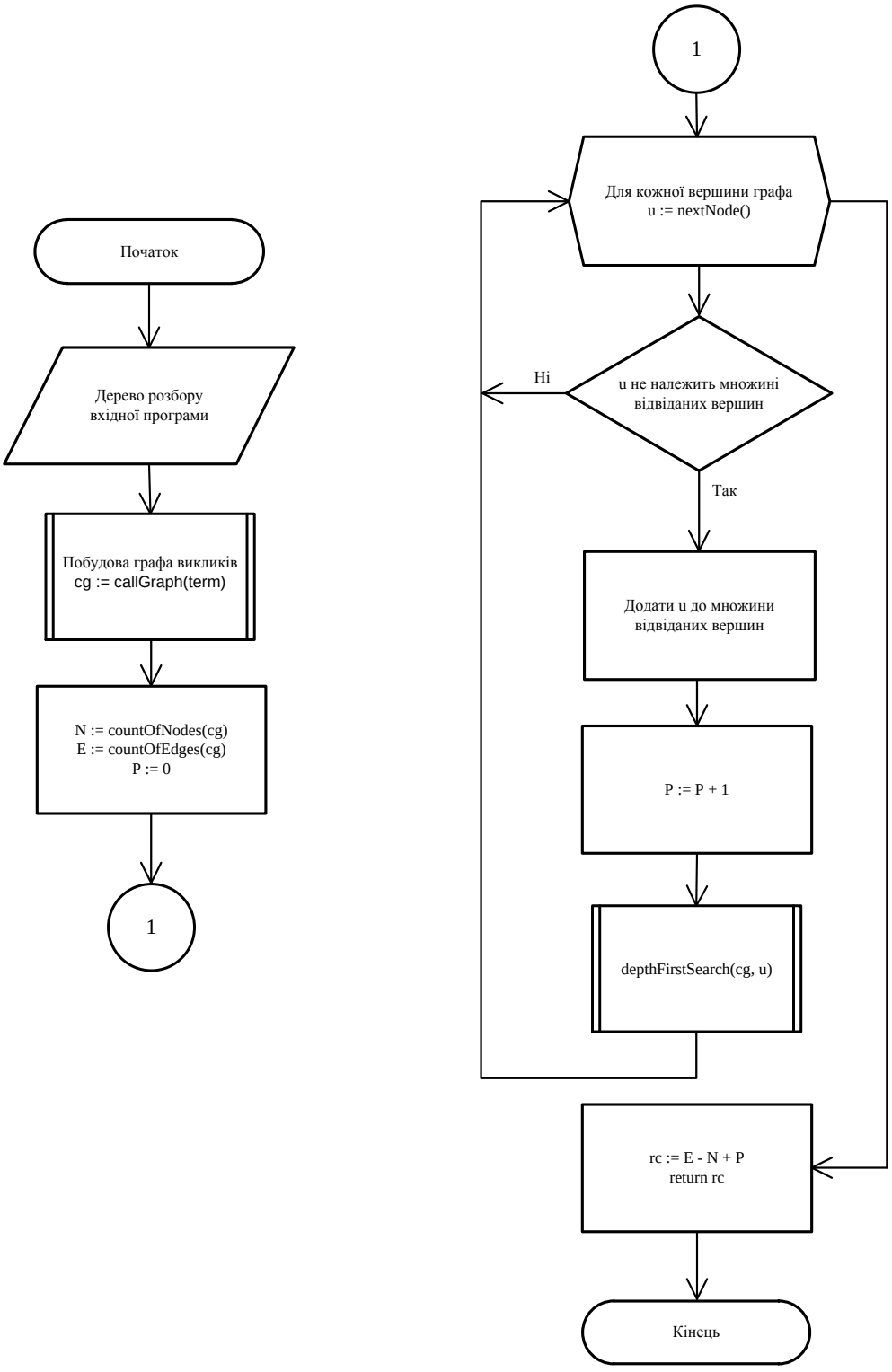


Рис. 2.20. Блок-схема алгоритму обчислення рекурсивної складності

### 2.4.7. Індекс підтримуваності

Індекс підтримуваності (maintainability index) — метрика ПЗ, яка характеризує наскільки складно підтримувати даний програмний код. Індекс підтримуваності є числом в межах [0..100] і обчислюється з параметрів, визначених за допомогою метрик Галстеда та інших числових характеристик коду.

Існує декілька формул обчислення індексу підтримуваності, які використовуються в різних системах статичного аналізу програмного коду.

Класична формула (4) враховує об’єм коду $V$ (один із параметрів метрик Галстеда), цикломатичну складність $G$ та розмір коду $L$ в LOC.

$$MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L \quad (4)$$

Інститут Програмної Інженерії (Software Engineering Institute — SEI) пропонує модифіковану формулу (5), яка враховує наявність коментарів.

Тут $C$ — кількість коментарів у LOC.

$$MI = 171 - 5.2 \log_2 V - 0.23G - 16.2 \log_2 L + 50 \sin(\sqrt{2.4C}) \quad (5)$$

Microsoft Visual Studio використовує іншу модифіковану формулу (6).

$$MI = \max\left[0, 100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L}{171}\right] \quad (6)$$

Деякі системи оцінювання якості програм (наприклад, Radon для мови python) використовують комбіновану формулу (7).

$$MI = \max\left[0, 100 \frac{171 - 5.2 \log_2 V - 0.23G - 16.2 \log_2 L + 50 \sin(\sqrt{2.4C})}{171}\right] \quad (7)$$

У системі оцінювання якості програм мовою Lisp було вирішено для обчислення індексу підтримуваності використовувати комбіновану формулу (7), оскільки вона враховує коментарі та побудована таким чином, що число в результаті завжди перебуватиме в проміжку [0..100].

Варто зауважити, що згідно з кожною формулою індекс підтримуваності сильно залежить від розмірів коду. Таким чином, можна констатувати, що для будь-якої мови програмування існує такий розмір програми, для якого індекс підтримуваності буде дорівнювати нулю, незалежно від інших показників. Це означає, що індекс підтримуваності є сенс застосовувати лише для окремих модулів, класів, функцій чи інших структурних одиниць мови програмування, але не для цілих систем.

Таким чином було вирішено обчислювати індекс підтримуваності для окремих гілок, а не цілого дерева розбору. Окремо для кожної функції та окремо для кожного файлу.

Отже розроблена функція `maintainabilityIndex` приймає параметр типу `Node` і повертає значення індексу підтримуваності. Для обчислення використовуються функції

`halsteadMetrics` та `cyclomaticComplexity`, опис яких наведено в розділах 2.4.3 та 2.4.6 відповідно.

# 3. ТЕСТУВАННЯ КОМПОНЕНТІВ СИСТЕМИ

## 3.1. Тестування синтаксичного аналізатора та допоміжного модуля

Для тестування СА було обранj функції `parseLsp` та `parse'`.

Опис та результати тестування подано в таблиці 3.1.

Таблиця 3.1.

Опис та результати тестування роботи СА

| № | Назва | Опис | Вхідні дані | Очікуваний результат | Отриманий результат | Результат тесту |
|---|---|---|---|---|---|---|
| P1 | parseLsp. Коректні дані | Коректні вхідні дані | “(defvar x nil) (defun foo () x)” | [[“defvar”, “x”, “nil”], [“defun”, “foo”, [], “x”]] | [[“defvar”, “x”, “nil”], [“defun”, “foo”, [], “x”]] | + |
| P2 | parseLsp. Коректні дані. Коментарі | Вхідний рядок містить коментарі | “; line comment (defvar x nil) (defun foo () #\| block comment \|# x)” | [[“defvar”, “x”, “nil”], [“defun”, “foo”, [], “x”]] | [[“defvar”, “x”, “nil”], [“defun”, “foo”, [], “x”]] | + |
| P3 | parseLsp. Синтаксична помилка | Вхідний рядок містить синтаксичну помилку | “(defvar x nil) (defun foo () x” | ParseError | ParseError | + |
| P4 | parse'. Коректні дані | Коректні вхідні дані | “(+ a 1)” | [“+”, “a”, “1”] | [“+”, “a”, “1”] | + |
| P5 | parse'. Коректні дані. Коментарі | Вхідний рядок містить коментарі | “; line comment (+ a #\| block comment \|# 1)” | [“+”, “a”, “1”] | [“+”, “a”, “1”] | + |
| P6 | parse'. Синтаксична помилка | Вхідний рядок містить синтаксичну помилку | “+ a 1)” | ParseError | ParseError | + |

Базові функції допоміжного модуля: `nodeEq'` та `foreachCount`

Опис та результати тестування подано в таблиці 3.2.

Таблиця 3.2.

Опис та результати тестування роботи функцій допоміжного модуля

| № | Назва | Опис | Вхідні дані | Очікуваний результат | Отриманий результат | Результат тесту |
|---|---|---|---|---|---|---|
| U1 | nodeEq' Ідентичні гілки | Вхідні дані — ідентичні гілки | Sym “foo” Sym “foo” | True | True | + |
| U2 | nodeEq' Будь-який символьний вираз | Одна зі вхідних гілок означає будь-який символьний вираз | Sym “_” Lisp [“defvar”, “x”] | True | True | + |
| U3 | nodeEq' Різні гілки | Вхідні дані — різні гілки | Sym “foo” Sym “bar” | False | False | + |
| U4 | foreachCount. Усі вирази | Вхідне правило — будь-який вираз | “(+ a 1)” [“_”] | 4 | 4 | + |
| U5 | foreachCount. Списки | Вхідне правило — будь-який список | “(list (+ a 1) ‘(1 2 3))” [“(_*)”] | 3 | 3 | + |
| U6 | foreachCount. Конкретний символ | Вхідне правило — конкретний символ | “(if (eq lst nil) nil (cons nil lst))” [“nil”] | 3 | 3 | + |
| U7 | foreachCount. Без правил | Список правил — порожній | “(if (eq lst nil) nil (cons nil lst))” [] | 0 | 0 | + |

3.2. Тестування обчислення метрик

Для тестування метрик було взято невелику програму мовою Lisp. Програма реалізовує функції пошуку найкоротшого шляху в орграфі за допомогою алгоритму пошуку в глибину. Вихідний код програми подано нижче (рис. 3.1.).

```
(defparameter *visited* nil)
(defparameter *solutions* nil)

(defun visited? (v) (find v *visited*))

(defun df (g v &optional path)
    (push v *visited*)
    (mapc (lambda (e)
              (if (eq (cadr e) y)
                (push (reverse (cons e path)) *solutions*)
                (unless (visited? (cadr e))
                        (df (cadr e) (cons e path)))))
          (remove-if-not (lambda (e) (eq (car e) v)) g)))

(defun path-length (p)
    (reduce #'+ (mapcar #'third p)))

(defun shortest-path-df (g x y)
  (let (visited
        solutions)
      (df g x)
      (let ((solution (car *solutions*)))
        (mapc (lambda (s)
                (if (< (path-length s)
                       (path-length solution))
                    (setf solution s)))
```

Рис. 3.1. Код вхідної програми для тестування обчислення метрик

Результати обчислення розмірних метрик для тестової програми подано в таблиці 3.3.

Таблиця 3.3.

Результати обчислення розмірних метрик для тестової програми

| Назва метрики | Назва функції | Результат |
|---|---|---|
| Розмір програми в LOC | `countOfLines` | 26 |
| Загальна кількість символьних виразів | `countOfSexprs` | 141 |
| Кількість функцій | `countOfFunctions` | 4 |
| Середній розмір функції в LOC | `averageLOC` | 5 |
| Середня кількість виразів на функцію | `countOfSexprs` | 33.25 |
| Відношення кількості символьних виразів до LOC | `sexprPerLOC` | 5.4 |
| Розміри функцій в кількості виразів | `functionSizes` | 9, 64, 12, 48 |
| Квантиль Q50 розмірів функцій | `Quantile 0.5` | 48 |

Результати обчислення метрик Галстеда для тестової програми подано в таблиці 3.4.

Таблиця 3.4.

Результати обчислення метрик Галстеда для тестової програми

| Назва метрики | Умовне позначення | Значення |
|---|---|---|
| Кількість різних операторів | n1 | 30 |
| Кількість різних операндів | n2 | 59 |
| Кількість усіх операторів | N1 | 49 |
| Кількість усіх операндів | N2 | 85 |
| Словник програми | n | 89 |
| Довжина програми | N | 134 |
| Розрахункова довжина програми | N’ | 494.3 |
| Об’єм програми | V | 867.7 |
| Складність програми | D | 21.6 |
| Робота, потрібна для розробки | E | 18752 |

| | | |
|---|---|---|
| Час розробки | T | 1042 с |
| Кількість потенційних помилок | B | 0.24 |

Граф викликів у графічному вигляді зображено на рис. 3.2.

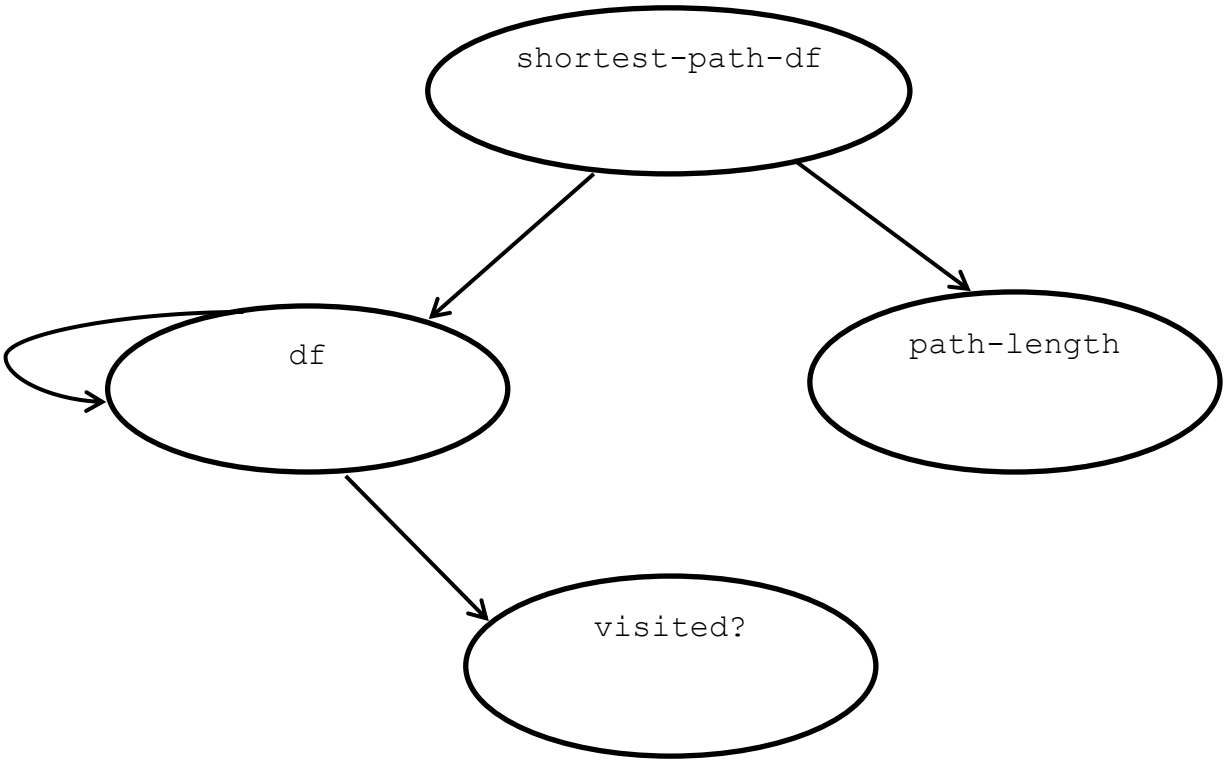


Рис. 3.2. Граф викликів тестової програми

Результати обчислення цикломатичної та рекурсивної складностей для тестової програми подано в таблиці 3.5.

Таблиця 3.5.

Результати обчислення цикломатичної та рекурсивної складностей

| Назва метрики | Назва функції | Результат |
|---|---|---|
| Цикломатична складність програми | `cyclomaticComplexity` | 13 |
| Рекурсивна складність програми | `recursiveComplexity` | 1 |
| Цикломатична складність функції `df` | `cyclomaticComplexity` | 7 |

Матриця залежностей «використання-визначення» представлена в таблиці нижче.

Таблиця 3.6.

Матриця залежностей «використання-визначення» тестової програми

| | `*visited*` | `*solutions*` | `visited?` | `df` | `path-` | `shortest-path-df` |
|---|---|---|---|---|---|---|

| | | | | | length | |
|---|---|---|---|---|---|---|
| *visited* | 0 | 0 | 1 | 1 | 0 | 0 |
| *solutions* | 0 | 0 | 0 | 1 | 0 | 2 |
| visited? | 1 | 0 | 0 | 1 | 0 | 0 |
| df | 1 | 1 | 1 | 0 | 0 | 1 |
| path-length | 0 | 0 | 0 | 0 | 0 | 0 |
| shortest-path-df | 0 | 2 | 0 | 1 | 2 | 0 |

Індекс підтримуваності програми дорівнює 66.5 згідно з результатом функції `maintainabilityIndex`.